



ONLINE PROBABILISTIC THEORY REVISION FROM EXAMPLES: A PROPPR APPROACH

Victor Augusto Lopes Guimarães

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientadores: Gerson Zaverucha
Aline Marins Paes Carvalho

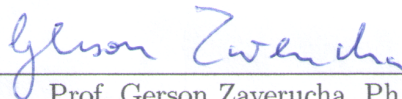
Rio de Janeiro
Março de 2018

ONLINE PROBABILISTIC THEORY REVISION FROM EXAMPLES: A
PROPPR APPROACH

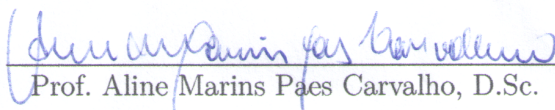
Victor Augusto Lopes Guimarães

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO
ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE
ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE
JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A
OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE
SISTEMAS E COMPUTAÇÃO.

Examinada por:



Prof. Gerson Zaverucha, Ph.D.



Prof. Aline Marins Paes Carvalho, D.Sc.



Prof. Valmir Carneiro Barbosa, Ph.D.



Prof. Fabio Gagliardi Cozman, Ph.D.

RIO DE JANEIRO, RJ – BRASIL
MARÇO DE 2018

Guimarães, Victor Augusto Lopes

Online Probabilistic Theory Revision from Examples:
A ProPPR Approach/Victor Augusto Lopes Guimarães. –
Rio de Janeiro: UFRJ/COPPE, 2018.

XI, 59 p.: il.; 29, 7cm.

Orientadores: Gerson Zaverucha

Aline Marins Paes Carvalho

Dissertação (mestrado) – UFRJ/COPPE/Programa de
Engenharia de Sistemas e Computação, 2018.

Referências Bibliográficas: p. 54 – 59.

1. online learning. 2. statistical relational learning.
3. theory revision from examples. 4. inductive
logic programming. 5. mining data streams. I.
Zaverucha, Gerson *et al.* II. Universidade Federal do Rio
de Janeiro, COPPE, Programa de Engenharia de Sistemas
e Computação. III. Título.

Agradecimentos

Primeiramente, agradeço aos meus pais Augusto e Lourdes, por todo o apoio e incentivos, assim como todo o investimento feito no meu desenvolvimento.

A Livia, por todo o suporte e atenção, e que acompanhou de perto todas as conquistas deste trabalho, assim como todas os desafios encontrados.

Agradeço aos meus orientadores: ao Professor Gerson Zaverucha, que aceitou ser meu orientador, acreditando no meu potencial; e a Professora Aline Paes, que me orientou desde o início da minha vida acadêmica até o final da minha graduação, e que, mais uma vez, aceitou me orientar neste trabalho. Sem eles este trabalho não seria possível. Agradeço ao tempo que eles dispuseram para a realização deste trabalho e ao conhecimento transferido a mim e gerado durante a orientação. Esses conhecimentos são de grande importância na minha formação.

Aos Professores Valmir Barbosa e Fabio Cozman, que aceitaram participar da banca desta dissertação, contribuindo com importantes comentários sobre o trabalho e ajudando a melhorar a mesma.

Aos professores do Programa de Engenharia de Sistemas e Computação que, direta ou indiretamente, contribuíram para minha formação. Em especial, ao Professor Valmir Barbosa por sua orientação durante o início de meu mestrado.

A equipe administrativa e de manutenção do Programa de Engenharia de Sistemas e Computação, por todo o suporte necessário para a conclusão do programa.

Aos meus colegas de curso e de orientação, pelo ótimo convívio durante o tempo do mestrado.

A todos meus familiares e amigos, por todo o apoio dado e pela compreensão pela minha frequente ausência.

A comunidade científica como um todo, principalmente aqueles cujos trabalhos são relacionados a esta dissertação.

Ao CNPq pelo suporte financeiro que viabilizou este trabalho.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

REVISÃO INCREMENTAL DE TEORIA PROBABILÍSTICA A PARTIR DE EXEMPLOS: UMA ABORDAGEM COM PROPPR

Victor Augusto Lopes Guimarães

Março/2018

Orientadores: Gerson Zaverucha

Aline Marins Paes Carvalho

Programa: Engenharia de Sistemas e Computação

A manipulação de fluxos de dados relacionais estruturados se tornou uma tarefa crucial, dada a disponibilidade de conteúdo produzido por sensores e pela Internet, como redes sociais e grafos de conhecimento. Esta tarefa é ainda mais desafiadora em um ambiente relacional do que em ambientes que lidam com exemplos i.i.d., dado que não podemos garantir que os exemplos são independentes. Além disso, a maioria dos métodos de aprendizado relacional ainda são projetados para aprender apenas a partir de conjuntos fechados de dados, não considerando modelos aprendidos em iterações anteriores de exemplos. Neste trabalho, nós propomos OSLR, um algoritmo de aprendizado relacional incremental que é capaz de lidar com fluxos de dados contínuos de exemplos, a medida em que eles chegam. Nós aplicamos técnica de revisão de teoria para aproveitar o conhecimento preliminar como ponto de partida, buscando onde o mesmo deve ser modificado para considerar novos exemplos e aplicando automaticamente essas modificações. Nós nos baseamos na teoria estatística do limitante de Hoeffding para decidir se o modelo, de fato, deve ser atualizado, de acordo com novos exemplos. Nosso sistema foi construído sobre a linguagem estatística relacional ProPPR, para descrever os modelos induzidos, visando considerar a incerteza inerente de dados reais. Resultados experimentais em bases de co-autoria e redes sociais mostram o potencial da abordagem proposta comparada com outros métodos de aprendizado relacional.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

ONLINE PROBABILISTIC THEORY REVISION FROM EXAMPLES: A PROPPR APPROACH

Victor Augusto Lopes Guimarães

March/2018

Advisors: Gerson Zaverucha

Aline Marins Paes Carvalho

Department: Systems Engineering and Computer Science

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 machine learning methods are still designed to learn only from closed batches of data, not considering the models acquired in previous iterations of incoming examples. In this work, we propose OSLR, an on-line relational learning algorithm that can handle continuous, open-ended streams of relational examples as they arrive. We employ techniques from theory revision to take advantage of the already acquired knowledge as a starting point, find 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 accordingly to the new examples. Our system is built upon ProPPR statistical relational language to describe the induced models, aiming at contemplating the uncertainty inherent to real data. Experimental results in entity co-reference and social networks datasets show the potential of the proposed approach compared to other relational learners.

Contents

List of Figures	ix
List of Tables	x
List of Algorithms	xi
1 Introduction	1
2 Background Knowledge	5
2.1 Logic and Statistics Fundamentals	5
2.1.1 First-Order Logic	5
2.1.2 SLD-Resolution	7
2.1.3 Inductive Logic Programming	10
2.1.4 Statistical Relational Artificial Intelligence	11
2.2 ProPPR	14
2.2.1 Language	14
2.2.2 Inference	15
2.2.3 Learning Parameters	17
2.3 Theory Revision from Examples	20
2.3.1 Revision Points	20
2.3.2 Revision Operators	20
2.4 Online Learning	21
2.5 Related Work	23
3 Online Learning of ProPPR Models Based on Theory Revision	26
3.1 Finding Revision Points	26
3.1.1 Revision Operators	28
3.2 Learning Algorithm	32
3.2.1 Leaf Potential Heuristic	33
3.2.2 Selecting Operators	34
3.2.3 Hoeffding’s Bound	35
3.2.4 Feature Generation	36

3.2.5	Applying the Revision	38
4	Experiments & Results	40
4.1	Simulating an Online Environment	40
4.1.1	Datasets	40
4.1.2	Methodology	42
4.1.3	Results	43
4.2	Batch Environment	49
5	Conclusions	51
5.1	Future Works	52
	Bibliography	54

List of Figures

2.1	SLD-Resolution Tree	9
2.2	ProPPR's Resolution Graph	17
3.1	Tree Structure	27
3.2	Bottom Clause Graph	31
3.3	Feature Selection Example	37
4.1	The Evaluation of the Cora Dataset Over the Iterations	44
4.2	The Evaluation of the UWCSE Dataset Over the Iterations	46

List of Tables

2.1	Kinship Knowledge Base	8
2.2	ProPPR Language Example	15
2.3	ProPPR Facts Example	17
2.4	The Relationships in Department	22
4.1	Size of the Datasets	41
4.2	Area Under the Iteration Curve of the Methods on Cora Datasets	44
4.3	Final Performance of the Area Under the Precision-Recall Curve of the Methods on Cora Datasets	45
4.4	Area Under the Iteration Curve of the Methods on UWCSE Dataset	47
4.5	Final Performance of the Area Under the Precision-Recall Curve of the Methods on UWCSE Dataset	47
4.6	Average Runtime for Each Learned Relation	47
4.7	AUC of the Alternative Methods on UWCSE Dataset	48
4.8	Final Performance of the Alternative Methods on UWCSE Dataset	48
4.9	AUC of the Alternative Methods on Cora Dataset	48
4.10	Final Performance of the Alternative Methods on Cora Dataset	49
4.11	Area Under the Precision-Recall Curve for the Same Paper Relation	49

List of Algorithms

1	Sequential Covering Algorithm	11
2	The PageRank-Nibble Algorithm	16
3	Theory Revision Algorithm	22
4	Algorithm to Place the Example in the Tree Structure	28
5	OSLR: The learning algorithm proposed in this work to learn from a stream of relational examples	33
6	Revise Leaves	34
7	Select the Best Operator for the Leaf	35
8	Apply Revision	38

Chapter 1

Introduction

Machine learning is a field of study concerned with creating systems that are able to learn how to perform a task by experience [1]. However, most of the machine learning algorithms are designed for a batch environment, i.e. that all the data (experience) are available at the beginning of the learning process.

The availability of sources that continuously generate data has aroused the need for specific machine learning 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 [2–5].

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 among the objects. However, real-world data that also present the continuous arrival behaviour, such as drug discovery [6], social networks [7] and knowledge bases built from texts [8], are heterogeneous, multi-related, uncertain and noisy.

Although there are languages and algorithms [9–13] that learn from them which 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) [14]. Two main components make up 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 [15], while addressing contributions from statistical and probabilistic models [16] 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 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 updated 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* [17–20] algorithms. Besides that, 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 [17, 19, 20].

Thus, an online relational 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 work, motivated by the need of continuously learning from relational, uncertain, and open-ended data, we propose the Online Structure Learner by Revision (OSLR), an algorithm that can learn and revise a StarAI model online. We built the system to learn models represented by ProPPR language [13] since its inference engine can efficiently handle inference on noisy relational data.

ProPPR language address the qualitative component of a StarAI language by the use of a first-order function-free logic language; while the quantitative component is represented by weights associated with the clauses in ProPPR language, which gives higher probabilities for proofs in the path of clauses with higher weights.

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 is not yet generalized could quickly lead to overfitting and impose high costs on the learning process. Thus, in this work we use Hoeffding’s bound [2, 21–24] to decide whether a revision improvement on the existing model is significant so that it should be implemented.

We have opted for using Hoeffding’s bound, instead of others statistical bounds, because Hoeffding’s bound does not make assumptions about the probability distribution of the random variable. However, this came at the cost that more examples must be necessary for a smaller bound if compared with bounds dependent of the probability distribution [2].

Furthermore, to avoid the bias caused by statistical dependences between relational instances [25], particularly because Hoeffding’s bound was originally devel-

oped to handle independent and identically distributed (i.i.d.) cases, we take extra care regarding the linkage and autocorrelation between instances. We use the intersection of atoms in the bottom clause [26] of the examples, which represents the atoms related to the example in the knowledge base, to define whether two examples are dependent on each other.

We compare OSLR¹ against batch StarAI learning methods, namely the RDN-Boost system, designed to learn Relational Dependency Networks (RDN) [27, 28], that has obtained the state-of-the-art results in a number of relational datasets; and against ProbFOIL [29] and Slipcover [30] which use the Sato’s semantic distribution [31] of possible worlds in order to perform probabilistic logic inference. We designed a simulated online environment to emulate what would happen in the real world when new examples appear incrementally.

Additionally, we also present competitive results against HTilde [23, 32], 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 main contribution of this work is the Online Structure Learner by Revision (OSLR), an algorithm that learns probabilistic logic theories online, which can be summarised in the following parts:

- In order to infer probabilistic theories we have used the ProPPR system [13], which uses a PageRank [33] based algorithm to give different probabilities of reaching different proofs of a logic query;
- To perform the online learning, while still considering the previously learned model, we have applied techniques from theory revision from examples [17–20], which considers a previous learned model as starting point for a new learning step;
- The learning step is performed when new examples arrive and is effectively applied when a significant improvement over the current theory is observed. This significant improvement relies on Hoeffding’s bound statistical theory [21], which takes the number of independent examples observed into account;
- Since Hoeffding’s bound assumes independent examples, we take some extra care in order to account for the number of examples, assuming that a dependence exists between two examples when they share a common atom on their bottom clauses [26].

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

In this work, we would like to answer two main research questions, regarding learning relational models in an online environment:

- Q1** Is it better to modify a previously learned model or to learn a new model from scratch, in order to cope with new examples, **regarding the quality of the learned model?**
- Q2** Is it better to modify a previously learned model or to learn a new model from scratch, in order to cope with new examples, **regarding the time for learning the model?**

In order to answer these questions, we have experimented the systems on two domains: UWCSE [10] and Cora [34]. The performance of OSLR is comparable to the RDN-Boost system on the first domain and it outperforms the RDN-Boost system on the later. Furthermore OSLR outperforms ProbFOIL and Slipcover on both datasets. The average runtime of the experiments shows that our system is faster than the RDN-Boost on most of the learning task. These experiments make us answer positively both of the proposed questions.

Another advantage of our online learning system is that it is always ready to make a prediction, considering the most recent examples, while a batch system would need to be retrained in order to generate an updated model.

Additionally to those questions, since we apply theory revision techniques, we also investigated the benefit of starting the learning algorithms with an initial human made theory for the UWCSE dataset. Our experiments have shown that, despite the human theory being only partially correct, OSLR was able to take advantage of this theory to create a better model (**Q1**) in less time (**Q2**).

The remainder of the dissertation is organized as follows: we first give the background knowledge to understand our work on Chapter 2, finishing the chapter with the works related to ours in Section 2.5; then we present our proposal in Chapter 3 followed by the performed experiments in Chapter 4. Finally, we conclude and give some directions for future works in Section 5.

Chapter 2

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 [13]. In this chapter, first, we will give the logic and statistic fundamentals to understand this work and then we explain ProPPR's language and its inference engine; after that, we describe the key concepts related to theory revision [35, 36]; and finally, we give an overview of the related works regarding online learning from relational examples.

2.1 Logic and Statistics Fundamentals

In this section we will present the logic fundamentals necessary to a better understanding of this work. In addition we will present some possible ways to combine logic with statistic and probabilistic approaches.

2.1.1 First-Order Logic

First-order logic is a formal language that allows for representing knowledge and reasoning about this knowledge [37]. First-order logic describes a world consisting of objects and the objects may relate with each other.

The main symbols of first-order logic are:

- constant and variables;
- predicates;
- functions;
- atoms and literals;
- connectives;
- quantifiers.

We will describe each one of them in more details.

The constants are used to represent objects from the worlds. Variables are terms that may be substituted by constants or function terms in order to conform with a desired goal; for instance, to answer a query about which constants may have a specified relation. Predicates are used to define relations between objects or properties of an object. A predicate is represented by a *name* and have an arity n , which is the number of terms an atom of this predicate must have; the predicate is usually referred as *name/n*.

A function, as usual in mathematics, is a map that relates a set of inputs to an unique output and is represented as a symbol followed by a n-tuple of terms between brackets. A term is a constant, a variable or a function applied to a set of terms.

An atom express the relation between objects (or a property of an object) through a predicate symbol. It is represented by the name of the predicate followed by an n-tuple of terms between brackets, where n is the arity of the predicate. For instance, the atom below, which states that *john* is married to *mary*.

$$isMarriedTo(john, mary) \tag{2.1}$$

This atom is composed by: a predicate *isMarriedTo*, which represents the married relation; and the constants *john* and *mary*, which represents people (objects) of the real world. In this case, the relation has arity 2, which is also called a binary relation. It is also possible to have more than a predicate with the same name, but varying in its arity. The atom may be either *true* or *false*, depending on whether it holds or not in the described world. In this way, the predicate represents the relation that might happen between objects and the atom express such a relation for a set of objects.

A literal can be either an atom or the negation of an atom. In the case it is a negation of an atom, it is true whenever the atom is false, or have failed to be proved.

In addition, we may use connectives to connect atoms in order to build formulas. An atom is a formula. A connection of formulas is also a formula. The connectives below may be used to build more complex formulas.

Considering the formulas A and B , the connectives are:

- \neg (negation), which states that $(\neg A)$ is true whenever A is false;
- \wedge (and) which states that $(A \wedge B)$ is true whenever both A and B are true;
- \vee (or) which states that $(A \vee B)$ is true whenever at least one of A or B is true;
- \rightarrow which states that $(A \rightarrow B)$ is true whenever A is false or both A and B are true, intuitively if A is true, B must also be true;

- \leftrightarrow which states that $(A \leftrightarrow B)$ is true whenever A and B have the same logic value;

Finally we have the quantifiers, that are useful to express properties about collections of objects. The quantifiers are used to quantify variables. There are two types of quantifiers, the *existential* quantifier (\exists) and the *universal* quantifier (\forall).

The quantified formulas use variables to express generic concepts about the objects in the world and is very useful for reasoning. For instance, the formula below states that every human is a mammal.

$$\forall x \text{ human}(x) \rightarrow \text{mammal}(x).$$

While the formula below says that exists, at least, a mammal that flies.

$$\exists x \text{ mammal}(x) \wedge \text{fly}(x).$$

When using first-order logic on computational problems it is often restricted to a subset language constituted by Horn clauses [38]. A Horn clause is a disjunction of literals with at most one positive literal where all variables are universally quantified as the example below:

$$\forall x_1 \dots x_n H \vee \neg B_1 \vee \dots \vee \neg B_m$$

Where x_i are variables that appear in the clause. A Horn clause is commonly written in its implication form, omitting the quantifiers:

$$H \leftarrow B_1 \wedge \dots \wedge B_m$$

In this form, H is called the head of the clause and $B_1 \wedge \dots \wedge B_m$ is called the body. A Horn clause with exactly one positive literal is called a definite clause. When the body is empty, we call it a fact. We will also call a definite clause as a rule.

In this work, we will use definite clauses and facts to represent the knowledge.

2.1.2 SLD-Resolution

An advantage of having the knowledge described by a formal logic language is that now we are able to reason about this knowledge. Since the knowledge is composed by Horn clauses, the most usual way of reasoning about it, we use the Selective Linear Definite clause resolution, or simply, SLD-Resolution.

The SLD-Resolution allows us to query the data by goal. A goal is a headless Horn clause, that we would like to answer if it can be proved, given a knowledge

Table 2.1: Kinship Knowledge Base

(f_1)	$father(rick, ned) \leftarrow$
(f_2)	$father(ned, robb) \leftarrow$
(f_3)	$father(ross, kate) \leftarrow$
(f_4)	$mother(kate, robb) \leftarrow$
(c_1)	$parent(X, Y) \leftarrow father(X, Y)$
(c_2)	$parent(X, Y) \leftarrow mother(X, Y)$
(c_3)	$ancestor(X, Y) \leftarrow parent(X, Y)$
(c_4)	$ancestor(X, Y) \leftarrow parent(X, Z) \wedge ancestor(Z, Y)$

base, and which values the variables of the goal might assume in order to prove it. Where a knowledge base is a set of definite clauses and facts.

To understand the SLD-Resolution, consider the kinship knowledge described on Table 2.1. In this table we have some facts about mother and father relations, and we have rules defining the concept of parent and ancestor.

Supposing we would like to know *Robb's* ancestors, we could represent this as the following query:

$$\leftarrow ancestor(X, robb).$$

In other words, it is asking which constants may replace the variable X , such that X is an ancestor of Robb.

To answer this query we start a SLD-Resolution tree with the query as the root, which represents a list of goals that must be proved, in this case, a single goal: $ancestor(X, robb)$. Then, for each goal in the list, we try to solve it by applying a clause that has an atom on its head which can be unified with the goal. Two atoms are unifiable when there is a variable substitution that, when applied to both, lead to the same substituted atom, by replacing each variable of the atoms for another term.

In order to unify the atoms, the *most general unifier* is used. The most general unifier is, roughly speaking, the simplest variable substitution that makes both atoms equal. The variables substitution replaces each variable in a formula by a term, simultaneously. The substitution is applied to the whole clause, including its body.

When a clause is used to solve a goal, we create a node, child of this one, by removing the goal from the list and append, at the beginning of the list, the body of the used clause, when it exists. When the list of goals is empty, we say we found a solution, when there are no more possible clauses to be applied, we say the path fails to prove the query. This algorithm is usually performed on a depth-first order.

In this way, each solution is associated with a (possibly empty) substitution, when the substitution is applied to the query, it provides the de-

sired answers, in this case the ancestors of *robb* are: *ned*, *kate*, *rick* and *ross*. In this way, we say the knowledge in Table 2.1 entails (\models) the facts: $ancestor(ned, robb)$, $ancestor(kate, robb)$, $ancestor(rick, robb)$ and $ancestor(ross, robb)$; this is, even not explicitly represented in the table, the truth values of these facts are implicit by the truth values of the clauses c_3 and c_4 [37].

In Figure 2.1 we can see a subset of the SLD-Resolution tree for the problem mentioned above. In each edge we can see the clause used to solve the goal and the substitution θ used to unify the atoms. The underlined goals are the ones to be solved at the time. We omitted some variables renaming substitutions in order to make the figure clear. The nodes with a \square represent the solutions, emphasising the desired value of X , which represents the name of Robb's ancestor, while the \times node represents a failure.

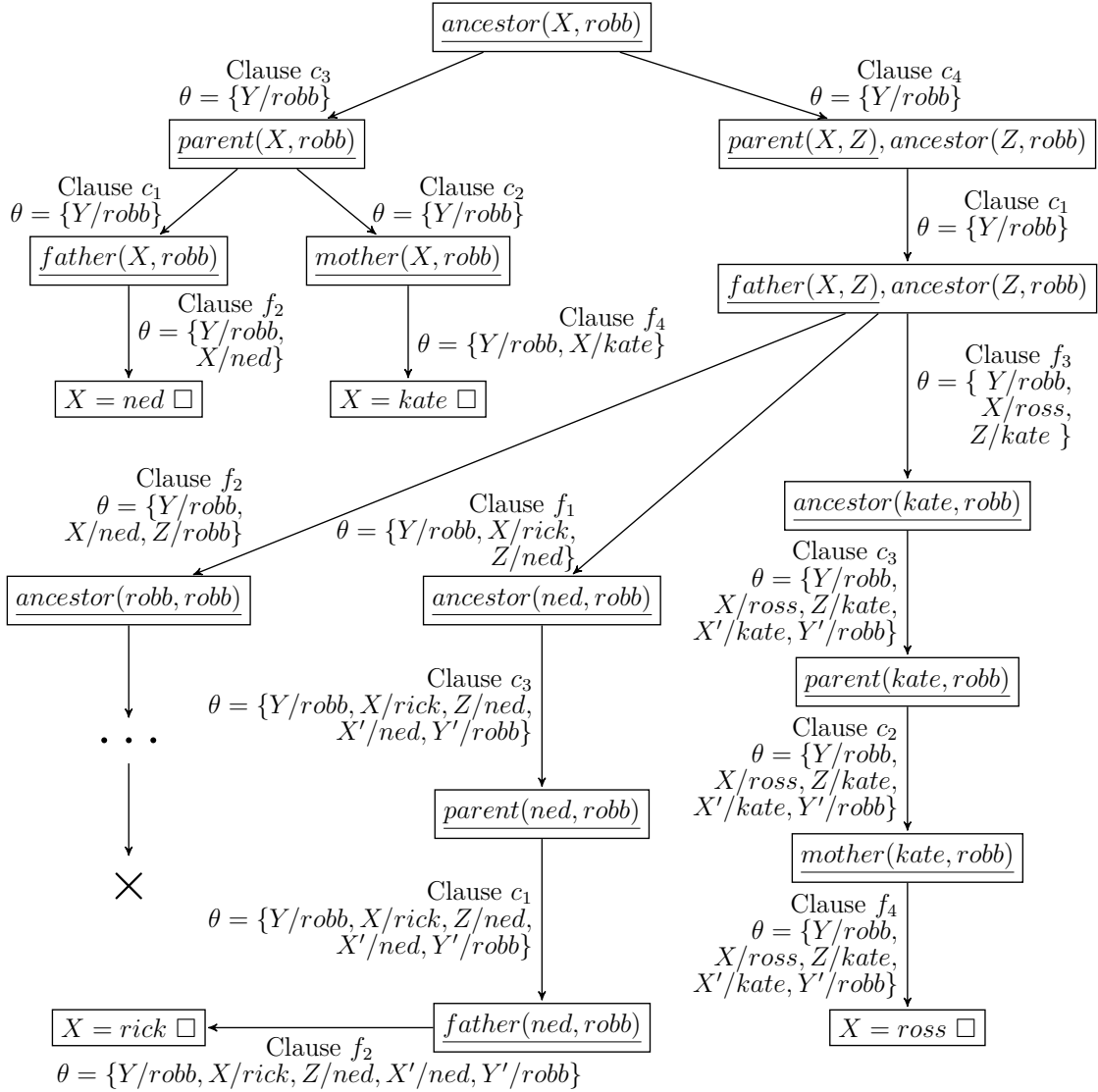


Figure 2.1: SLD-Resolution Tree

SLD-Resolution is the proof mechanism behind a Prolog interpreter [39], which

is proved to be complete for a set of Horn clauses. For a more detailed overview on first-order logic and SLD-Resolution we refer the reader to [40].

2.1.3 Inductive Logic Programming

Inductive Logic Programming (ILP) is a subset of machine learning and programs induction that combines the knowledge representation with the ability of learning from examples from machine learning [40]. The main concern of ILP is to induce rules that are able to prove a set of positive examples without proving a set of negative examples for a given background knowledge.

Formally, given *background knowledge* (BK) defined as a set of definite clauses and facts; and a set of examples $E = E^+ \cup E^-$, defined as facts, where E^+ represents the set of positive examples and E^- represents the set of negative ones; we would like to find a hypotheses H , which is a finite set of definite clauses (also called *theory*), such that it conforms with the following constraints:

1. $BK \wedge H \models E^+$, this is, H is *complete*; and
2. $BK \wedge H \not\models E^-$, this is, H is *consistent*.

When the hypotheses H conforms with both the constraints, we say that H is *correct* [40].

Usually, it is not possible to find a hypotheses that satisfies these both constraints, thus, we relax the constraints and try to find a hypotheses as close as possible to *correct*.

A simple way to find a theory hypotheses is the Sequential Covering Algorithm shown on [1] and described in Algorithm 1. This algorithm starts from an empty theory and find rules that prove as much as possible the positive examples avoiding proving the negative ones. The rules whose performance, arbitrarily defined by the function *performance*, is above a given threshold τ are inserted into the theory and the examples proved by this rule are removed from the set of examples. In the end, we have a theory that covers as much as possible all the positive examples, if the threshold τ is small enough. The sequential covering name arises since we pass through each uncovered (not proved) examples and tries to prove it with a new rule.

Algorithm 1 is a *bottom-up* approach from the point of view of learning logic theories, it starts from an empty theory, which is the most specific possible theory, not proving any example, and generalizes it at each step by adding a new rule that might prove examples not proved before. Besides the *bottom-up* approach, there is also the *top-down*, which is the other way around, starting from an overly generic theory and specifying it as needed. There are also methods that mix these both approaches, generalizing and specifying the theory in order to better describe the

Algorithm 1 Sequential Covering Algorithm

Input:

A Background Knowledge (BK);
A set of examples (E);
An improvement threshold (τ);

Output: A Theory;

```
function Sequential-Covering( $BK, E, \tau$ )  
   $theory \leftarrow \emptyset$   
   $rule \leftarrow learnOneRule(BK, E)$   
  while  $performance(rule, BK, E) > \tau$  do  
     $theory \leftarrow theory \cup rule$   
     $E \leftarrow E \setminus \{\text{positive example proved by the rule}\}$   
     $rule \leftarrow learnOneRule(BK, E)$   
  end while  
return  $theory$   
end function
```

examples. This concept of *bottom-up* and *top-down* may be analogously applied to the rule learning level.

Since the space of hypotheses of ILP systems is very large, possibly containing each possible combination of literals and their terms, it is common for ILP systems to require for a language bias to define which hypotheses are possible and which are not. This language bias is usually represented in the form of modes, which specifies which literals may appear in the heads and bodies of the rules, and which variables may be new or must already be in the rule.

The mode declarations may improve both the quality of the learned theory and the computation time of the learning algorithm, by avoiding the test of bad hypotheses, but it came at the cost of a good knowledge of both the domain of the task at hand and the declaration language of the system used. A too restrictive mode declaration or a wrongly biased one may degrade the performance of the learned model, by discarding potentially good hypotheses. Thus, a balance between the restrictiveness of the language bias and the search time of the algorithm might be found in order to find good hypotheses in feasible time [40].

2.1.4 Statistical Relational Artificial Intelligence

Despite the expressive power of logic for both representing knowledge and reasoning about it, it has a limitation: its proofs are boolean, i.e. either true (prove) or false (fails to prove). This limitation makes it difficult for ILP to deal with noise and uncertainty, two aspects inherent in real world problems.

In order to overcome this limitation, several approaches have been proposed in the last decades, many of them trying to combine the well founded fields of

Probability and Statistics with logic.

Several approaches have been proposed with different ways of combining logic and probabilities and thus many names have been used to refer to this field of study: Probabilistic Logic Learning and Statistical Relational Learning are some of them, and there are disagreements about which framework belongs to which field. For the concern of this work, we will use Statistical Relational Artificial Intelligence (StarAI) as a generic term in order to refer to any system that combines logic with statistics and probability [14].

There are two main approaches to combine logic with probability: First-Order Logic and Probabilistic Graphical Models, each of which with their own methods. On one hand, Probabilistic Logic Programming aggregates probability to the logic reasoning by labelling the rules (possibly also the facts) with weights that will be used to give different probabilities to different proofs. On the other hand, Probabilistic Graphical Models uses the relational expressiveness of logic language to build graphical models and performs the inference in the graphical model.

In the following sections we will describe two well-known frameworks from the First-Order Logic approach: Probabilistic Logic Programming and Stochastic Logic Programming which are closely related to this work, specially the later; and then, will give a general overview of Probabilistic Graphical Models.

Probabilistic Logic Programming

A well-known way of combining logic with probability is the field known as Probabilistic Logic Programming (PLP). PLP is a family of languages which usually uses Sato's distribution semantics [31], which defines a probability over possible *models* (or worlds). The program consists of a set R of definite clauses and a set F of ground facts (that do not contain variables). It considers that each fact $f_i \in F$ is associated with a probability value $p_i \in [0, 1]$ and represents an independent boolean random variable that is true with probability p_i and false with probability $1 - p_i$ [41]. Each model M is a set of ground facts and has a probability defined by the equation below:

$$\Pr(M) = \prod_{f_i \in M} p_i * \prod_{f_i \in F \setminus M} (1 - p_i)$$

In this semantic, the probability of a specific fact to be true is given by the sum of the probabilities of the worlds in which the fact is true.

A prominent system that uses this semantics is ProbLog [11] which is a probabilistic extension of Prolog [39]. ProbLog uses binary decision diagrams in order to perform, efficiently, the inference for this semantics. ProbLog's language also allows one to specify probabilities for definite clauses, in addition to grounded facts. We refer the reader to [41] for more details about probabilistic logic concepts.

Stochastic Logic Program

Another approach that assigns confidences to logic proofs is Stochastic Logic Program (SLP) [9]. Instead of defining probability facts that may or may not be true in possible models, Stochastic Logic Program is a language which states that each clause has a label $p \in [0, 1]$ meaning the probability of following through such a clause in the SLD-Resolution tree. In this way, different edges in the SLD-Resolution of a given query may have different weights, thus, different solutions may be preferred.

Assuming that a node in the SLD-Resolution has n edges with weights p_1, \dots, p_n , respectively, the probability of choosing the edge is represented by p_i is $\frac{p_i}{p_1 + \dots + p_n}$. The probability of a given solution is the product of the probability in the path from the root to the solution. Since a solution may occur repeatedly in different leaves of the SLD-Resolution tree, the probability of those leaves are summed.

In this way, SLP defines a probability distribution over the proved facts. This formalism is the one closest related to the ProPPR language used in this work, as we will see in Section 2.2.

Probabilistic Graphical Models

A Probabilistic Graphical Model is a graph-based representation to compactly encode a, potentially complex, joint probability distribution over a set of random variables [42]. The random variables in the domain are represented by the nodes in the graph, while an edge between two nodes represents that there is a direct probability interaction between the two random variables represented by these nodes.

There are models that use directed graphs, such as Bayesian networks, which is an acyclic directed graph and others that use undirected edges, such as Markov networks [42].

In order to account for the relational nature of the logic programs, while still reasoning under uncertainty, some approaches have been proposed. Two prominent approaches in this field are Markov Logic Network (MLN) [10] and Relational Dependency Network (RDN) [12].

Markov Logic Network uses first-order logic in order to describe how to build a grounded Markov network. Each clause in the first-order language has a real-valued weight which states that as higher the weight of the clause, possible worlds that violate this clause are less likely to be true. Inferences in such a language are performed in the constructed Markov network. MLNs algorithms also allow for learning the weights of the first-order clauses (parameter learning) and even learn the clauses itself using ILP techniques (structure learning) [10].

Relational Dependency Network is an extension of Dependency Networks (DN) [43] for relational data [12]. DN approximates a joint probability distribution with a set of Conditional Probability Distributions (CPD) independently learned from data. RDNs extends DN by specifying a joint probability distribution using CPDs over the attribute values of relational data.

In this work we opted for using ProPPR because of its efficiency. ProPPR is similar to the SLP language, but performs the inference based on an approximation of the PageRank algorithm in a SLD-Resolution graph, as we explain in the next section. Nevertheless, we believe that StarAI models learned by first-order logic paradigm, which aggregates probability to the logic reasoning, are easier to be interpreted and explained by humans, than the probabilist graphical model paradigm.

2.2 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) [13].

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

A knowledge base written in ProPPR’s language can be composed of grounded atoms (atoms that have only constants in their terms) and definite clauses, following the same definitions described in Section 2.1.

Features

In addition to the logic part, ProPPR language allows for specifying features for the rules. The features are in the form of atoms and, when defined, they appear

Table 2.2: ProPPR Language Example

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

at the end of the rule between curly braces. The role of those features is related to the weights parameters and will become clear in the following two subsections, where we will explain the ProPPR’s inference method and parameter learning. Table 2.2 shows an example of a set of rules in the ProPPR language. This example is inspired by the example in [13] and represents the domain of webpages, where we would like to learn the topic of a webpage, based on its words and its hyperlinks. The underlined part has no logical utility; it is only there to define the features.

2.2.2 Inference

ProPPR defines logical inference as a search on a graph, similar to the one produced by a Prolog interpreter [39]. Let P be a logic program that contains a set of rules $C = \{c_1, \dots, c_n\}$. 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 ; then add new nodes to G such as: let $(Q, (R_1, \dots, R_k))$ be represented by a node u , and $c \in C$ be a rule of the form $R' \leftarrow S'_1, \dots, S'_l$, 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 $v = (Q\theta, (S'_1, \dots, S'_l, R_2, \dots, R_k)\theta)$. $(S'_1, \dots, S'_l, R_2, \dots, R_k)$ 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; since the application of different rules (or atoms) to solve a subgoal may lead to the equal nodes, these nodes are represented by a single node in the graph structure.

Figure 2.2 shows a subset of a graph constructed by ProPPR given the query $about(a, L)$, the set of facts described in Table 2.3, and the program defined in Table 2.2. The nodes with a \square 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 [13], performed on the graph described above, via an algorithm called PageRank-Nibble [44, 45]. This algorithm is used to implement a probabilistic SLD-Resolution and is outlined as Algorithm 2 [13].

The algorithm works by maintaining two vectors \mathbf{p} and \mathbf{r} with the same size as the number of nodes in G , where \mathbf{p} represents the PageRank approximation, and \mathbf{r}

Algorithm 2 The PageRank-Nibble Algorithm

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

```
function push( $u$ ) ▷ this function modifies  $\mathbf{p}, \mathbf{r}$  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  
       $\mathbf{r}[v] \leftarrow \mathbf{r}[v] + \text{Pr}(v|u) * \mathbf{r}[u]$   
    else  
       $\mathbf{r}[v] \leftarrow \mathbf{r}[v] + (\text{Pr}(v|u) - \alpha') * \mathbf{r}[u]$   
    end if  
  end for  
end function
```

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 $r[v_1] \dots r[v_n]$, based on $\text{Pr}(v_i|v)$, that represents the probability of going from node v to v_i , where $v_i \in N(v)$ represents the neighbours 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 .

The order on which the nodes are chosen does not matter for the analysis of the algorithm, but it does matter for inference of logic programs; ProPPR follows the same order that a Prolog interpreter would, or an order as close as possible of it [13].

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 solution 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.

Table 2.3: ProPPR Facts Example

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

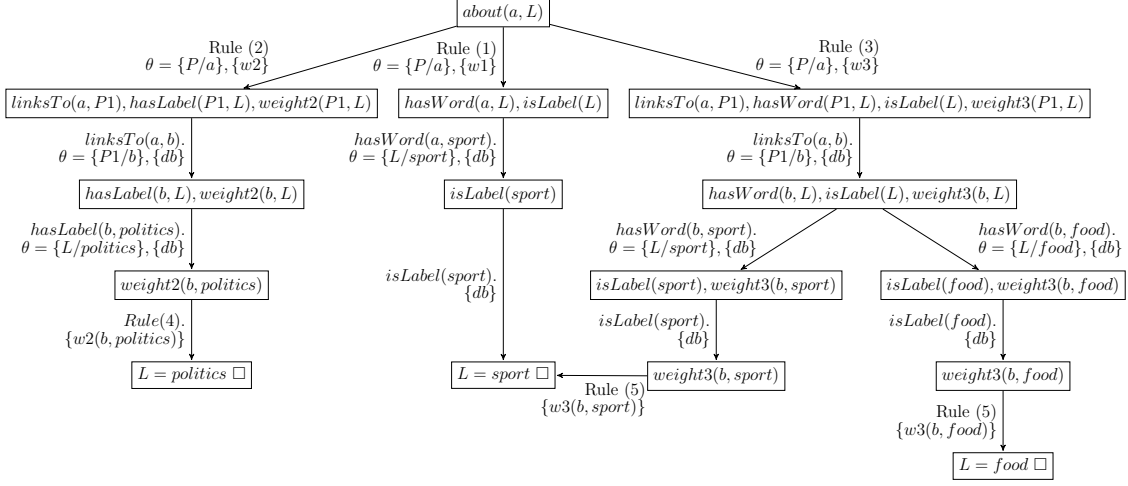


Figure 2.2: ProPPR's Resolution Graph

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 [46]. 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 a solution node, since there are more ways to get to it.

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. For instance, lets assume that the probability of getting the edge returning to v_0 , at each step, is $\alpha \in [0, 1]$. Then, we have that the probability of reaching a node that is d steps away from v_0 is $(1 - \alpha)^d$. The bias to short proofs is desired at least for three reasons: (1) given the Occam's razor [1], short explanations, represented by short proofs, are more likely to be the correct ones; (2) they are easier to compute; and (3) they are easier to be interpreted by humans.

2.2.3 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 on each other, ProPPR defines an example as composed of a query of the type $p(X_1, \dots, X_n)$, where p is a predicate, $n \geq 1$, and $X_i \in X_1, \dots, X_n$ is either a

constant or a variable, and at least one term $X_i \in X_1, \dots, X_n$ 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:

$$\textit{about}(a, X) + \textit{about}(a, \textit{sport}) + \textit{about}(a, \textit{politics}) - \textit{about}(a, \textit{food})$$

The first atom is the query and the others represent possible answers by grounding X .

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 a unique feature for it. These features are used to change the probability $\Pr(v|u)$ of going from node u to node v , 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.

ProPPR, different from other StarAI systems such as ProbLog [11] and MLN [10], is capable of learning different weights for different constants instantiated by the rules during the inference, rather than only a fixed weight for the rule itself. This gives more flexibility to tune this weights, that is the role of the rules (4 – 5) in Table 2.2. From a logic point of view, these rules have no meaning, since it proves anything; but for the ProPPR inference mechanism, it creates specific weights for the variables on their features, depending on the constants used to instantiate them.

It is important for the variable of the features to appear in the head of the rule because, when the rule is used to prove the goal, the variable must have been already replaced by the constant defined by the applied substitution. Since the goal it proves is only used at the end of the rule and the variables of such a goal have already appeared before in the rule (see the underlined parts of clauses (2 – 3) in Table 2.2) the variables will be already replaced by constants by the time the SLD-Resolution tries to prove the last literal of the rule.

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.

Stochastic Gradient Descent (SGD) is an optimization process that aims at finding the parameters that minimize a given loss function, with respect to a set of examples. To achieve this goal, at each step it changes the parameters of the functions following the inverse direction of the gradient, based on a, randomly chosen, batch of examples. The method iterates over the whole set of examples performing a batch at a time until it reaches a stop criteria, which, in our case is to pass through each one of the examples n times.

In the ProPPR's case, the loss function is the Cross-Entropy, which is defined by the equation below:

$$-\frac{1}{n} \sum_{i=1}^n [y_i \ln(y'_i) + (1 - y_i) \ln(1 - y'_i)]$$

Where y_i is the correct value of the answer i , 0 for *false* and 1 for *true*; y'_i is the predicted value for the proof of the answer i ; and n is the number of answers.

Intuitively, the first term, inside the summing, concerns the positive answers, penalizing the prediction as far it is from 1; while the second term concerns the negative answers, penalizing the predictions as far it is from 0. By optimizing this function, we find the parameters that make the desired predictions as close as possible to 1 when it is a right prediction or to 0 when it is wrong.

In addition to the Cross-Entropy, ProPPR adds a *L2 Regularization* term to the loss function. The L2 Regularization is a regularization technique that adds the sum of the square of the parameters to the loss functions, avoiding the absolute value of the parameters to be tuned indefinitely, in order to prevent overfitting. The equation below show the Cross-Entropy with the L2 Regularization term:

$$-\frac{1}{n} \sum_{i=1}^n [y_i \ln(y'_i) + (1 - y_i) \ln(1 - y'_i)] + \lambda \sum_{w \in W} w^2$$

Where W is the set of parameters to be learned and λ a parameter to regulate the impact of the regularization in the comparison to the loss function.

This regularization term makes the gradient decay the absolute value of the weight towards zero, so a balance must be found between the absolute value of a parameter and its contribution in reducing the loss function, avoid overfitting.

In this way, the current theory, and its parameters, may be used to predict new answers through logic inference. We refer the reader to [13] for more details about ProPPR.

2.3 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 [35, 47]. 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 applying the proper change to these points, using a number of *revision operators*; and (4) deciding if the changes are going to be applied. Next, we briefly explain the concepts of *revision points* and *revision operators*.

2.3.1 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, by performing the logic inference and finding the rules that were used in the proof of negative examples or the literals that prevents positive examples to be proved, given the misclassified examples; to finally apply the respective revision operators to these points.

2.3.2 Revision Operators

Revision operators modify the theory, at some specific revision point, in order to improve the theory to a set of examples. The four most common theory revision operators are:

1. **Add rule:** inserts a rule into the theory aiming at proving positive examples, either taking advantage of an existing clause, or creating one from scratch;
2. **Delete antecedent:** erases literals, from the body of a rule, that prevents positive examples of being proved;
3. **Delete rule:** erases a rule taking place in the proof of a negative example;
4. **Add antecedent:** includes literals in the body of a rule, so that it becomes more specific to avoid incorrectly proved 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 previously 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 appropriated 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 [17].

An overview of a theory revision algorithm, similar to the one in [17] can be seen in Algorithm 3. At each step, the algorithm: finds the revision points of the theory, where the theory misclassifies any example; applies the revision operators to generate new theories; and finally, keeps the theory that better improves the performance on an arbitrary metric. The algorithm keeps repeating these steps until no more changes are able to improve the current theory, returning the best theory found in the process.

2.4 Online Learning

Traditional machine learning algorithms are batch learners, this is, they assume that all the information is available at the beginning of the learning process. If new information arrives, in order to be considered by the learned model, the batch algorithm must be retrained from scratch. To overcome this limitation, several online learning algorithm have been proposed [2–5].

An online learning algorithm learns over a stream of data, thus, when new data arrives, it must be able to adapt its model in order to consider the new information [2].

Since theory revision from examples assumes an, not necessarily correct, initial model, it becomes well suited for learning logic theories online. In this way, when

Algorithm 3 Theory Revision Algorithm

Input:

A Background Knowledge (BK);
A Theory ($theory$);
A set of examples (E);

Output: A (possibly modified) Theory

```
function Revise( $BK, theory, E$ )
   $newTheory \leftarrow theory$ 
  do
     $theory \leftarrow newTheory$ 
     $E' \leftarrow findMisclassifiedExamples()$ 
     $revisionPoints \leftarrow generateRevisionPoints(BK, theory, E')$ 
     $candidates \leftarrow \emptyset$ 
    for each  $rp \in revisionPoints$  do
      if isGeneralization( $rp$ ) then
         $revisions \leftarrow applyGeneralizationOperators(BK, theory, E, rp)$ 
      else
         $revisions \leftarrow applySpecializationOperators(BK, theory, E, rp)$ 
      end if
       $candidates \leftarrow candidates \cup revisions$ 
    end for
     $newTheory \leftarrow pickTheBest(candidates)$ 
  while  $performance(newTheory) > performance(theory)$ 
return  $theory$ 
end function
```

new data arrives, we may evaluate the current model, find the revision points and apply the revision operators in order to better fit the model to the new data.

For example, consider the problem of learning the advisory relations between people in an university department, inspired by the UWCSE dataset [10]. The knowledge base in Table 2.4 describes the relationships in an university department and we would like to learn the relation $advisedBy(x, y)$, that states that the person x is advised by the person y .

Consider that we have the following positive examples to learn the advisedBy relationship:

$$advisedby(person2, person1).$$

Table 2.4: The Relationships in Department

$publication(title1, person1).$	$publication(title2, person3).$
$publication(title1, person2).$	$publication(title2, person4).$
$taughtby(course1, person1, winter).$	$taughtby(course2, person3, spring).$
$ta(course1, person2, winter).$	$student(person4).$
$professor(person1).$	
$student(person2).$	

advisedby(person4, person3).

Both ILP learning algorithms and theory revision algorithms could easily learn the following rule:

$$advisedBy(X, Y) \leftarrow publication(Z, X) \wedge publication(Z, Y). \quad (2.2)$$

Which would correctly prove both of the positive examples.

Now, consider that we have a new information that a person cannot be advised by him/herself, which is very reasonable. The Rule 2.2 becomes too generic, since it proves the following negative examples:

advisedby(person1, person1).
advisedby(person2, person2).
advisedby(person3, person3).
advisedby(person4, person4).

In order to account for this new information, a batch ILP learning algorithm would discard the already learned Rule 2.2, which is partially correct (since it proves the positive examples) and relearn everything from scratch.

On the other hand, a theory revision algorithm could start from the already learned Rule 2.2 and specialise it, by adding the literals *student(X) ∧ professor(Y)* at the end of the rule, becoming:

$$advisedBy(X, Y) \leftarrow publication(X, Z) \wedge publication(Y, Z) \\ \wedge student(X) \wedge professor(Y).$$

And thus, not proving the negative examples while still proving the positive ones.

It is already known that starting from a previous, yet partially correct, theory may improve the quality of the model even using less examples [17, 19, 20].

Analogously, we can apply this capacity of starting from a not empty theory to online learning, considering as the “initial theory” the one learned at a previous moment of the learning process, which is subjected to be revised as new information arrives.

The use of theory revision brings another advantage, even when applied to online learning: it allows the use of previous acquired theories, either provided by another learning system or by human experts.

2.5 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 [17]. 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 [19], we share the idea of using a Bottom Clause [26] to support the generation of new literals. However, we do not employ stochastic search here as the FORTE-MBC successor, YAVFORTE [20]. 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 Hoeffding’s bound [21]. The Hoeffding’s bound has already been used in a classical machine learning method, yielding the VFDT [2] algorithm, which is a system to learn decision trees incrementally, by applying Hoeffding’s bound to decide whether to split a node, based on the information gained from the examples. Also, Hoeffding’s bound has been applied to learn massive data streams from relational databases [48]. 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 this work is HTilde, introduced in [23]. HTilde learns relational decision trees, where the test of a node is the proof of a logic literal. It grows the decision tree online, using the Hoeffding’s bound to decide whether to split a node, given the number of examples. In [32] this algorithm was enhanced, and it generated new results. HTilde was implemented over Tilde [49, 50] to incrementally learn first-order logical decision trees by using 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 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 on 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 [13] that is a StarAI system based on an approximation of the PageRank algorithm [33] 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 [10] or ProbLog [11] because of its efficiency on both answering the logic queries and learning the weight of the desired answers.

A well-known first-order logic system is FOIL [51]. FOIL learns first-order logic theories by starting from the most general clause and specifying it in order to better fit the examples, adding this clauses to the theory in a greedy way [51]. FOIL was extended in [29] to deal with probability, by using ProbLog [11] as language and inference engine. ProbFOIL is related to our work because it also learns probabilistic logic theories but in a different inference semantics, in their case, the Sato’s distribution semantics [31].

Another work that learns probabilist logic theories under this semantics is Slipcover [30]. Slipcover uses a beam search to find potentially good clauses, which are generated by refining bottom clauses, generated by examples, through theory revision techniques; and by adding the best clauses to compose a theory, in a greedy way, in order to maximize the likelihood of the examples [30].

Furthermore, [24] investigate the adoption of online learning algorithms to deal with relational data. The authors also extends ALEPH [52], which is a relational learning system, to deal with learning streams.

Finally, as explained before on Subsection 2.1.4, there are approaches that use graphical representation in order to make probabilistic inference. There are two of those which are more related to this work: MLN and RDN, since they use first-order logic to describe their knowledge. Furthermore, [28] proposes a gradient-based boosting approach in order to efficiently learn RDN models. Thus, the boosted RDN algorithm proposed in [28] was used in our experiments as a comparing method.

Chapter 3

Online Learning of ProPPR Models Based on Theory Revision

In this section, we present the Online Structure Learner by Revision (OSLR), the algorithm proposed in this work, 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.

First, we introduce the data structure used by our algorithm to revise the model, then we present the algorithm itself.

3.1 Finding Revision Points

Finding the correct places to change in a theory is quite a challenging and expensive task [20]. To smoothly do this task, we defined a tree structure to represent the model, aiming at efficiently traversing it to suggest possible modifications. A single tree is used to represent rules with the same 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.

Each node in the tree is associated either to a single literal in some rule or to a special node labelled as *false*. 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. Figure 3.1 illustrates an example of such a tree. The special node labelled as *false* is always a leaf, appearing as the child of an internal node n_i in the tree, and representing the failure of the literals that come after the literal associated with n_i in the rule, i.e. the failure of

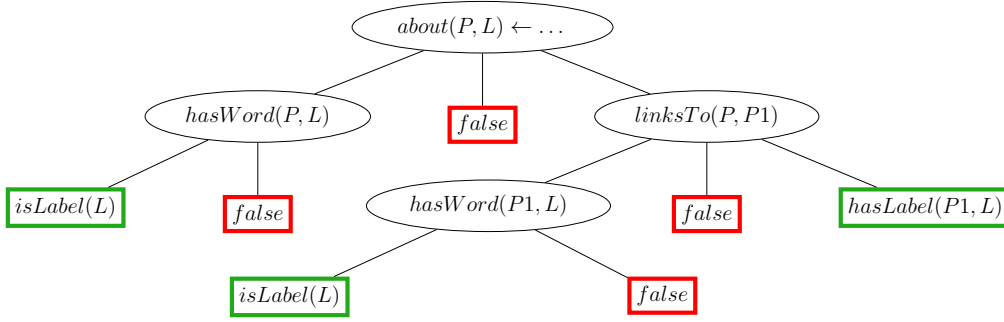


Figure 3.1: Tree Structure

all children of n_i . Each path in the tree, from the root to a leaf, that does not end in a *false* node represents a single rule in the theory. In this way, we do not consider rules that are a subset of another rule. As the leaves in the tree have a special role, representing either the end of the rule or a failure point, they are differentiated from the other nodes and graphically exhibited as squares.

Note that the features introduced in ProPPR language do not take part in the tree, as they act as latent variables to cope with the weights of the rules but they do not interfere with the logical part. Each leaf (square nodes) in this tree holds a set of examples. When a new example arrives, we pass it through the tree, to decide in which sets to put it. We start with the example 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 . If *none* of the children of v proves the example, it is placed on the *false* node, child of v . We repeat this process until the example reaches the leaves of the tree. In case an example is proved by more than one child node, it goes to all the nodes that prove them.

Since we use the ProPPR's example format and in this format several answers may be grouped into a single example, we split such answers into the ones that are covered by a rule and the ones that are not. Only the covered ones are passed to the children nodes, as the answers not covered eventually get stuck in a *false* leaf. A naive yet easy to understand algorithm to place the answers in the leaves of the tree can be seen in Algorithm 4.

The leaves of the tree point out the revision points of the model. In addition, the set of examples of each leaf represents the examples that should be taken into account when revising its correspondent revision point. The *false* leaves of the tree contain examples that are not covered by the correspondent part of the theory while the other squared leaves includes the examples covered by its respective rule. In this way, positive examples in the *false* leaves and negative examples in the other squared leaves are potential examples to suggest revisions to the theory. We use this information to choose which part of the theory should be revised.

Algorithm 4 Algorithm to Place the Example in the Tree Structure

Input:

Background Knowledge (BK);
A set of answers (A);
The tree structure ($tree$);

Output:

The tree structure ($tree$) with the answers in place;

```
function placeAnswers( $BK, A, tree$ )
  for each answer  $a \in A$  do
     $queue \leftarrow tree.getRoot()$ 
    while  $queue$  is not  $\emptyset$  do
       $node \leftarrow queue.pickFirstAndRemove()$ 
       $children \leftarrow passAnswerToChildren(BK, a, node)$ 
       $queue \leftarrow children \cup queue$ 
    end while
  end for
return  $tree$ 
end function
```

```
function passAnswerToChildren( $BK, a, node$ )
   $passedChildren \leftarrow \emptyset$ 
  if  $node.isLeaf()$  then
     $node.put(a)$ 
  else
    for each node  $n \in node.getChildren()$  do
      if  $proves(BK, node.getPartialRule(), a)$  then
         $passedChildren \leftarrow passedChildren \cup node$ 
      end if
    end for
    if  $passedChildren$  is  $\emptyset$  then
       $node.getFalseNode().put(a)$ 
    end if
  end if
return  $passedChildren$ 
end function
```

3.1.1 Revision Operators

Now that we have the tree structure indicating the revision points, and the set of examples that may be affected when modifying each point, we can define how such points are going to be changed, i.e., the revision operators. They are also supported by the tree and grouped into two types of revision operators: (1) addition of nodes; and (2) deletion of nodes.

For each revision point, we apply the possible operator and use the examples in the revision point in order to measure the improvement of the revised theory over

the current one. Following, we present the proposed revision operators.

Adding Nodes

A not-false leaf with negative examples indicates that the theory needs to be specialized, in an attempt to make the negative examples unprovable. On the other hand, a *false* leaf with positive examples indicates that the theory needs to be generalized.

Thus, in the first case, we add literals in a rule proving a negative example by adding a new path as a child of a not-false squared leaf (that will no longer be a leaf) in an attempt of making the negative examples unprovable due to these new literals, or at least reduce its proofs. In the second case, we would like to generalize the theory by adding a new rule. In this case we add a path as a new branch of the parent of a *false* leaf, creating a new path from this parent to a not-false leaf, therefore, creating a new rule.

To create such new paths, we rely on the concept of the bottom clause [26], to define the set of atoms that are candidates to become nodes in the tree. To generate the bottom clause, we transform the knowledge base in a graph GKB such that the nodes in this graph are the constants in the knowledge base (KB) and there is an edge between two constants if there is a fact in the KB with those two constants as its terms. Even in the presence of facts with arity greater than two, we still represent them in the graph, by creating the edges between all pairs of constants in the fact. For a fact with arity one, we include a loop edge, linking the node to itself.

Since we use the bottom clause in order to define the possible literals to be added into a rule, either to an existent rule or to start a new one, we dismiss the mode declarations, making the input configuration of the learning algorithm much simpler, not requiring a deep knowledge of the task at hand. This is appropriate to online environments, since new relations (predicates) may appear over time, and requiring a language bias specific for each new predicate would require constant human interaction with the system.

We define the bottom clause of depth 0 created from an example $p(c_1, \dots, c_n)$ as the set of edges connected to the nodes $c_1, \dots, c_n \in GKB$. A bottom clause of depth 1 is gathered from the set of edges from the bottom clause of depth 0 plus the edges connected to the nodes that are 1 edge away from the nodes c_1, \dots, c_n , i.e. the nodes that can be reached by going from a node $c_i \in \{c_1, \dots, c_n\}$ to another node passing through only 1 edge. For a bottom clause of depth n , we include the edges from nodes that are n edges away from the example's constants. This bottom clause is easily obtained by a breadth-first search in the graph of the knowledge base.

In order to create the bottom clause, we randomly pick a subset of a ProPPR example that is in the node indicating the revision and calculate the bottom clause of depth i for each one of the positive answers within these examples. Next, we

replace the constant terms by variables and combine their atoms into a single set. Both the depth of the bottom clause and the size of the subset of examples to be used are parameters to the algorithm.

Given the bottom clause, we have the candidate literals to generate the nodes to be included in the tree. There are two possible ways of including them in the tree:

Hill-climbing: includes a literal at a time, choosing the one that better improves the theory according to a scoring function. We keep adding predicates, in a greedy way, until one of the following stop criteria is met:

1. the score can no longer be improved, after a predefined number of tries; or
2. there are no more literals left to be added.

Since different literals from the bottom clause may generate rules that are logically equivalent, we remove the rules that are equivalent to previously generated ones.

Relational Path-finding: finds a path between the variables¹ of the example, within the subgraph of the KB represented by the bottom clause, following the classical relational path-finding algorithm [53].

We allow for both algorithms to be executed concurrently and keep the modification that better improves the theory.

Figure 3.2 shows an example of a subgraph of the knowledge base presented in Table 2.4 concerning the example *advisedby(person2, person1)*. The name of the constants were shortened for clarity. To create a new rule from scratch, the hill-climbing approach would consider any of the atoms connected either to *p1* or *p2* as a possibility and would create several rules, each of which with one of the possible atoms. On the other hand, the relational path-finding would look for paths connecting *p1* to *p2* and could propose, for instance, the following rule:

$$advisedby(X, Y) \leftarrow publication(Z, X) \wedge publication(Z, Y)$$

After substituting each distinct constant by a distinct variable.

As we perform the learning process over iterations, a rule previously learned by one method can be later improved by the other.

Deleting Nodes

The *deletion of nodes* operator removes nodes from the tree, and this change is reflected in the theory. This operator is applied on leaves that are not labelled as

¹The relational path-finding algorithm is only defined over binary target relations.

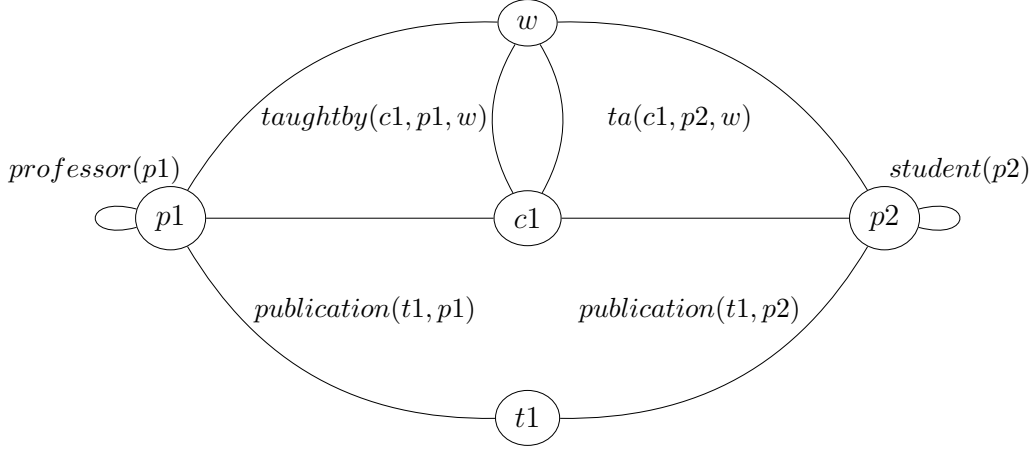


Figure 3.2: Bottom Clause Graph

false and may have two different behaviours, depending on the node on which it is applied. In case the leaf is the only not-*false* leaf child of its parent node, this deletion represents a **deletion of the antecedent** in the rule that the node represents. In this way, the parent of the node becomes a leaf and the rule is shortened. The other case happens when the leaf's parent has other not-*false* leaves as children, which incites the **deletion of a rule** in the theory, as the path ending in that leaf is erased.

The deletion of nodes in the tree represents either a deletion of a literal in a rule, or the deletion of an entire rule. In the first case, it provides the effect of generalizing the theory, while in the second case it specializes the theory. Here, we only allow the deletion of a single literal, to preserve as much as we can from the existing rules. In case there is a not-*false* leaf with negative examples, we delete the path from this leaf. To make sure that only a single rule is deleted at once, only the path leading to that leaf is deleted.

Deleting Nodes (Alternative Approaches)

Since the **deletion of the antecedent** case is a generalization of the theory, the most affected set of examples is the one in the *false* leaf child of the same parent of the deleted node. This set contains the examples that will be now proved, while the proof of the set of examples of the deleted node will remain unchanged.

In order to take this set of examples into account, we implemented an alternative version of the deleting operator that takes the examples of the *false* leaf (sister to the deleted node) as the set of example to use in the evaluation of the improvement of the revision; instead of the examples from the deleted node itself, in the case of a deletion of the antecedent. As such, this deleting of literal is now triggered by applying this operator on a *false* leaf whose parent node has a single not-*false* child. Differing from the *original* approach, that does not allow the application of

the deleting node operator on *false* leaves.

In case the antecedent is deleted, the examples proved by it, which are still proved by its parent node, are moved to its parent node; and its parent node now becomes a leaf. We will call this the *alternative 1* of the deletion operator, in addition to the *original*.

In addition to these two approaches, we have implemented another modification, on the top of the *alternative 1* approach, that allows the exclusion of a rule by deleting the leaf and its parent node, if any, until it reaches a bifurcation on the tree; which represents a node in common with another rule. We will call this implementation as the *alternative 2*.

In this case, suppose the node $isLabel(L)$ from Figure 3.1, which represents the last literal from the rule (1) from Table 2.2. There will be different behaviours, depending on the implementation used:

1. In the *original* approach, we would delete the literal, using the examples from this leaf to evaluate the proposed revision;
2. In the *alternative 1* approach, we would delete the literal, when applying the deleting node operator on the *false* leaf, child of its parent; using the examples from the *false* leaf, to evaluate the proposed revision;
3. In the *alternative 2* approach, it could either delete the literal following the *alternative 1* approach, when applied to the *false* leaf; or delete the whole Rule (1), from Table 2.2, when applied on the not-*false* leaf, evaluating the revision on the examples from the not-*false* leaf. Since these operations occur on different leaves, the one to be applied will depend on the potential of the leaf and if the improvement is significant, based on the Hoeffding's bound, as will be explained later in this section.

In this alternative, we allow for deeper modification in the theory, which could make the model more susceptible to overfit, especially in online environment where some new examples could make a big modification and lose the representation of the older examples.

In fact, as we will show in Chapter 4, these alternatives make little difference on the performance of the learned model.

3.2 Learning Algorithm

The top-level procedure designed in this work to learn ProPPR models in an online fashion is described in Algorithm 5. The input is background knowledge and a

possibly infinite stream of examples. For simplicity we assume that this stream contains examples of a single predicate.

Algorithm 5 OSLR: The learning algorithm proposed in this work to learn from a stream of relational examples

Input:

Background Knowledge (BK);
 An initial theory ($theory$), possibly empty;
 Stream of Examples (E), possibly infinite;
 Maximum number of revisions (N);

Output:

A ProPPR model

```

tree ← initialTree(E, theory)
for each  $e \in E$  do
  leaves ← putExamplesIntoLeaves(BK, tree,  $e$ )
  theory, tree ← revise(BK, theory, leaves,  $N$ )
end for

```

First, we start the tree to handle the predicate of the stream of examples. Then, for each set of examples received in an instant of time, from the stream, we perform two steps: (1) we place the examples in the leaves of the tree structure, as already explained in the previous section; and (2) we call the revision on the leaves that may have absorbed new examples.

It is important to point out that the *revise* function does not necessarily change the current model, but, instead, proposes modifications to it and evaluates the benefits of really changing it. This function is detailed in Algorithm 6, which is next described in the following subsections.

3.2.1 Leaf Potential Heuristic

In line 8 of the *revise* function we sort the leaves based on their potential to improve the quality of the theory. Since we already know if a leaf proves or not the examples in it, and the number of positive and negative examples of the leaf, we can easily calculate the number of misclassified examples of the leaf, and use it as a heuristic to estimate the potential of the leaf to improve the theory.

The heuristic is that the leaves with the largest number of incorrectly classified examples are the ones that have more potential to improve the theory if revised [17]. If it is a *false* leaf, we pick the number of positive examples on it; when it is not a *false* leaf, we pick the number of negative ones. Thus, we sort the leaves by these numbers, from the highest to the lowest.

Algorithm 6 Revise Leaves

Input:

- 1: Background Knowledge (BK);
- 2: An initial theory ($theory$), possibly empty;
- 3: Set of leaves ($leaves$);
- 4: Maximum number of revisions (N);

Output:

- 5: A ProPPR model ($theory$)
- 6: The tree structure ($tree$)

```
7: function revise( $BK, theory, leaves, N$ )
8:    $leaves \leftarrow sortLeavesByHeuristic(leaves)$ 
9:   for  $i \leftarrow 1; i \leq minimum(size(leaves), N)$  do
10:     $tree \leftarrow leaves[i].getTree()$ 
11:     $examples \leftarrow getExamples(leaves[i])$ 
12:     $operator \leftarrow pickBestOperator(BK, theory, leaves[i], examples)$ 
13:     $theory, tree \leftarrow apply(BK, theory, tree, examples, operator)$ 
14:   end for
15: return  $theory, tree$ 
16: end function
```

3.2.2 Selecting Operators

Given each selected leaf, we now apply both the operators, when possible, as described in line 12 of the algorithm; each operator proposes a new theory by changing the leaf. We evaluate these new theories and pick the one that better improves the pre-defined score function computed on the correspondent set of examples, i.e. the set of examples of the leaf where the operator was applied.

Algorithm 7 depicts line 12 of Algorithm 6. If a revision operator cannot be applied to a specific leaf (for instance, a deletion of a *false* leaf), it returns an empty theory. An empty theory has no score, as such, it would not be considered as the best possible operator to be applied, since there is always an applicable operator for each leaf.

It is worth noticing that the operators act on the logical part of the model. However, as we have explained in the ProPPR's inference method (2.2.2), the logical part influences 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. As we use a probabilistic evaluation function to compute the quality of the model, these probabilities are taken into account to decide whether a model is better than the other.

Algorithm 7 Select the Best Operator for the Leaf

Input:

- 1: Background Knowledge (BK);
- 2: A theory ($theory$);
- 3: A leaf ($leaf$);
- 4: A set of examples ($examples$);

Output:

- 5: A revision operator ($bestOperator$)

 - 6: **function** pickBestOperator($BK, theory, leaf, examples$)
 - 7: $bestScore \leftarrow 0$
 - 8: $currentTheory \leftarrow theory$
 - 9: $bestOperator \leftarrow null$
 - 10: **for** each $operator \in operators$ **do**
 - 11: $currentTheory \leftarrow applyOperator(BK, currentTheory, operator, leaf)$
 - 12: $currentScore \leftarrow evaluate(BK, currentTheory, examples)$
 - 13: **if** $currentScore > bestScore$ **then**
 - 14: $bestScore \leftarrow currentScore$
 - 15: $bestOperator \leftarrow operator$
 - 16: **end if**
 - 17: **end for**
 - 18: **return** $bestOperator$
 - 19: **end function**
-

3.2.3 Hoeffding's Bound

After choosing the operator that most benefits the model, we have to decide if it is going to be implemented and then replace the current model, or if it is better to keep the previous model.

We use the Hoeffding's bound [21] 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 observations of this random variable, with empirical mean \bar{r} , $\Pr(\mu_r \geq \bar{r} - \epsilon) = 1 - \delta$, where μ_r is the true mean of the random variable r , δ is a chosen parameter and ϵ is given by Equation 3.1:

$$\epsilon = \sqrt{\frac{R^2 \ln(1/\delta)}{2n}} \quad (3.1)$$

We use the difference between the evaluation of the two theories, in the examples of the leaf, as the random variable. Consider a bounded metric M and a set of examples E containing n examples. Let C be the current theory and N be the new theory proposed by revising C based on E . Let $\Delta M = M(N, E) - M(C, E)$ 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 Equation 3.1. 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 new theory and the current one exceeds the threshold ϵ established by the Hoeffding’s bound, given the number of examples, we say that it is better to make the revised theory as the current one, otherwise, we discard the revise theory and the algorithm continues. If the revision is accepted, the examples from the used leaf are discarded.

It is worthy to mention that the Hoeffding’s bound assumes that the examples are independent given the class, which is usually not true in the relational case, since there may be relations between different examples [25]. To overcome this problem, we consider that two ProPPR examples are dependent on each other if they have a literal in common on their respective bottom clauses, given a defined depth. Then, we use as the number of examples, in the Hoeffding’s bound equation, only the number of independent examples. By definition, all the answers of an example are related, since all of them have, at least, a node in common, making all the edges of this node to appear in the bottom clause of all the answers. Thus, it counts as only one independent example, even if such an example has a number of answers.

This method may not guarantee the full independence of the examples, however, it is a good estimator of the *effective* sample size. It is shown in [25] that the *effective* size of a sample can be drastically reduced by the relation among dependent examples, as such, increasing the variance of estimators based on such data. Since the Hoeffding’s bound assumes that the sample is composed by independent examples, using the number of examples on the sample would overestimate its size. In order to overcome this issue, we use the number of independent examples to better estimative the *effective* size of the sample.

3.2.4 Feature Generation

Whenever we propose a revision on a rule, we use a heuristic to define the rule’s feature, in the ProPPR format. Then, if the revision is implemented, we call the ProPPR parameter learning routine to learn the weights given the examples used on the revision, improving the quality of the model.

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 make 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 of the rule that better guide the PageRank-Nibble in the graph. In

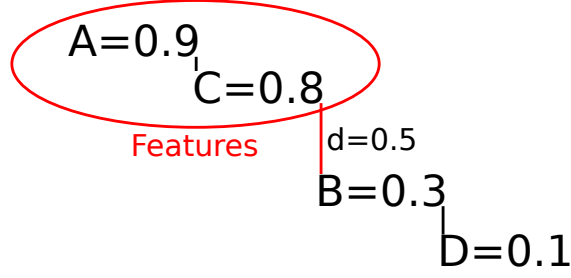


Figure 3.3: Feature Selection Example

this way, we propose a heuristic to decide which variables are better to be in this subset. To do that, first we get all possible θ substitutions that prove the examples used to propose the revision. Then, for each variable, in the clause used to prove the examples, we create two sets: the set of constants that substitute this variable and appear in the positive examples and the ones that appear in the negative examples. Given these two sets, we calculate the value of the heuristic H , for each variable, according to Equation 3.2:

$$H = \frac{|Intersection|}{|Union|} \quad (3.2)$$

where *Union* is the union of constants appearing in the positive and negative sets and *Intersection* is the intersection between them. It is also known as Jaccard's index.

Intuitively, 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. In this way, ProPPR is able to reduce the weights of such rules, in order to avoid paths leading to both positive and negative answers, as has been observed in our experiments.

We calculate this heuristic for each variable in the rule, then we sort the variable by its value, from 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.

For instance, consider an example where we have four variables A, B, C, D , and the heuristic values for each one of them are 0.9, 0.3, 0.8, 0.1, respectively. Thus, we have that the order of the variable, from highest to lowest is A, C, B, D , as illustrated in Figure 3.3. Then, we calculate the biggest difference d , which lies in between variables C and B and is of 0.5. In this case, we choose variables A and C to appear in the feature of the rule.

The ProPPR's 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.

3.2.5 Applying the Revision

We have explained the overall of the learning algorithm in the subsections above. In this subsection, we will describe in more details the final part of the revision iteration, which is to decide if the revision, of a *single leaf*, will be applied or if it will be discarded. Finally, we summarize the revision iteration of the learning system.

Algorithm 8 shows the *apply* function called in line 13 of Algorithm 6. The algorithm starts by comparing the current theory and the theory proposed by the revision operator in the examples of the leaf revised at the moment (lines 9-10). Then, it calculates the Hoeffding's bound threshold value ϵ , based on the number of independent examples (line 11). Finally, it compares the improvement of the score from the revised theory, over the score of the current theory, against ϵ (line 12). If the improvement is larger than ϵ , the revision is applied; if not, the revision is discarded and the algorithm proceeds to revise the next leaf.

Algorithm 8 Apply Revision

Input:

- 1: Background Knowledge (*BK*);
- 2: A theory (*theory*);
- 3: The current tree (*tree*);
- 4: A set of examples (*examples*);
- 5: A revision operator (*operator*)

Output:

- 6: A revised theory (*theory*);
- 7: A revised tree (*tree*);

```
8: function apply(BK, theory, leaf, examples, operator)
9:   currentScore  $\leftarrow$  evaluate(BK, theory, examples)
10:  revisedScore  $\leftarrow$  evaluate(BK, operator.getTheory(), examples)
11:   $\epsilon$   $\leftarrow$  calculateHoeffdingBound(getIndependent(examples))
12:  if revisedScore - currentScore >  $\epsilon$  then
13:    theory  $\leftarrow$  operator.getTheory()
14:    tree  $\leftarrow$  operator.getTree()
15:    learnProPPRParameters(examples)
16:  end if
17: return theory, tree
18: end function
```

In the case of the improvement of the revision on the leaf do exceeds the threshold imposed by the Hoeffding's bound, the revised theory becomes the current theory and the tree is updated to reflect the changes on the theory. Then, we call the parameter learning function of the ProPPR system, on the examples from the revised leaf, in order to learn the weight of the features that might have changed on the

revision process. Finally, the examples from the revised leaf are discarded, which is important for systems that learn from stream, since the stream may be very large or even infinite.

We can summarize an iteration of the learning system in the following steps:

1. New examples arrive and are placed in the tree structure;
2. The revision is called for each leaf that might have absorbed new examples, sorted by the leaf's potential;
3. The operator that leads to the best revised theory for the current leaf is chosen to be applied;
4. If the improvement of the revised theory exceeds the Hoeffding's bound:
 - (a) The revision is implemented;
 - (b) The ProPPR's parameters are retrained;
5. The system proceeds to the next leaf.

The system processes each iteration at a time, always having a model ready to make predictions. In the next chapter we present the experiment we have performed along with their results.

Chapter 4

Experiments & Results

In this section we present the experiments that we have performed to evaluate the method proposed in this work and the results obtained from them. 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.

4.1 Simulating an Online Environment

We proposed a simulated online environment in order to evaluate the application if theory revision techniques online would be beneficial in order to learn a relational model in terms of: quality of the learned model, the research question **Q1**; and runtime of the learning algorithm, the research question **Q2**.

In these experiments, we have compared OSLR against: (1) RDN-Boost, which is an approach that learns Relational Dependency Networks proposed in [28]; (2) ProbFOIL, which is an extension of FOIL [51] to run over ProbLog [29]; and (3) Slipcover, which implements an improved search strategy to learn first-order theories [30]. We considered two domains that were made available by [28], namely: the Cora [34] domains, composed of four target relations (datasets); and the UWCSE [44] domains, with a target relation. In the following, we describe the experimental methodology, the datasets and the obtained results.

4.1.1 Datasets

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

Cora is a dataset for citation matching [34]. This dataset is composed of six background binary relations, represented by the following predicates: author, has-

Table 4.1: Size of the Datasets

Relation	# Positives	# Negatives	# ProPPR Format
Same Author	488	66	88
Same Bib	30,971	21,952	1,295
Same Title	661	714	209
Same Venue	2,887	4,976	403
Advised By	113	16,601	91

WordAuthor, hasWordTitle, hasWordVenue, title and venue; having a total of 6,540 facts. 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.

UWCSE dataset describes thirteen relations about the professors, students and courses from the University of Washington, having 12,615 facts [10]. The goal of this dataset is to predict the advisedBy relations between students and professors, given the other relations.

Before we can split the dataset into iterations, we have to convert the examples to the ProPPR’s format, as described in 2.2.3. We do this by creating a query $p(a, X)$ for each predicate p and constant a that appears in the first term of each example, then we group all the original examples that have the same first term as the answers for their respective queries. Table 4.1 shows the number of examples for each relation, the first four are from the Cora domain; and the last one is from the UWCSE. The last column shows the number of examples in the ProPPR’s format. It is worthy remembering that in the ProPPR’s format, several original examples are grouped into the same example.

Since the number of negative examples in the UWCSE is much bigger than the positive ones, we downsample the set of negative examples to be twice as much as the number of positives ones, as it is done in [28].

Finally, we split the relations into iterations. For the Cora dataset, 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 2.20GHz with 16GB 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 averages and standard deviations of those runs; at each time we shuffle the examples and for the UWCSE dataset we resample the negative examples. Slipcover did not successfully finish 5 of the 30 runs for the UWCSE dataset, so their results

is the average over the 25 successfully finished runs.

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. In OSLR we use the following parameters by default (unless explicitly specified otherwise):

- We pass all the examples of the iteration at once to the algorithm 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, in ProPPR’s format, from the set of examples;
- As evaluation score, we optimize the area under the Precision-Recall curve;
- The Hoeffding’s bound δ parameter is 10^{-3} , with a decay of 0.5 every time a revision is implemented, we use a decay to allow us to start with a higher δ , allowing the algorithm to learn quicker in the initial of the prose, being able to answer queries as soon as possible; and then, reduce the δ as revisions are implemented, avoiding overfitting; and
- We use the depth of 0 to define whether two examples are dependent on each other, so that the algorithm could not consider almost all of the examples as dependent from each other.

The RDN was run with the parameters made available by its authors along with the datasets. The ProbFOIL algorithm was run with its default parameters, as it is done in [29]. The Slipcover algorithm has a lot of parameters to tune, we also uses its default parameters on our experiments.

It is worthy pointing out that our goal is not to tune the parameters of each system in order to get the best possible model for each one of them, but to evaluate the systems on different datasets with minimal tuning.

4.1.2 Methodology

We simulate the online incoming of examples by splitting the examples in the dataset into iterations and by passing one 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 [54]: we start from an empty model and test it on the first iteration of examples, then we pass this iteration to train the algorithm, it (possibly) updates its model and then the model is tested on the

next iteration. In this way, every example from E will be used to evaluate the model before the model is able to train on it. We repeat this process until no more iterations are available. We assume that the KB is fixed during the iterations.

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

4.1.3 Results

We now present the results for experiments simulating an online environment separately for each dataset; then we will present the runtime of experiments; and we close this subsection with the results of the alternative operators that we presented in 3.1.1.

Cora Dataset

Figure 4.1 shows the evaluation of the area under the Precision-Recall curve for each one of the Cora’s relation 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 in the charts, OSLR outperforms the RDN and Slipcover on all the Cora’s relation 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.

We can also see that Slipcover does not performs very well, being the worst method on all target relations, except by the Same Venue relation, where it outperforms the RDN approach.

The values are considered statistically relevant, by a two-tailed paired t-test with $p < 0.05$, for all pair of methods for each relation, excepted by the final performance for Same Author relation between our approach and the RDN approach.

The AUC results are summarized in Table 4.2 while the Final results are on Table 4.3. ProbFOIL does not work well on the Cora dataset, as it returns the most general clause for all target relations.

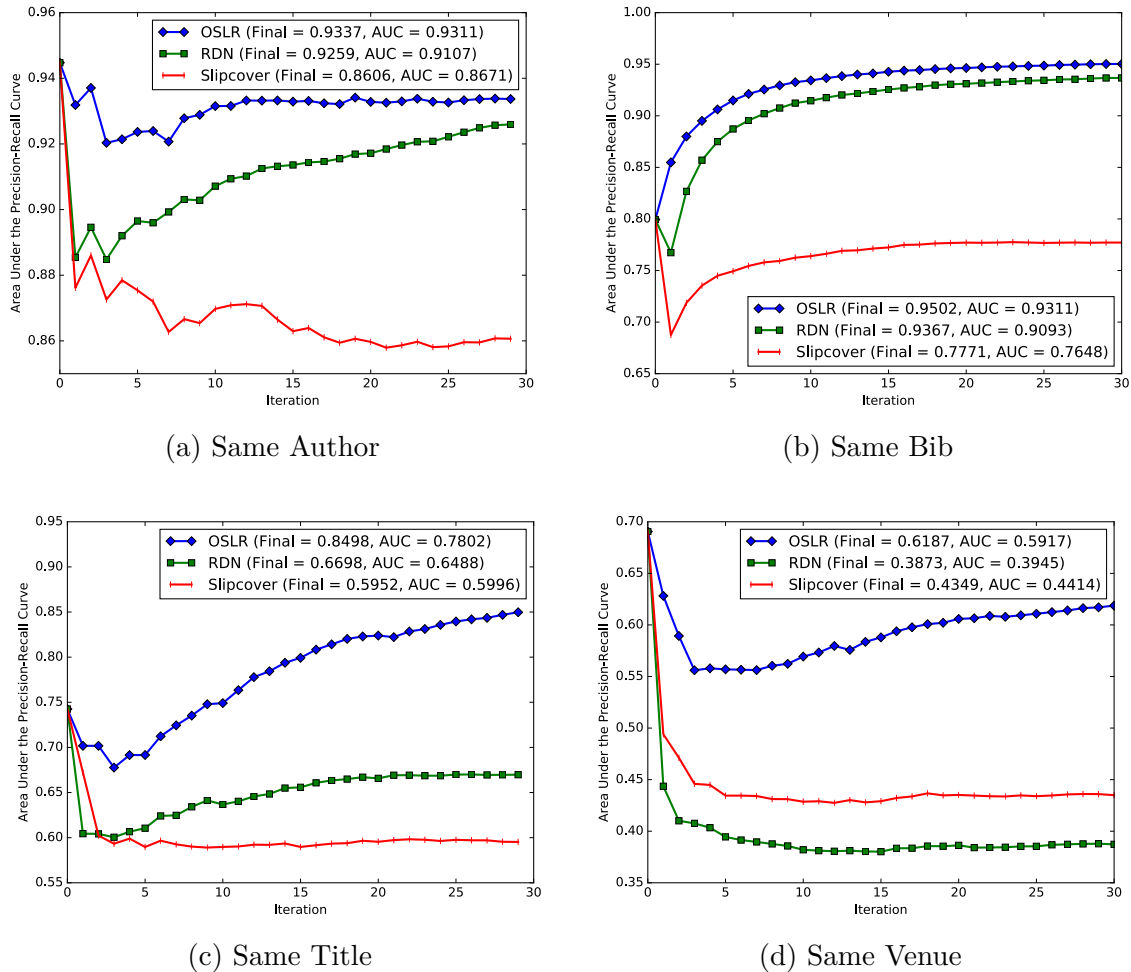


Figure 4.1: The Evaluation of the Cora Dataset Over the Iterations

Table 4.2: Area Under the Iteration Curve of the Methods on Cora Datasets

Relation	OSLR	RDN	Slipcover
Same Author	0.9311 ± 0.0319	0.9107 ± 0.0211	0.8671 ± 0.0244
Same Bib	0.9311 ± 0.0150	0.9093 ± 0.0073	0.7648 ± 0.0103
Same Title	0.7802 ± 0.0689	0.6488 ± 0.0316	0.5996 ± 0.0363
Same Venue	0.5917 ± 0.0801	0.3945 ± 0.0293	0.4414 ± 0.0232

UWCSE Dataset

Figure 4.2 shows the results obtained from the UWCSE dataset. In the upper Figure 4.2a we can see that the RDN approach is slightly better than OSLR and both ProbFOIL and Slipcover methods achieve poor performance on this dataset. In the lower Figure 4.2b we present the methods learning from an initial theory. Instead of starting from an empty hypothesis, we started from this initial theory in the Background Knowledge¹.

¹This 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, by getting only the clauses with the target atom in their heads and removing the other atoms from their heads, except

Table 4.3: Final Performance of the Area Under the Precision-Recall Curve of the Methods on Cora Datasets

Relation	OSLR	RDN	Slipcover
Same Author	0.9337 ± 0.0399	0.9259 ± 0.0145	0.8606 ± 0.0140
Same Bib	0.9502 ± 0.0097	0.9367 ± 0.0028	0.7771 ± 0.0083
Same Title	0.8498 ± 0.0654	0.6698 ± 0.0165	0.5952 ± 0.0158
Same Venue	0.6187 ± 0.0854	0.3873 ± 0.0154	0.4349 ± 0.0100

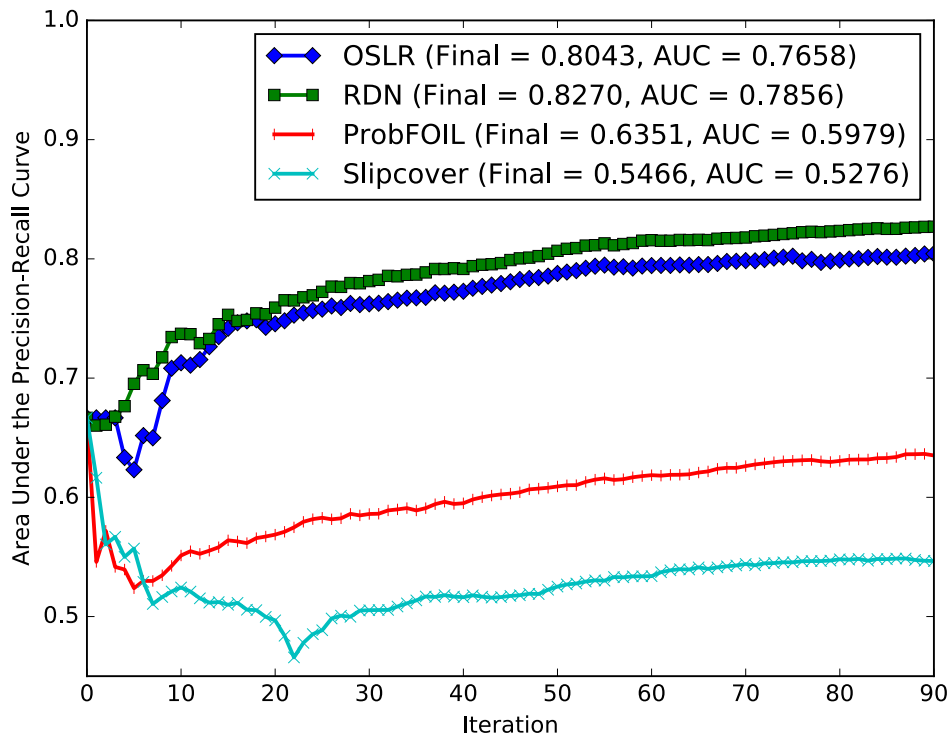
Since the RDN is the only approach that have its performance decreased by the presence of an initial theory, we also include in this chart a line for the RDN without initial theory (the RDN line) and we call RDN+I the line for the RDN with the initial theory. Theory line is the performance of the initial theory itself (according to ProPPR language and inferred with our system) without any revision, and may be used as a baseline.

As we can see, when starting from an initial model, and allowing the revision system to modify it, we obtained slightly better results than the RDN with 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 the performance of the other approaches increase. This might be caused by errors in the initial theory; since the RDN is not able to modify it, and it learns an expressive model, the errors are going to be taken to the final model. On the other hand, the other approaches were unable to learn complex models, and thus, an initial (even partially correct) theory might bring a gain in their performances. However, 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.

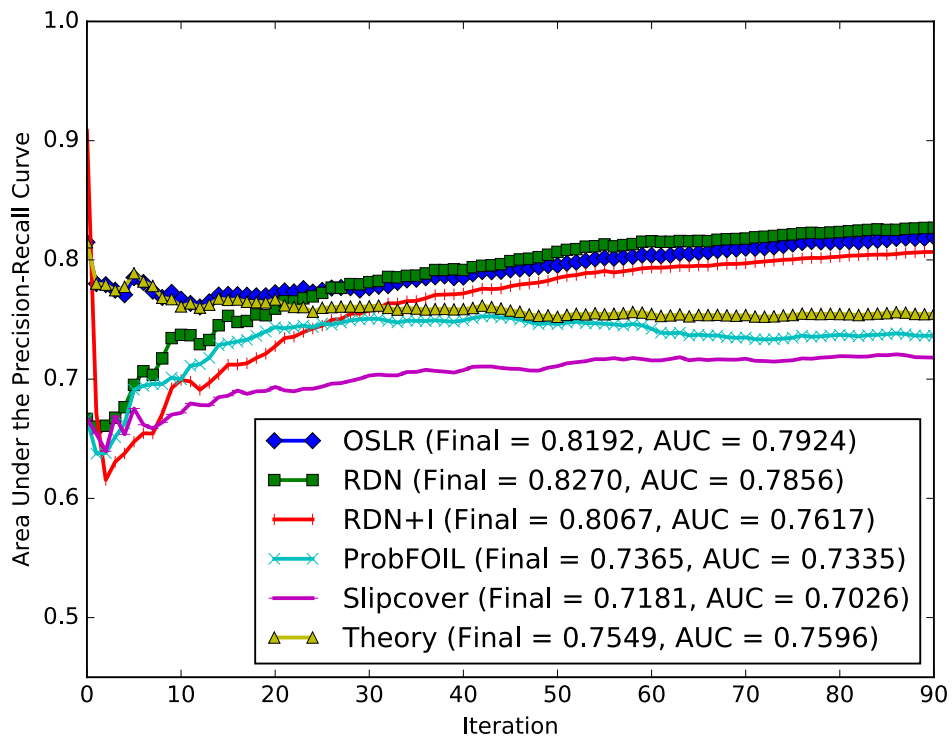
Another interesting phenomena that happened in the experiments is that even the initial theory improving the performance of both ProbFOIL and Slipcover approaches, their performance remain lower than the initial theory itself, inferred by the ProPPR system. In fact, the ProPPR’s authors have already notice that logic theories inferred by ProPPR get good results, even without parameter training, using the uniform weights, given its bias to assign higher probability values to shorter answers [13]; this might be the case of better results of the original theory found here.

It is worthy pointing out that there is no statistical significance, based on a two-tailed paired t-test with $p < 0.05$, in any of the comparisons between our approach (with and without initial theory) and the RDN without initial theory in the UWCSE dataset. Table 4.4 summarizes the AUC results for the UWCSE dataset, while Table 4.5 summarizes the Final results; the last row (Init), on both tables, represents the

by the target, since they allow for a clause to have multiples atoms on its head. We also remove rules whose set of atoms in its body are a superset of the atoms on the body of a more generic rule



(a) Advised By



(b) Advised By with Initial Theory

Figure 4.2: The Evaluation of the UWCSE Dataset Over the Iterations

Table 4.4: Area Under the Iteration Curve of the Methods on UWCSE Dataset

Experiment	OSLR	RDN	ProbFOIL	Slipcover
Advised By	0.7658 ± 0.0510	0.7856 ± 0.0509	0.5979 ± 0.0423	0.5276 ± 0.1153
Advised By - Init	0.7924 ± 0.0526	0.7617 ± 0.0523	0.7335 ± 0.0288	0.7026 ± 0.0492

Table 4.5: Final Performance of the Area Under the Precision-Recall Curve of the Methods on UWCSE Dataset

Experiment	OSLR	RDN	ProbFOIL	Slipcover
Advised By	0.8043 ± 0.0506	0.8270 ± 0.0363	0.6351 ± 0.0412	0.5466 ± 0.1019
Advised By - Init	0.8192 ± 0.0463	0.8067 ± 0.0380	0.7365 ± 0.0186	0.7181 ± 0.0410

approaches with the initial theory.

Runtime Analysis

Table 4.6 shows the average runtime of each system on each relation. The columns show the average runtime of all the experiments, i.e. the N task per relation, where N is the number of iterations of the relation.

We can see that our system takes, in the worst case, approximately 2 times longer than the RDN would take to learn a task, as it is in the Same Venue relation, but it is faster than the RDN in all other datasets. 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.

Despite the low running time of the Slipcover system on every dataset, it was only able to learn a very simple theory, with only a single rule with a single literal on the body, for most of the runs; which reflects its poor performance on the experiments.

It is also interesting to notice that our approach was able to gain from an initial theory both in the quality of the learned model and in the decreasing of the learning time.

Table 4.6: Average Runtime for Each Learned Relation

Relation	OSLR	RDN	ProbFOIL	Slipcover
Same Author	0min19s:483	6min17s:979	-	0min3s:400
Same Bib	29min0s:606	1h18min1s:383	-	0min5s:167
Same Title	3min9s:413	10min31s:380	-	0min19s:867
Same Venue	1h24min0s:26	46min11s:78	-	0min21s:700
Advised By	9min0s:972	23min41s:939	39min20s:602	1min8s:680
Advised By Init	5min59s:588	38min11s:854	5h51min25s:372	1min8s:400

Alternative Operators Results

As we have explained before, we have implemented two alternative versions of the deleting node operator of our approaches. Tables 4.7 and 4.8 show the performance of the alternative approaches for the UWCSE dataset, on the AUC and Final, respectively; for both learning from scratch and from an initial theory. The best result from each column is underlined. As can be seen, there are minimal changes on the performance of the methods, compared with each other.

Just to remember, *Alternative 1* approach is the one that considers the examples in the *false* leaf of the same parent of the node that will be deleted, in a literal deletion operation, as the examples to evaluate the revised theory. While the *Alternative 2* is the one that, in addition to *Alternative 1*, allows the exclusion of any rule represented by the not-false leaf it is applied to, possibly removing several nodes from the tree structure. See 3.1.1 for further details on the alternative approaches.

Table 4.7: AUC of the Alternative Methods on UWCSE Dataset

Methods	Advised By	Advised By - Init
Original	0.7658 ± 0.0510	0.7924 ± 0.0526
Alternative 1	<u>0.7675 ± 0.0523</u>	<u>0.7925 ± 0.0524</u>
Alternative 2	0.7614 ± 0.0522	0.7925 ± 0.0527

Table 4.8: Final Performance of the Alternative Methods on UWCSE Dataset

Methods	Advised By	Advised By - Init
Original	0.8043 ± 0.0506	0.8192 ± 0.0463
Alternative 1	<u>0.8058 ± 0.0503</u>	<u>0.8194 ± 0.0462</u>
Alternative 2	0.8008 ± 0.0542	0.8191 ± 0.0464

The results of Cora dataset can be seen on Table 4.9, for the AUC performance; and on Table 4.10 for the Final performance. Again we underline the best result from each columns.

Table 4.9: AUC of the Alternative Methods on Cora Dataset

Methods	Same Author	Same Bib	Same Title	Same Venue
Original	0.9311 ± 0.0319	<u>0.9311 ± 0.0150</u>	0.7802 ± 0.0689	0.5917 ± 0.0801
Alternative 1	<u>0.9378 ± 0.0318</u>	0.9118 ± 0.0438	<u>0.7960 ± 0.0814</u>	<u>0.5951 ± 0.0849</u>
Alternative 2	0.9307 ± 0.0281	0.8979 ± 0.0502	0.7671 ± 0.0618	0.5834 ± 0.0908

As can be observed from the results of the alternative approaches, in most of the cases, *Alternative 1* approach improves the performance of the learned model. Thus, making us conclude that this set of examples is more informative to be taken into account in this type of revision. On the other hand, *Alternative 2* decreases the performance of the learned model on most of the cases. The behaviour of this approach

Table 4.10: Final Performance of the Alternative Methods on Cora Dataset

Methods	Same Author	Same Bib	Same Title	Same Venue
Original	0.9337 ± 0.0399	0.9502 ± 0.0097	0.8498 ± 0.0654	0.6187 ± 0.0854
Alternative 1	0.9428 ± 0.0371	0.9329 ± 0.0420	0.8493 ± 0.0921	0.6155 ± 0.0894
Alternative 2	0.9351 ± 0.0306	0.9262 ± 0.0490	0.8311 ± 0.0722	0.6013 ± 0.0989

probably allows deeper modification in the theory, which may cause an overfit to the current examples, which is not desired, specially in an online environment.

4.2 Batch Environment

In this experiment we compare OSLR in a batch environment against the HTilde system [23, 32].

We use a 5x2 fold cross-validation, on another relation extracted from Cora dataset², the samePaper relation. Each of these five folds has a knowledge base of approximately 231,500 facts and about 838,500 examples, equally distributed in the two folds. For each of the five folds we perform two trainings, using a fold at a time as training set and the other as test, having a total of 10 trainings per method. OSLR process 10 examples at a time until it finishes the fold. Since the RDN was the best algorithm to compare with, we considered only it to this dataset.

This base is highly unbalanced, with the negative classes representing about 96% of the examples. We did not down sample the negative class here, because we would like to reproduce the results previously published at [32].

Table 4.11 summarizes the results of this experiment. These results are the average over the 10 runs of the 5x2 fold cross-validation, and has statistical significance of both OSLR and RDN methods against HTilde on a two-tailed paired t-test with $p < 0.05$.

Table 4.11: 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

As can be seen, the HTilde outperforms (with statistical significance) both OSLR and the RDN approach. However, HTilde needs a large number of examples, while OSLR seems to benefit from small amount of examples, as it obtains better results on early iterations, where only few examples were used to train the model. We

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

were not able to use HTilde on the UWCSE dataset, for instance, due to the small number of positive examples.

Another point that is worthy mentioning is that HTilde proposes logic theories larger than OSLR, often with more than 10 rules, reaching 21 rules in one of the folds, while OSLR's theories are usually composed by a few short rules, hence it is easier to understand their meaning.

Since we had to run the HTilde on a virtual machine, due to compatibility issues, we will not report the learning time for this experiment.

Chapter 5

Conclusions

In this work, we proposed a novel method to online learn the structure written in ProPPR language, relying on theory revision techniques, called Online Structure Learner by Revision (OSLR). In addition to the structure learning algorithm, we also proposed a heuristic to create the ProPPR’s features.

Online learning is a field of study that presents many challenges, especially in the area of learning first-order logic models, where this works lies in. This is because we have to learn the structure of the first-order models, this is, the logic theory and this is a challenging task for mainly two reasons: it has a very large hypotheses space; and, in order to do it online, the ideal would be to consider the structure of previously learned model.

To mitigate the problem of the large hypotheses space, we first proposed a tree structure to efficiently identify the revision points of the logic theory, limiting the number of possible revisions to the leaves of this tree structure. In addition, we have used the concept of the bottom clause [26] to restrict the search space of possible literals to be added to a clause.

In order to cope the challenge of considering the previously learned structure we have used techniques from theory revision from examples. Theory revision assumes that might exist a (not necessarily correct) theory that can be used as starting point to describe the training examples. In this way, theory revision benefits from this initial theory, making adjustments to it as needed in order to better explain the examples. In this work, we use as initial theory the theory learned in the past, and uses theory revision in order to keep this theory up to date with the arrival of new examples.

Revising the theory for each new misclassified example would be a very costly task and could easily lead to an overfitted model. To overcome these two issues, we used the Hoeffding’s bound statistical theory [21] to decide whether a revision should be implemented or not. Since the Hoeffding’s bound assumes a set of independent examples, which is usually not the case for a relational environment, we proposed

a mechanism based on the bottom clause to decide if two examples are dependent given a specified depth (or strictness) and we consider, for the Hoeffding’s bound, only the number of independent examples.

The application of theory revision techniques online brings two important benefits to the learning process: (1) the method 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; as was shown beneficial, on our experiments.

Our experiments showed that OSLR performs well on a simulated online environment, even on the beginning of the learning process, where only few examples are used to train the model, as have already been observed by [17, 19, 20] that theory revision works well with few examples. In our simulated online environment, we outperformed a state-of-the-art RDN-Boost approach [27, 28] on the Cora dataset [34], achieving a larger area under the Precision-Recall curve on all the target relations; and the system was comparable to the RDN-Boost on the UWCSE [10], slightly worst without initial theory and slightly better with an initial theory, for the same metric.

In addition, OSLR outperforms ProbFOIL [29] and Slipcover [30] which are two learning systems for learning probabilistic first-order theories from examples.

However, it has not performed so well in a batch environment, passing a set of examples at a time, since it is not designed to act in this kind of circumstance.

5.1 Future Works

We believe that this work opens several possibilities of future research, as such, we will point out some further directions in this section.

Arguably a natural extension of this work would be to add new logic inference mechanisms such as ProbLog [55] or Markov Logic Networks [44] in order to allow more expressive languages to be used, since they can handle negation. The implementation of mechanisms that allows more expressive languages would bring the need to evolve the structure learning system to allow it to take advantage of this new language expressiveness. An example would be to deal with non-monotonic knowledge by including, for instance, negation-as-failure and also to allow the system to learn structures with those negations.

Another direction, instead of making the system more expressive, would be to make it smarter by implementing ways of finding the best parameters of the learning system for a given task, in an approach similar to Auto Machine Learning [56]. Finding a way of adjusting these parameters as needed, or even learning to automatically adjust them, would be very beneficial for an online learning system [57].

Considering further experiments, it would be advantageous to better investigate, for instance, the impact of considering the dependent examples in the Hoeffding’s bound against considering all examples as independent, or to strict even more the dependency criteria by increasing the depth of the dependence bottom clause. Another interesting aspect to investigate would be the performance of OSLR on other datasets, specially in the presence of expert-made initial theories.

Furthermore, it would be interesting to investigate the performance of OSLR in the presence of *Concept Drift* [58]. If it performs well or not and which measures could be taken in order to make it better suited to such a task. Investigating whether a better choice of parameters, for instance a smaller δ value for the Hoeffding’s bound, would be sufficient; or it would be necessary to implement active mechanisms to detect the drifts in the data streams.

Nevertheless, in this work we demonstrated a way of applying theory revision in an online environment. The performed experiments revealed that revising previously learned model online is beneficial for both the quality of the learned model and the runtime of the learning algorithm; archiving better results in the Cora domains and competitive results in the UWCSE domains. Additionally, the experiments have shown that the use of a human made initial theory, despite being only partially correct, has provided useful information for the theory revision system, allowing it to learn better models. Finally, we believe that theory revision is well suited to be applied to online learning tasks.

Bibliography

- [1] MITCHELL, T. M. *Machine Learning*. McGraw Hill series in computer science. 1 ed. New York, NY, USA, McGraw-Hill, Inc., 1997.
- [2] DOMINGOS, P., HULTEN, G. “Mining High-speed Data Streams”. In: *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '00, pp. 71–80, Boston, Massachusetts, USA, August 2000. ACM.
- [3] BIFET, A., HOLMES, G., KIRKBY, R., et al. “MOA: Massive Online Analysis”, *Machine Learning*, v. 11, n. May, pp. 1601–1604, 2010.
- [4] DRIES, A., DE RAEDT, L. “Towards Clausal Discovery for Stream Mining”. In: De Raedt, L. (Ed.), *Inductive Logic Programming: 19th International Conference*, v. 5989, pp. 9–16, Leuven, Belgium, July 2010. Springer Berlin Heidelberg.
- [5] GAMA, J. A., KOSINA, P. “Learning Decision Rules from Data Streams”. In: *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence*, v. 2, *IJCAI'11*, pp. 1255–1260, Barcelona, Catalonia, Spain, July 2011. AAAI Press.
- [6] TSUNOYAMA, K., AMINI, A., STERNBERG, M., et al. “Scaffold hopping in drug discovery using inductive logic programming”, *Journal of chemical information and modeling*, v. 48, n. 5, pp. 949–957, 2008.
- [7] GETOOR, L., DIEHL, C. P. “Link mining: a survey”, *ACM SigKDD Explorations Newsletter*, v. 7, n. 2, pp. 3–12, 2005.
- [8] CRAVEN, M., DIPASQUO, D., FREITAG, D., et al. “Learning to construct knowledge bases from the World Wide Web”, *Artificial intelligence*, v. 118, n. 1-2, pp. 69–113, 2000.
- [9] MUGGLETON, S. “Stochastic Logic Programs”. In: *New Generation Computing*, v. 32, pp. 254–264, Cambridge, Massachusetts, EUA, January 1996. Academic Press.

- [10] RICHARDSON, M., DOMINGOS, P. “Markov logic networks”, *Machine Learning*, v. 62, n. 1, pp. 107–136, 2006.
- [11] DE RAEDT, L., KIMMIG, A., TOIVONEN, H. “ProbLog: A Probabilistic Prolog and Its Application in Link Discovery”. In: *Proceedings of the 20th International Joint Conference on Artificial Intelligence, IJCAI’07*, pp. 2468–2473, Hyderabad, India, 2007. Morgan Kaufmann Publishers Inc.
- [12] NEVILLE, J., JENSEN, D. “Relational Dependency Networks”, *Machine Learning*, v. 8, n. Mar, pp. 653–692, 2007.
- [13] WANG, W. Y., MAZAITIS, K., LAO, N., et al. “Efficient inference and learning in a large knowledge base”, *Machine Learning*, v. 100, n. 1, pp. 1–26, 2015.
- [14] RAEDT, L. D., KERSTING, K., NATARAJAN, S., et al. “Statistical relational artificial intelligence: Logic, probability, and computation”, *Synthesis Lectures on Artificial Intelligence and Machine Learning*, v. 10, n. 2, pp. 1–189, 2016.
- [15] MUGGLETON, S., DE RAEDT, L. “Inductive Logic Programming: Theory and Methods”, *Journal of Logic Programming*, v. 19/20, n. May, pp. 629–679, 1994.
- [16] MURPHY, K. P. *Machine Learning: a probabilistic perspective*. 1 ed. Cambridge, Massachusetts, EUA, The MIT Press, 2012.
- [17] RICHARDS, B. L., MOONEY, R. J. “Automated Refinement of First-Order Horn-Clause Domain Theories”, *Machine Learning*, v. 19, n. 2, pp. 95–131, 1995.
- [18] PAES, A., REVOREDO, K., ZAVERUCHA, G., et al. “Probabilistic First-Order Theory Revision from Examples”. In: *Inductive Logic Programming, 15th International Conference, ILP 2005*, pp. 295–311, Bonn, Germany, August 2005.
- [19] DUBOC, A. L., PAES, A., ZAVERUCHA, G. “Using the Bottom Clause and Modes Declarations on FOL Theory Revision from Examples”, *Machine Learning*, v. 76, n. 1, pp. 73–107, 2009.
- [20] PAES, A., ZAVERUCHA, G., COSTA, V. S. “On the use of stochastic local search techniques to revise first-order logic theories from examples”, *Machine Learning*, v. 106, n. 2, pp. 197–241, 2017.

- [21] HOEFFDING, W. “Probability Inequalities for Sums of Bounded Random Variables”, *Journal of the American Statistical Association*, v. 58, n. 301, pp. 13–30, 1963.
- [22] CARDOSO, P. M., ZAVERUCHA, G. “Comparative Evaluation of Approaches to Scale Up ILP”. In: *16th International Conference on Inductive Logic Programming (ILP 2006)*, pp. 37–39, Santiago de Compostela, August 2006. Corunna: UDC Press.
- [23] LOPES, C., ZAVERUCHA, G. “HTILDE: Scaling Up Relational Decision Trees for Very Large Databases”. In: *Proceedings of the 2009 ACM Symposium on Applied Computing, SAC '09*, pp. 1475–1479, Honolulu, Hawaii, USA, 2009. ACM.
- [24] SRINIVASAN, A., BAIN, M. “An empirical study of on-line models for relational data streams”, *Machine Learning*, v. 106, n. 2, pp. 243–276, 2017.
- [25] JENSEN, D. D., NEVILLE, J. “Linkage and Autocorrelation Cause Feature Selection Bias in Relational Learning”. In: *Machine Learning, Proceedings of the Nineteenth International Conference (ICML 2002)*, v. 2, pp. 259–266, Sydney, Australia, July 2002. Morgan Kaufmann.
- [26] MUGGLETON, S. “Inverse entailment and Progol”, *New Generation Computing*, v. 13, n. 3, pp. 245–286, 1995.
- [27] NATARAJAN, S., KHOT, T., KERSTING, K., et al. “Gradient-based boosting for statistical relational learning: The relational dependency network case”, *Machine Learning*, v. 86, n. 1, pp. 25–56, 2012.
- [28] KHOT, T., NATARAJAN, S., KERSTING, K., et al. “Gradient-based boosting for statistical relational learning: the Markov logic network and missing data cases”, *Machine Learning*, v. 100, n. 1, pp. 75–100, 2015.
- [29] DE RAEDT, L., DRIES, A., THON, I., et al. “Inducing Probabilistic Relational Rules from Probabilistic Examples”. In: *Proceedings of the 24th International Conference on Artificial Intelligence, IJCAI'15*, pp. 1835–1843, Buenos Aires, Argentina, July 2015. AAAI Press.
- [30] BELLODI, E., RIGUZZI, F. “Structure learning of probabilistic logic programs by searching the clause space”, *Theory and Practice of Logic Programming*, v. 15, n. 2, pp. 169–212, 2015.
- [31] SATO, T. “A Statistical Learning Method for Logic Programs with Distribution Semantics”. In: *In Proceedings of the 12th International Conference On*

Logic Programming (ICLP'95, pp. 715–729, Tokyo, Japan, June 1995. The MIT Press.

- [32] MENEZES, G. *HTILDE-RT: Um Algoritmo de Aprendizado de árvores de Regressão de Lógica de Primeira Ordem para Fluxos de Dados Relacionais*. M.Sc. dissertation, PESC, COPPE, Federal University of Rio de Janeiro, Rio de Janeiro, RJ, Brazil, 2011.
- [33] PAGE, L., BRIN, S., MOTWANI, R., et al. *The PageRank Citation Ranking: Bringing Order to the Web*. Technical Report 1999-66, Stanford InfoLab, 1999.
- [34] POON, H., DOMINGOS, P. “Joint Inference in Information Extraction”. In: *Proceedings of the 22Nd National Conference on Artificial Intelligence*, v. 1, *AAAI'07*, pp. 913–918, Vancouver, British Columbia, Canada, July 2007. AAAI Press.
- [35] DE RAEDT, L. *Logical and Relational Learning*. 1 ed. Berlin, Heidelberg, Springer-Verlag Berlin Heidelberg, 2008.
- [36] WROBEL, S. *Concept formation and knowledge revision*. 1 ed. Berlin, Heidelberg, Springer Science & Business Media, 2013.
- [37] BRACHMAN, R., LEVESQUE, H. *Knowledge Representation and Reasoning*. 1 ed. San Francisco, CA, USA, Morgan Kaufmann Publishers Inc., 2004. ISBN: 1558609326.
- [38] HORN, A. “On Sentences Which are True of Direct Unions of Algebras”, *The Journal of Symbolic Logic*, v. 16, n. 1, pp. 14–21, 1951.
- [39] WARREN, D. H. D., PEREIRA, L. M., PEREIRA, F. “Prolog - the Language and Its Implementation Compared with Lisp”, *SIGPLAN Not.*, v. 12, n. 8, pp. 109–115, 1977.
- [40] SHAN-HWEI NIENHUYS-CHENG, R. D. W. A. *Foundations of Inductive Logic Programming*. Lecture Notes in Computer Science 1228 : Lecture Notes in Artificial Intelligence. 1 ed. Berlin, Heidelberg, Springer-Verlag Berlin Heidelberg, 1997.
- [41] DE RAEDT, L., KIMMIG, A. “Probabilistic (logic) programming concepts”, *Machine Learning*, v. 100, n. 1, pp. 5–47, 2015.
- [42] KOLLER, D., FRIEDMAN, N. *Probabilistic Graphical Models: Principles and Techniques*. Adaptive Computation and Machine Learning series. 1 ed. Cambridge, Massachusetts, EUA, The MIT Press, 2009.

- [43] HECKERMAN, D., CHICKERING, D. M., MEEK, C., et al. “Dependency networks for inference, collaborative filtering, and data visualization”, *Journal of Machine Learning Research*, v. 1, n. Oct, pp. 49–75, 2000.
- [44] ANDERSEN, R., CHUNG, F., LANG, K. “Local Graph Partitioning Using PageRank Vectors”. In: *Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science*, FOCS '06, pp. 475–486, Washington, DC, USA, October 2006. IEEE Computer Society.
- [45] ANDERSEN, R., CHUNG, F., LANG, K. “Local partitioning for directed graphs using PageRank”. In: Bonato, A., Chung, F. R. K. (Eds.), *Algorithms and Models for the Web-Graph: 5th International Workshop, WAW 2007*, Springer Berlin Heidelberg, 2007.
- [46] TONG, H., FALOUTSOS, C., PAN, J.-Y. “Fast Random Walk with Restart and Its Applications”. In: *Proceedings of the Sixth International Conference on Data Mining*, ICDM '06, pp. 613–622, Washington, DC, USA, December 2006. IEEE Computer Society.
- [47] SHAPIRO, E. Y. *Algorithmic program debugging*. 1 ed. Cambridge, Massachusetts, EUA, The MIT Press, 1983.
- [48] HULTEN, G., ABE, Y., DOMINGOS, P. “Mining Massive Relational Database”. In: *Proceedings of the IJCAI-2003 Workshop on Learning Statistical Models from Relational Data*, pp. 53–60, Acapulco, Mexico, August 2003.
- [49] BLOCKEEL, H., DE RAEDT, L. “Top-down Induction of First-order Logical Decision Trees”, *Artificial Intelligence*, v. 101, n. 1-2, pp. 285–297, 1998.
- [50] BLOCKEEL, H., DE RAEDT, L., JACOBS, N., et al. “Scaling Up Inductive Logic Programming by Learning from Interpretations”, *Data Mining and Knowledge Discovery*, v. 3, n. 1, pp. 59–93, 1999.
- [51] QUINLAN, J. “Learning Logical Definitions from Relations”, *Machine Learning*, v. 5, n. 3, pp. 239–266, August 1990.
- [52] SRINIVASAN, A. *The aleph manual*. Relatório técnico, 2001.
- [53] RICHARDS, B. L., MOONEY, R. J. “Learning Relations by Pathfinding”. In: *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92)*, pp. 50–55, San Jose, CA, July 1992.

- [54] DAWID, A. P. “Present Position and Potential Developments: Some Personal Views: Statistical Theory: The Prequential Approach”, *Journal of the Royal Statistical Society. Series A (General)*, v. 147, n. 2, pp. 278–292, 1984.
- [55] DRIES, A., KIMMIG, A., MEERT, W., et al. “ProbLog2: Probabilistic Logic Programming”. In: *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD, Proceedings, Part III*, v. 9286, *LCNS*, pp. 312–315, Porto, Portugal, September 2015. Springer.
- [56] THORNTON, C., HUTTER, F., HOOS, H. H., et al. “Auto-WEKA: Combined Selection and Hyperparameter Optimization of Classification Algorithms”. In: *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '13*, pp. 847–855, Chicago, Illinois, USA, August 2013. ACM.
- [57] MITCHELL, T., COHEN, W., HRUSCHKA, E., et al. “Never-Ending Learning”. In: *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence (AAAI-15)*, pp. 2302–2310, Austin, Texas, USA, January 2015.
- [58] GAMA, J. A., ŽLIOBAITĖ, I., BIFET, A., et al. “A Survey on Concept Drift Adaptation”, *ACM Comput. Surv.*, v. 46, n. 4, pp. 1053–1060, 2014.