Machine Learning (2019) 108:1111–1135
https://doi.org/10.1007/s10994-018-5750-0

Lifted discriminative learning of probabilistic logic programs

Arnaud Nguembang Fadja[1] · **Fabrizio Riguzzi**[2]

Received: 4 December 2017 / Accepted: 1 August 2018 / Published online: 20 August 2018

Abstract
Probabilistic logic programming (PLP) provides a powerful tool for reasoning with uncertain relational models. However, learning probabilistic logic programs is expensive due to the high cost of inference. Among the proposals to overcome this problem, one of the most promising is lifted inference. In this paper we consider PLP models that are amenable to lifted inference and present an algorithm for performing parameter and structure learning of these models from positive and negative examples. We discuss parameter learning with EM and LBFGS and structure learning with LIFTCOVER, an algorithm similar to SLIPCOVER. The results of the comparison of LIFTCOVER with SLIPCOVER on 12 datasets show that it can achieve solutions of similar or better quality in a fraction of the time.

Keywords Statistical relational learning · Probabilistic inductive logic programming · Probabilistic logic programming · Lifted inference · Expectation maximization

1 Introduction

Probabilistic Logic Programming (PLP) is a powerful tool for reasoning in uncertain relational domains that is gaining popularity in Statistical Relational Artificial Intelligence (StarAI) due to its expressiveness and intuitiveness. PLP has been applied successfully to a variety of fields, such as natural language processing (Sato and Kubota 2015; Riguzzi et al. 2017b; Nguembang Fadja and Riguzzi 2017), bioinformatics (Mørk and Holmes 2012; De Raedt et al. 2007; Sato and Kameya 1997), link prediction in social networks (Meert et al. 2010), entity resolution (Riguzzi 2014) and model checking (Gorlin et al. 2012).

Among the different approaches that have been proposed for representing probabilistic information in Logic Programming, the distribution semantics (Sato 1995) achieved wide

Editors: Nicolas Lachiche and Christel Vrain.

✉ Fabrizio Riguzzi
fabrizio.riguzzi@unife.it
http://mcs.unife.it/∼friguzzi/

Arnaud Nguembang Fadja
arnaud.nguembafadja@unife.it

[1] Dipartimento di Ingegneria, University of Ferrara, Via Saragat 1, 44122 Ferrara, Italy

[2] Dipartimento di Matematica e Informatica, University of Ferrara, Via Saragat 1, 44122 Ferrara, Italy

use (Raedt and Kimmig 2015; Riguzzi and Swift 2018) and underlies many languages, such as Independent Choice Logic (Poole 2000), PRISM (Sato 1995), Logic Programs with Annotated Disjunctions (Vennekens et al. 2004a) and ProbLog (De Raedt et al. 2007). While these languages differ syntactically, they have the same expressive power, as there are linear transformations among them (Vennekens and Verbaeten 2003).

The problem of learning probabilistic logic program has received considerable attention. However, PLP usually require expensive learning procedures due to the high cost of inference. SLIPCOVER (Bellodi and Riguzzi 2015) for example performs structure learning of probabilistic logic programs using knowledge compilation for parameter learning: the expectations needed for the EM parameter learning algorithm are computed using the Binary Decision Diagrams (BDDs) that are built for inference. Compiling explanations for queries into BDDs has a #P cost in the number of random variables. Lifted inference (Poole 2003) was proposed for improving the performance of reasoning in probabilistic relational models by reasoning on whole populations of individuals instead of considering each individual separately. For example, consider the following Logic Program with Annotated Disjunctions (adapted from Raedt and Kimmig (2015)):

$$popular(X) : p :- friends(X, Y), famous(Y)$$

which states that a person is popular with probability $p \in [0, 1]$ if he has a famous friend. To compute the probability that $john$ is popular, let n be the number of $john$'s famous friends. $john$ is not popular if the rule doesn't fire (the head is not implied) for any his famous friends therefore $P(\neg popular(john)) = (1 - p)^n$. So $P(popular(john)) = 1 - (1 - p)^n$. To compute this probability, we do not need to know information about $john$'s individual famous friends, we just need to know their number. Computing this value has a cost logarithmic in n, as computing a^n is $\Theta(\log n)$ with the "square and multiply" algorithm (Gordon 1998), rather than #P in n.

Various algorithms have been proposed for performing lifted inference for PLP (Van den Broeck et al. 2014; Bellodi et al. 2014), see Riguzzi et al. (2017a) for a survey and comparison of the approaches.

In this paper we consider a simple PLP language (called *liftable PLP*) where programs contain clauses with a single annotated atom in the head and the predicate of this atom is the same for all clauses. In this case, all the above approaches for lifted inference coincide and reduce to a computation similar to the one of the example above.

For this language, we discuss how to perform discriminative parameter learning by using EM or optimizing the likelihood with Limited-memory BFGS (LBFGS) (Nocedal 1980).

A previous approach for performing lifted learning (Haaren et al. 2016) targeted generative learning for Markov Logic Networks, so it cannot be applied directly to PLP.

We also present LIFTCOVER for "LIFTed slipCOVER", an algorithm for performing discriminative structure learning of liftable PLP programs obtained from SLIPCOVER (Bellodi and Riguzzi 2015) by simplifying structure search and replacing parameter learning with one of the specialized approaches. We thus obtain LIFTCOVER-EM and LIFTCOVER-LBFGS that performs EM and LBFGS respectively.

We compare LIFTCOVER-EM, LIFTCOVER-LBFGS and SLIPCOVER on 12 datasets. The results show that LIFTCOVER-EM is nearly always faster and more accurate than SLIPCOVER while LIFTCOVER-LBFGS is often faster and similarly accurate than SLIPCOVER.

Liftable PLP can also be seen as a language where the contributions of different groundings of a clause and of different clauses are combined using a noisy-OR combining rule and is therefore very much related to languages such as First-Order Probabilistic Logic (Koller

and Pfeffer 1997), Bayesian Logic Programs (Kersting and De Raedt 2002) and the First-Order Conditional Influence Language (Natarajan et al. 2009). Liftable PLP can be seen as a special case of each of these languages in which simpler inference and learning algorithms can be used. The experimental results show that the algorithm still yield good quality results notwithstanding the language restrictions.

The paper is organized as follows: Sect. 2 introduces PLP under the distribution semantics, Sect. 3 presents the liftable PLP language, Sects. 4 and 5 illustrate parameter and structure learning respectively, Sect. 6 discusses related work, Sect. 7 describes the experiments performed and Sect. 8 concludes the paper.

2 Probabilistic logic programming

We consider Probabilistic Logic Programming under the distribution semantics (Sato 1995) for integrating logic programming with probability. Languages with this semantics were shown expressive enough to represent a wide variety of domains (Riguzzi et al. 2016; Alberti et al. 2017). A program in a language adopting the distribution semantics defines a probability distribution over normal logic programs called *instances* or *worlds*. Each normal program is assumed to have a total well-founded model (Gelder et al. 1991). Then the distribution is extended to queries and the probability of a query is obtained by marginalizing the joint distribution of the query and the programs.

A PLP language under the distribution semantics with a general syntax is *Logic Programs with Annotated Disjunctions* (LPADs) (Vennekens et al. 2004b). We present here the semantics of LPADs for the case of no function symbols, if function symbols are allowed see Riguzzi (2016). However, function symbols are usually absent from the learning problems we consider, where the task is to learn a knowedge-based classifier. Function symbols are necessary for program induction which however is outside the scope of this paper.

In LPADs, heads of clauses are disjunctions in which each atom is annotated with a probability. Let us consider an LPAD T with n clauses: $T = \{C_1, \ldots, C_n\}$. Each clause C_i takes the form: $h_{i1} : \Pi_{i1}; \ldots; h_{iv_i} : \Pi_{iv_i} :- b_{i1}, \ldots, b_{iu_i}$, where $h_{i1}, \ldots, h_{iv_i}$ are logical atoms, $b_{i1}, \ldots, b_{iu_i}$ are logical literals and $\Pi_{i1}, \ldots, \Pi_{iv_i}$ are real numbers in the interval $[0, 1]$ that sum to 1. $b_{i1}, \ldots, b_{iu_i}$ is indicated with $body(C_i)$. Note that if $v_i = 1$ the clause corresponds to a non-disjunctive clause. We also allow clauses where $\sum_{k=1}^{v_i} \Pi_{ik} < 1$: in this case the head of the annotated disjunctive clause implicitly contains an extra atom $null$ that does not appear in the body of any clause and whose annotation is $1 - \sum_{k=1}^{v_i} \Pi_{ik}$. We denote by $ground(T)$ the grounding of an LPAD T.

Each grounding $C_i\theta_j$ of a clause C_i corresponds to a random variable X_{ij} with values $\{1, \ldots, v_i\}$ where v_i is the number of head atoms of C_i. The random variables X_{ij} are independent of each other. An *atomic choice* (Poole 1997) is a triple (C_i, θ_j, k) where $C_i \in T$, θ_j is a substitution that grounds C_i and $k \in \{1, \ldots, v_i\}$ identifies one of the head atoms. In practice (C_i, θ_j, k) corresponds to an assignment $X_{ij} = k$.

A *selection* σ is a set of atomic choices that, for each clause $C_i\theta_j$ in $ground(T)$, contains an atomic choice (C_i, θ_j, k). Let us indicate with S_T the set of all selections. A selection σ identifies a normal logic program l_σ defined as $l_\sigma = \{(h_{ik} \leftarrow body(C_i))\theta_j | (C_i, \theta_j, k) \in \sigma\}$. l_σ is called an *instance*, *possible world* or simply *world* of T. Since the random variables associated with ground clauses are independent, we can assign probabilities to instances: $P(l_\sigma) = \prod_{(C_i,\theta_j,k)\in\sigma} \Pi_{ik}$.

We consider only *sound* LPADs where, for each selection σ in S_T, the well-founded model of the program l_σ chosen by σ is two-valued. We write $l_\sigma \models q$ to mean that the ground atomic query q is true in the well-founded model of the program l_σ. Since the well-founded model of each world is two-valued, q can only be true or false in l_σ.

We denote the set of all instances by L_T. Let $P(L_T)$ be the distribution over instances. Consider a ground atomic query q. The probability of q given an instance l is $P(q|l) = 1$ if $l \models q$ and 0 otherwise. The probability of q is given by

$$P(q) = \sum_{l \in L_T} P(q, l) = \sum_{l \in L_T} P(q|l)P(l) = \sum_{l \in L_T : l \models q} P(l) \tag{1}$$

Computing $P(q)$ by generating all the worlds is impractical because their number is exponential in the number of ground probabilistic clauses. A successful alternative approach finds explanations for the query q (De Raedt et al. 2007), where an explanation is a set of clause choices that are sufficient for entailing the query. Explanations are then encoded as a Boolean formula and the problem is reduced to that of computing the probability that the formula is true given the probabilities of being true of all the (mutually independent) Boolean random variables. This is the *disjoint-sum* problem so called because it can be solved by finding a DNF formula where all the disjuncts are mutually exclusive. The problem has complexity #P (Valiant 1979) in the number of random variables so it is very difficult but problems of significant size have been solved in practice using *knowledge compilation* (Darwiche and Marquis 2002), i.e. converting the Boolean formula into a language from which the computation of the probability is polynomial (De Raedt et al. 2007; Riguzzi and Swift 2011), such as Binary Decision Diagrams.

3 Liftable PLP

We restrict the language of LPADs by allowing only clauses of the form

$$C_i = h_i : \Pi_i :- b_{i1}, \ldots, b_{iu_i}$$

in the program where all the clauses share the same predicate for the single atom in the head, let us call this predicate $target/a$ with a the arity. The literals in the body have predicates other than $target/a$ and are defined by facts and rules that are certain, i.e., they have a single atom in the head with probability 1. The predicate $target/a$ is called *target* and the others *input predicates*. Suppose there are n probabilistic clauses of the form above in the program. We call this language *liftable PLP*.

The problem is to compute the probability of a ground instantiation q of $target/a$. This can be done at the lifted level. We should first find the number of ground instantiations of clauses for $target/a$ such that the body is true and the head is equal to q. Suppose there are m_i such instantiations $\{\theta_{i1}, \ldots, \theta_{im_i}\}$, for rule C_i for $i = 1, \ldots, n$. Each instantiation θ_{ij} corresponds to a random variable X_{ij} taking values 1 with probability Π_i and 0 with probability $1 - \Pi_i$. The query q is true if at least one of the random variables for a rule takes value 1: $q = true \Leftrightarrow \bigvee_{i=1}^{n} \bigvee_{j=1}^{m_i} (X_{ij} = 1)$. In other words q is false only if no random variable takes value 1. All the random variables are mutually independent so the probability that none takes value 1 is $\prod_{i=1}^{n} \prod_{j=1}^{m_i} (1 - \Pi_i) = \prod_{i=1}^{n} (1 - \Pi_i)^{m_i}$ and the probability of q being true is $P(q) = 1 - \prod_{i=1}^{n} (1 - \Pi_i)^{m_i}$. So once the number of clause instantiations with the body true is known, the probability of the query can be computed in logarithmic time. Note that finding an assignment of a set of logical variables that makes a conjunction true

is an NP-complete problem (Kietz and Lübbe 1994), therefore computing the probability of the query may be prohibitive.

However, when using knowledge compilation, to the cost of finding the assignment, we must sum the cost of performing the compilation, that is #P in the number of satisfying logical variables assignments (clause instantiations with the body true). Therefore inference in liftable PLP is significantly cheaper than in the general case. Moreover, in machine learning the conjunctions are usually short and the knowledge compilation cost dominates.

The general language of PLP is necessary when the user wants to induce a knowledge base or an ontology regarding the domain. In that case, the possibility of having more than one head, possibly involving more than one predicate, and the possibility of learning probabilistic rules for subgoals is useful because the resulting program can thus represent and organize general knowledge about the domain. Moreover, the resulting program can then be used for answering different types of queries instead of being restricted to answering queries about a single predicate. This is similar to the problem of learning multiple predicates in Inductive Logic Programming. While this problem has received considerable attention, most work concentrated on learning a single predicate, for example systems such as FOIL (Quinlan 1990), Progol (Muggleton 1995) and Aleph (Srinivasan 2007) learn a single predicate at a time. Furthermore, most benchmark datasets are focused on predicting the truth value of atoms for a single predicate. For example, all the datasets we consider in the experimental evaluation, Sect. 7, include positive and negative example for a single predicate.

We believe that the problem of inducing general knowledge bases will become very important in the near future because of the growth of the Semantic Web: more and more data is being published on the web but ontologies are often shallow. If we want to be able to provide answers for complex queries given the available data, we need deep and complex knowledge bases and learning them appears to be a promising direction.

We can picture the dependence of the random variable q associated with the query from the random variables of clause groundings with the body true as in Fig. 1. Here the conditional probability table of q is that of an or: $P(q) = 1$ if at least one of its parents is equal to 1 and 0 otherwise. The variables from clause groundings are

$$\{X_{11}, \ldots, X_{1m_1}, X_{21}, \ldots, X_{2m_2}, \ldots, X_{n1}, \ldots, X_{nm_n}\}.$$

These are parentless variables, with X_{ij} having the conditional probability table (CPT) $P(X_{ij} = 1) = \Pi_i$ and $P(X_{ij} = 0) = 1 - \Pi_i$.

This is an example of a *noisy-OR* model (Good 1961; Pearl 1988): an event is associated to a number of conditions each of which alone can cause the event to happen. The conditions/causes are noisy, i.e., they have a probability of being active and they are mutually unconditionally independent. A liftable PLP program encodes a noisy-OR model where the

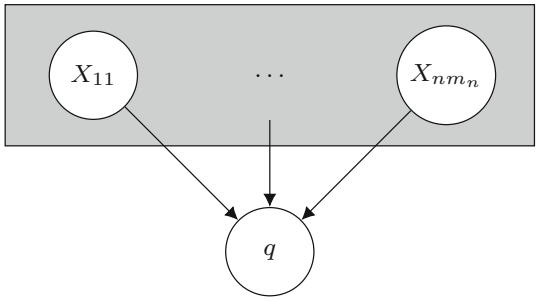

Fig. 1 Bayesian Network representing the dependency between the query q and the random variables associated with groundings of the clauses with the body true

event is the query q being true and causes are the ground instantiations of the clauses that have the body true: each can cause the query to be true with the probability given by the clause annotation.

Example 1 Let us consider the UW-CSE domain (Kok and Domingos 2005) where the objective is to predict the "advised by" relation between students and professors. In this case the target predicate is $advisedby/2$ and a program for predicting such predicate may be

$$\begin{aligned} &advisedby(A, B) : 0.4 :- \\ &\quad student(A), professor(B), publication(C, A), publication(C, B). \\ &advisedby(A, B) : 0.5 :- \\ &\quad student(A), professor(B), ta(C, A), taughtby(C, B). \end{aligned}$$

where $publication(A, B)$ means that A is a publication with author B, $ta(C, A)$ means that A is a teaching assistant for course C and $taughtby(C, B)$ means that course C is taught by B. The probability that a student is advised by a professor depends on the number of joint publications and the number of courses the professor teaches where the student is a TA, the higher these numbers the higher the probability.

Suppose we want to compute the probability of $q = advisedby(harry, ben)$ where $harry$ is a student, ben is a professor, they have 4 joint publications and ben teaches 2 courses where $harry$ is a TA. Then the first clause has 4 groundings with head q where the body is true, the second clause has 2 groundings with head q where the body is true and $P(advisedby(harry, ben)) = 1 - (1 - 0.4)^4(1 - 0.5)^2 = 0.9676$.

4 Parameter learning

Learning problems can be divided into discriminative and generative (Koller and Friedman 2009). Given input data $\mathbf{x}$, *generative learning* means learning the joint distribution $P(\mathbf{x})$. *Discriminative learning* instead means identifying one of the data variables y which we want to predict and learning the conditional distribution $P(y|\mathbf{x})$. If y is Boolean, as in our case, it is natural to identify values $y = 1$ as positive examples and values $y = 0$ as negative examples. In generative learning identifying positive and negative examples is less obvious. We consider discriminative learning because we want to predict only atoms for the target predicate, while the atoms for the input predicates are assumed as given.

The problem of discriminative learning of the parameters of a liftable PLP $T = \{C_1, \ldots, C_n\}$ can be expressed as follows: given a liftable PLP T, a set

$$E^+ = \{e_1, \ldots, e_Q\}$$

of positive examples (ground atoms for the target predicate) and a set

$$E^- = \{e_{Q+1}, \ldots, e_R\}$$

of negative examples (ground atoms for the target predicate) and background knowledge B, find the parameters of T such that the likelihood

$$L = \prod_{q=1}^{Q} P(e_q) \prod_{r=Q+1}^{R} P(\neg e_r)$$

is maximized. The likelihood is given by the product of the probability of each example.

The background knowledge B is a normal logic program defining the input predicates with certainty. In the simplest case it is a set of ground facts, i.e., an interpretation I, describing the domain by means of the observed facts for the input predicates. It also called a *mega-example* because we can consider the case where we have a set of interpretations $\mathcal{I} = \{I_1, \ldots, I_U\}$ each describing a different sub-domain from the universe considered. In that case, each mega-example I_u will be associated with its set of positive and negative examples E_u^+ and E_u^- that are to be evaluated against I_u. These examples can be opportunely encoded in I_u so that the training data is represented fully by $\mathcal{I}$, for example by encoding positive examples as facts for the target predicate and negative examples as facts of the form $neg(e_r)$ with $e_r \in E_u^-$. This is a common situation in StarAI.

The likelihood can be unfolded to

$$L = \prod_{q=1}^{Q} \left(1 - \prod_{l=1}^{n} (1 - \Pi_l)^{m_{lq}}\right) \prod_{r=Q+1}^{R} \prod_{l=1}^{n} (1 - \Pi_l)^{m_{lr}} \tag{2}$$

where m_{iq} (m_{ir}) is the number of instantiations of C_i whose head is e_q (e_r) and whose body is true. We can aggregate the negative examples

$$L = \prod_{l=1}^{n} (1 - \Pi_l)^{m_{l-}} \prod_{q=1}^{Q} \left(1 - \prod_{l=1}^{n} (1 - \Pi_l)^{m_{lq}}\right) \tag{3}$$

where $m_{l-} = \sum_{r=Q+1}^{R} m_{lr}$.

We can maximize L using an Expectation Maximization (EM) algorithm (Dempster et al. 1977) since the X_{ij} variables are hidden. To perform EM, we need to compute the conditional probabilities $P(X_{ij} = 1|e)$ and $P(X_{ij} = 1|\neg e)$ where e is an example (a ground atom) and X_{ij} are its parents.

Alternatively, we can use gradient descent to optimize L. In this case, we need to compute the gradient of the likelihood with respect to the parameters. In the following subsections we consider each method in turn.

4.1 EM algorithm

The EM algorithm (Dempster et al. 1977) finds the maximum likelihood estimates of parameters in models with hidden variables by alternating between an expectation (E) step and a maximization (M) step. The algorithm starts with random values for the parameters. Then, in the E step, it computes the distribution of values of the hidden variables given the observed ones and the current value of the parameters. In the M step it computes the value of the parameters that maximize the expected log-likelihood (LL). Then the parameters are updated and the algorithm goes back to the E step, stopping when the log-likelihood does not improve anymore.

To perform EM, we need to compute the distribution of the hidden variables given the observed ones, in our case $P(X_{ij} = 1|e)$ and $P(X_{ij} = 1|\neg e)$. e is a single example that is a ground atom for the target predicate. The X_{ij} variables are relative to the ground instantiations of the probabilistic clauses whose body is true when the head is unified with e. Different examples don't share clause groundings, as the constants in them are different. Therefore the X_{ij} variables are not shared among examples.

Let us now compute $P(X_{ij} = 1, e)$:

$$P(X_{ij} = 1, e) = P(e|X_{ij} = 1)P(X_{ij} = 1) = P(X_{ij} = 1) = \Pi_i$$

since $P(e|X_{ij} = 1) = 1$, so

$$P(X_{ij} = 1|e) = \frac{P(X_{ij} = 1, e)}{P(e)} = \frac{\Pi_i}{1 - \prod_{i=1}^{n}(1 - \Pi_i)^{m_i}} \tag{4}$$

$$P(X_{ij} = 0|e) = 1 - \frac{\Pi_i}{1 - \prod_{i=1}^{n}(1 - \Pi_i)^{m_i}} \tag{5}$$

$P(X_{ij} = 1|\neg e)$ is given by

$$P(X_{ij} = 1, \neg e) = P(\neg e|X_{ij} = 1)P(X_{ij} = 1) = 0$$

since $P(\neg e|X_{ij} = 1) = 0$, so

$$P(X_{ij} = 1|\neg e) = 0 \tag{6}$$

$$P(X_{ij} = 0|\neg e) = 1. \tag{7}$$

This leads to the EM algorithm of Algorithm 1, with the EXPECTATION and MAXIMIZATION functions shown in Algorithms 2 and 3.

Function EM stops when the difference between the current value of the LL and the previous one is below a given threshold or when such a difference relative to the absolute value of the current one is below a given threshold.

Function EXPECTATION updates, for each clause C_i, two counters, c_{i1} and c_{i2}, one for each value of the random variables X_{ij} associated with clause C_i. These counters accumulate the values of the conditional probability of the values of the hidden variables. The counters are updated taking into account first the negative examples and then the positive ones. Negative examples can be considered in bulk because their contribution is the same for all groundings of all examples, while positive examples must be considered one by one, for each one updating the counters of all the clauses.

Function Maximization then simply computes the new values of the parameters by dividing c_{i1} by the sum of c_{i1} and c_{i2}.

Algorithm 1 Function EM

```
1: function EM(restarts, max_iter, ϵ, δ)
2:     BestLL ← −inf
3:     BestPar ← []
4:     for j ← 1, restarts do
5:         for i ← 1, n do                          ▷ n: number of rules
6:             Π_i ← random
7:         end for
8:         LL = −inf
9:         iter ← 0
10:        repeat
11:            iter ← iter + 1
12:            LL_0 = LL
13:            LL = EXPECTATION
14:            MAXIMIZATION
15:        until LL − LL_0 < ϵ ∨ LL − LL_0 < −LL · δ ∨ iter > max_iter
16:        if LL > BestLL then
17:            BestLL ← LL
18:            BestPar ← [Π_1, ..., Π_n]
19:        end if
20:    end for
21:    return BestLL, BestPar
22: end function
```

Algorithm 2 Function Expectation

```
function EXPECTATION
    LL ← Σ_{i∈Rules} m_{i−} log(1 − Π_i)
                          ▷ m_{i−}: total number of groundings of rule i with the body true in a negative example
    for i ← 1, n do
        c_{i1} ← 0
        c_{i2} ← m_{i−}
    end for
    for r ← 1, P do                                   ▷ P: number of positive examples
        probex ← 1 − Π_{i∈Rules} (1 − Π_i)^{m_{ir}}
                          ▷ m_{ir}: number of groundings of rule i with the body true in example r
        LL ← LL + log probex
        for i ← 1, n do
            condp ← Π_i / probex
            c_{i1} ← c_{i1} + m_{ir} condp
            c_{i2} ← c_{i2} + m_{ir} (1 − condp)
        end for
    end for
    return LL
end function
```

Algorithm 3 Function Maximization

```
procedure MAXIMIZATION
    for i ← 1, n do
        Π_i = c_{i1} / (c_{i1} + c_{i2})
    end for
end procedure
```

4.2 Gradient-based optimization

Gradient-based methods include gradient descent and its derivatives, such as second-order methods like Limited-memory BFGS (LBFGS) (Nocedal 1980), an optimization algorithm in the family of quasi-Newton methods that approximates the Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm using a limited amount of computer memory.

To perform gradient-based optimization we need to compute the partial derivatives of the likelihood with respect to the parameters. Let us recall the likelihood

$$L = \prod_{l=1}^{n} (1 - \Pi_l)^{m_{l-}} \prod_{q=1}^{Q} \left(1 - \prod_{l=1}^{n} (1 - \Pi_l)^{m_{lq}} \right) \tag{8}$$

where $m_{l-} = \sum_{r=Q+1}^{R} m_{lr}$. Its partial derivative with respect to Π_i is

$$\frac{\partial L}{\partial \Pi_i} = \frac{\partial \prod_{l=1}^{n} (1 - \Pi_l)^{m_{l-}}}{\partial \Pi_i} \prod_{q=1}^{Q} \left(1 - \prod_{l=1}^{n} (1 - \Pi_l)^{m_{lq}} \right) + \prod_{l=1}^{n} (1 - \Pi_l)^{m_{l-}} \frac{\partial \prod_{q=1}^{Q} \left(1 - \prod_{l=1}^{n} (1 - \Pi_l)^{m_{lq}} \right)}{\partial \Pi_i}$$

$$= -m_{i-}(1-\Pi_i)^{m_{i-}-1}\prod_{l=1,l\neq i}^{n}(1-\Pi_l)^{m_{l-}}\prod_{q=1}^{Q}\left(1-\prod_{l=1}^{n}(1-\Pi_l)^{m_{lq}}\right)$$
$$+\prod_{l=1}^{n}(1-\Pi_l)^{m_{l-}}\sum_{q=1}^{Q}m_{iq}(1-\Pi_i)^{m_{iq}-1}\prod_{l=1,l\neq i}^{n}(1-\Pi_l)^{m_{lq}}$$
$$\cdot\prod_{q'=1,q'\neq q}^{Q}\left(1-\prod_{l=1}^{n}(1-\Pi_l)^{m_{lq'}}\right) \tag{9}$$

for the differentiation product rule, and

$$\frac{\partial L}{\partial \Pi_i} = -m_{i-}(1-\Pi_i)^{m_{i-}-1}\prod_{l=1,l\neq i}^{n}(1-\Pi_l)^{m_{l-}}\frac{(1-\Pi_i)^{m_{i-}}}{(1-\Pi_i)^{m_{i-}}}$$
$$\prod_{q=1}^{Q}\left(1-\prod_{l=1}^{n}(1-\Pi_l)^{m_{lq}}\right)$$
$$+\prod_{l=1}^{n}(1-\Pi_l)^{m_{l-}}\sum_{q=1}^{Q}m_{iq}\frac{\prod_{l=1}^{n}(1-\Pi_l)^{m_{lq}}}{1-\Pi_i}$$
$$\prod_{q'=1,q'\neq q}^{Q}\left(1-\prod_{l=1}^{n}(1-\Pi_l)^{m_{lq'}}\right)\cdot\frac{1-\prod_{l=1}^{n}(1-\Pi_l)^{m_{lq}}}{1-\prod_{l=1}^{n}(1-\Pi_l)^{m_{lq}}} \tag{10}$$

by dividing and multiplying for $(1-\Pi_i)^{m_{i-}}$, $(1-\Pi_i)$ and $1-\prod_{l=1}^{n}(1-\Pi_l)^{m_{lq}}$ various factors. Then

$$\frac{\partial L}{\partial \Pi_i} = -\frac{m_{i-}(1-\Pi_i)^{m_{i-}-1}L}{(1-\Pi_i)^{m_{i-}}}+\sum_{q=1}^{Q}\frac{m_{iq}\prod_{l=1}^{n}(1-\Pi_l)^{m_{lq}}L}{(1-\Pi_i)(1-\prod_{l=1}^{n}(1-\Pi_l)^{m_{lq}})}$$
$$= -\frac{m_{i-}L}{1-\Pi_i}+\sum_{q=1}^{Q}\frac{m_{iq}\prod_{l=1}^{n}(1-\Pi_l)^{m_{lq}}L}{(1-\Pi_i)(1-\prod_{l=1}^{n}(1-\Pi_l)^{m_{lq}})}$$
$$= \frac{L}{1-\Pi_i}\left(\sum_{q=1}^{Q}\frac{m_{iq}\prod_{l=1}^{n}(1-\Pi_l)^{m_{lq}}}{1-\prod_{l=1}^{n}(1-\Pi_l)^{m_{lq}}}-m_{i-}\right)$$
$$= \frac{L}{1-\Pi_i}\left(\sum_{q=1}^{Q}m_{iq}\frac{1-P(e_q)}{P(e_q)}-m_{i-}\right)$$
$$= \frac{L}{1-\Pi_i}\left(\sum_{q=1}^{Q}m_{iq}\left(\frac{1}{P(e_q)}-1\right)-m_{i-}\right) \tag{11}$$

by simple algebra.

The equation $\frac{\partial L}{\partial \Pi_i}=0$ does not admit a closed form solution, not even where there is a single clause, so we must use optimization to find the maximum of L.

5 Structure learning

The discriminative structure learning problem can be expressed as: given a set $E^+ = \{e_1, \ldots, e_Q\}$ of positive examples, a set $E^- = \{e_{Q+1}, \ldots, e_R\}$ of negative examples and a background knowledge B, find a liftable PLP T such that the likelihood is maximized. The background knowledge B may be a normal logic program defining all the predicates except the target (the input predicates).

We solve this problem by first identifying good clauses guided by the log likelihood (LL) of the data. Clauses are found by a top-down beam search. The refinement operator adds a literal to the body of the current clause, the literal is taken from a *bottom clause* built as in Progol (Muggleton 1995). The set of clauses found in this phase is then considered as a single theory and parameter learning is performed on it. Then the clauses with a parameter below a user define threshold *WMin* are discarded and the theory is returned. The resulting algorithm, LIFTCOVER, is very similar to SLIPCOVER (Bellodi and Riguzzi 2015). The difference between the two is that LIFTCOVER uses lifted parameter learning instead of the EM algorithm over BDDs of Bellodi and Riguzzi (2013). Moreover they use a different approach for performing the selection of the rules to be included in the final model: while SLIPCOVER does a hill-climbing search in which it adds one clause at a time to the theory, learns the parameters and keeps the clause if the LL is smaller than before, LIFTCOVER learns the parameters for the whole set of clauses found during the search in the space of clauses. This is allowed by the fact that parameter learning in LIFTCOVER is much faster so it can be applied to large theories. Then rules with a small parameter can be discarded as they provide small contributions to the predictions. In practice structure search is thus performed in LIFTCOVER by parameter learning, as is done for example in Nishino et al. (2014), Wang et al. (2014).

Algorithm 4 shows the main LIFTCOVER function. Line 2 calls INITIALBEAM (see Algorithm 5) that builds an initial beam $Beam$ consisting of bottom clauses.

The set ot literals allowed in the bottom clause is defined by the *language bias* that is expressed by means of *mode* declarations. They are atoms of the form $modeh(r, s)$ (head declarations) or $modeb(r, s)$ (body declaration), where s, the *schema*, is a ground literal and r is an integer called the *recall*. A schema is a template for literals in the head or body of a

Algorithm 4 Function LIFTCOVER

```
1: function LIFTCOVER(NB, NI, NInt, NS, NA, NV)
2:     Beam =INITIALBEAM(NInt, NS, NA)                          ▷ Bottom clauses building
3:     CC ← ∅
4:     Steps ← 1
5:     NewBeam ← []
6:     repeat
7:         Remove the first couple ((Cl, Literals), LL) from Beam        ▷ Remove the first clause
8:         Refs ←CLAUSEREFINEMENTS((Cl, Literals, NV))   ▷ Find all refinements Refs of (Cl, Literals)
9:         for all (Cl', Literals') ∈ Refs do
10:            (LL'', {Cl''}) ←LEARNWEIGHTS(I, {Cl'})
11:            NewBeam ←INSERT((Cl'', Literals'), LL'', NewBeam, NB)   ▷ The refinement is inserted in the beam in
    order of likelihood, possibly removing the last clause if the size of the beam NB is exceeded
12:            CC ← CC ∪ {Cl'}
13:        end for
14:        Beam ← NewBeam
15:        Steps ← Steps + 1
16:    until Steps > NI or Beam is empty
17:    (LL, Th) ←LEARNWEIGHTS(CC)
18:    Remove from Th the clauses with a weight smaller than WMin
19:    return Th
20: end function
```

Algorithm 5 Function INITIALBEAM

```
1: function INITIALBEAM(NInt, NS, NA)
2:     Beam ← []
3:     for all modeh declarations modeh(r, s) do
4:         for i = 1 → NInt do
5:             Select randomly a mega-example I
6:             for j = 1 → NA do
7:                 Select randomly an atom h from I matching schema(s)
8:                 Bottom clause BC ←SATURATION(h, r, NS), let BC be Head :− Body
9:                 Beam ← [((h : 0.5 :− true, Body), −∞)|Beam]
10:            end for
11:        end for
12:    end for
13:    return Beam
14: end function
```

clause and can contain special placemarker terms of the form `#type`, `+type` and `-type`, which stand, respectively, for ground terms, input variables and output variables of a type. An input variable in a body literal of a clause must be either an input variable in the head or an output variable in a preceding body literal in the clause. If M is a set of mode declarations, $L(M)$ is the *language of* M, i.e. the set of clauses $\{C = h :- b_1, \ldots, b_m\}$ such that the head atom h (resp. body literals b_i) is obtained from some head (resp. body) declaration in M by replacing all `#type` placemarkers with ground terms and all `+type` (resp. `-type`) placemarkers with input (resp. output) variables. We extend this type of mode declarations with placemarker terms of the form -`#type`, which are treated as # when defining $L(M)$ but differ in the creation of the bottom clauses, see below.

The bottom clause is the clause with the longest true body in $L(M)$. The bottom clause is built with a method called *saturation*: an example e is randomly selected and the set of atoms $Body$ that are true regarding the example e is built incrementally. by considering the constants in e and querying the background for true atoms regarding these constants. A list of constants is kept and it is enlarged with those in `-type` placemarkers in the answers to the queries. The recall indicates how many answers to the queries must be considered. Besides an integer, it may be the symbol `*`, indicating all answers.

The procedure is iterated a user-defined number of times. Then a bottom clause is obtained from the clause $e \leftarrow Body$ by replacing ground terms with variables respecting the mode declarations. Placemarkers -`#type` are treated as `#type` when variabilizing because they are not replaced by variables but as `-type` placemarkers when building B, because terms in those positions are added to the current list of constants.

Function SATURATION, shown in Algorithm 6, builds a bottom clause for an example $Head$, where NS is a user-defined number of saturation steps to be performed.

In SLIPCOVER, a beam is a set of tuples $((Cl, Literals), LL)$ with Cl a clause, $Literals$ the set of literals admissible in the body of Cl and LL the log-likelihood of Cl. Function INITIALBEAM, shown in Algorithm 5, returns returns an initial beam containing tuples $((h : 0.5 :- true, Literals), -\infty)$ for each bottom clause $h : 0.5 :- Literals$. The likelihood is initialized to $-\infty$.

Then LIFTCOVER runs a beam search in the space of clauses for the target predicate.

In each beam search iteration, the first clause of the beam is removed and all its refinements are computed. Each refinement Cl' is scored by performing parameter learning with $T = \{Cl'\}$ and using the resulting LL as the heuristic. The scored refinements are inserted back into the beam in order of heuristic. If the beam exceeds a maximum user-defined size, the bottom elements are removed. Moreover, the refinements are added to a set of clauses CC.

Algorithm 6 Function SATURATION

```
1: function SATURATION(Head, r, NS)
2:    InTerms = ∅,
3:    BC = ∅                                              ▷ BC: bottom clause
4:    for all arguments t of Head do
5:       if t corresponds to a +type then
6:          add t to InTerms
7:       end if
8:    end for
9:    Let BC's head be Head
10:   repeat
11:      Steps ← 1
12:      for all modeb declarations modeb(r, s) do
13:         for all possible subs. σ of variables corresponding to +type in schema(s) by terms from InTerms do
14:            for j = 1 → r do
15:               if goal b = schema(s) succeeds with answer substitution σ′ then
16:                  for all v/t ∈ σ and σ′ do
17:                     if v corresponds to a −type or −#type then
18:                        add t to the set InTerms if not already present
19:                     end if
20:                  end for
21:                  Add b to BC's body
22:               end if
23:            end for
24:         end for
25:      end for
26:      Steps ← Steps + 1
27:   until Steps > NS
28:   Replace constants with variables in BC, using the same variable for equal terms
29:   return BC
30: end function
```

Algorithm 7 Function CLAUSEREFINEMENTS

```
1: function CLAUSEREFINEMENTS((Cl, Literals), NV)
2:    Refs = ∅, Nvar = 0;                    ▷ Nvar:number of different variables in a clause
3:    for all b ∈ Literals do
4:       Literals′ ← Literals \ {b}
5:       Add b to Cl body obtaining Cl′
6:       Nvar ← number of Cl′ variables
7:       if Cl′ is connected ∧ Nvar < NV then
8:          Refs ← Refs ∪ {(Cl′, Literals′)}
9:       end if
10:   end for
11:   return Refs
12: end function
```

For each clause Cl with $Literals$ admissible in the body, Function CLAUSEREFINEMENTS, shown in Algorithm 7, computes refinements by adding a literal from $Literals$ to the body. Furthermore, the refinements must respect the input-output modes of the bias declarations, must be connected (i.e., each body literal must share a variable with the head or a previous body literal) and their number of variables must not exceed a user-defined number NV. Refinements are of the form (Cl', L') where Cl' is the refined clause Cl' and L' is the new set of literals allowed in the body of Cl'.

Beam search is iterated a user-defined number of times or until the beam becomes empty. The output of this search phase is represented by the set CC of clauses. Then parameter learning is applied to the whole set CC, i.e., $T = CC$. Finally clauses with a weight smaller than *WMin* are removed.

The separate search for clauses has similarity with the covering loop of ILP systems such as Aleph (Srinivasan 2007) and Progol (Muggleton 1995). Differently from the ILP

case, however, the positive examples covered are not removed from the training set because coverage is probabilistic, so an example that is assigned nonzero probability by a clause may have its probability increased by further clauses. A selection of clauses is performed by parameter learning: clauses with very small weights are removed.

6 Related work

We first consider the work related to liftable PLP from the field of lifted inference and then that from the field of probabilistic rule learning.

Lifted inference for PLP under the distribution semantics is surveyed by Riguzzi et al. (2017a) that discuss three approaches.

LP^2 (Bellodi et al. 2014) uses an algorithm that extends Generalized Counting First Order Variable Elimination (GC-FOVE) (Taghipour et al. 2013) for taking into account clauses that have variables in bodies not appearing in the head (existentially quantified variables). Weighted First Order Model Counting (WFOMC) (Van den Broeck et al. 2014) uses a Skolemization algorithm that eliminates existential quantifiers from a theory without changing its weighted model count. Kisynski and Poole (2009) proposed an approach based on *Aggregation Parfactors* that can represent noisy-OR models. The three approaches have been compared experimentally in Riguzzi et al. (2017a) for general PLP and WFOMC was found the fastest.

Relational Logistic Regression (Kazemi et al. 2014) is a generalization of logistic regression that can also be applied to PLP.

LP^2, Aggregation Parfactors and Relational Logistic Regression reduce to the same algorithm for performing inference when the language is restricted to liftable PLP. LP^2 is based on GC-FOVE that in turn is an extension of Variable Elimination (VE) (Zhang and Poole 1994, 1996). VE was designed from the start to be able to exploit *causal independence*, the situation where multiple causes contribute independently to a common effect. Noisy-OR is a prominent example of causal independence. The capacity of VE to deal with noisy-OR is exploited in LP^2 to aggregate the contributions of multiple ground clause to the probability of the same atom in a lifted way., i.e., without generating the groundings.

We now discuss Aggregation Parfactors and Relational Logistic Regression. We first introduce *parametrized random variables* (PRV) that are represented by logical atoms. Each logical variable in a PRV is typed with a population. A *parfactor* is a triple $\langle \mathcal{C}, \boldsymbol{V}, F \rangle$ where $\mathcal{C}$ is a set of inequality constraints on parameters (logical variables), $\boldsymbol{V}$ is a set of PRV and F is a factor that is a function from the Cartesian product of ranges of PRVs in $\boldsymbol{V}$ to real values.

Aggregation parfactors (Kisynski and Poole 2009) can represent different kind of causal independence models, of which noisy-OR and noisy-MAX are special cases. Aggregation parfactors are a generalization of parfactors that are defined over two of PRVs one of which contains one more logical variable that the other. Therefore, the contributions of the PRV with the extra logical variable have to be aggregated and this is done by converting the aggregation parfactor into two regular parfactors. We can correctly encode liftable PLP with aggregation parfactors obtaining the same formula for calculating the probability of queries.

Relational Logistic Regression (Kazemi et al. 2014) generalizes logistic regression, where the probability of a child Boolean random variable Q is modeled on the basis of the values of parent random variables $\{X_1, \ldots, X_n\}$ as

$$P(q|X_1,\ldots,X_n) = \text{sigmoid}(w_0 + \sum_i w_i X_i)$$

where $q \equiv (Q = true)$ and $\text{sigmoid}(x) = 1/(1+e^{-x})$. For the case of Boolean variables, we can assume that the values are encoded with 0 for false and 1 for true.

To apply logistic regression to the relational case, the authors introduce the notion of *weighted parent formula* (WPF) for a PRV $Q(X)$, where X is a set of logical variables: a WPF is a triple $\langle L, F, w_i\rangle$ where L is a set of logical variables for which $L \cap X = \emptyset$, F is a Boolean formula of parent PRVs of $Q(X)$ such that each logical variable in F is either X or in L, and w_i is a weight.

Suppose $R_i(X_i)$ are the parents of PRV $Q(X)$, where X_i is the set of logical variables in R_i. A relational logistic regression (RLR) for Q with parents $R_i(X_i)$ is defined using a set of WPFs as:

$$P(Q(X)|\Pi) = \text{sigmoid}\left(\sum_{\langle L,F,w_i\rangle} w_i \sum_L F_{\Pi,X\to x}\right)$$

where Π represents the assigned values to parents of Q, x represents an assignment of an individual to each logical variable in X, and $F_{\Pi,X\to x}$ is formula F with each logical variable X in it being replaced according to x, and evaluated in Π. So RLR performs an aggregation of the parents of a PRV. The authors show that RLR can model noisy-OR therefore they can encode liftable PLP.

Lifted learning is still an open problem. An approach for performing lifted generative learning was proposed in Haaren et al. (2016): while the paper discusses both weight and structure learning, it focuses on Markov Logic Networks and generative learning, so it is not directly applicable to the setting considered in this paper.

Liftable PLP is very much related to Koller and Pfeffer (1997), Kersting and De Raedt (2002), Natarajan et al. (2009) where the contributions of different rules and different rule groundings are combined with noisy-OR combining rules. First-Order Probabilistic Logic (FOPL) (Koller and Pfeffer 1997) and Bayesian Logic Programs (BLP) (Kersting and De Raedt 2002) consider ground atoms as random variables and admit rules with a single atom in the head and only positive literals in the body. The meaning of such rules is that, for each grounding, the head atom random variable directly depends from the body atoms random variables. Thus rules are simply templates that can be used to generate a Bayesian network by a Knowledge-Based Model Construction (KBMC) approach (Wellman et al. 1992). Ground rules determine the families of the network and the random variables may have non-Boolean domains. The rules are also associated with parameters that define the CPT of the head variable given the body variables. In the case where an atom h appears in the head of more than one ground rule, the Bayesian network contains an extra family where the child is the variable for atom h and there is a parent h' for each rule whose family and CPT is defined by the rule. This extra family encodes a *combining rule*, i.e., a way of combining the contributions of the different rules for the same atom. Both FOPL and BLP allow different combining rules, including a noisy-OR combining rule where the CPT of the extra family encodes a disjunction, so liftable PLP models can be encoded directly in both FOPL and BLP. Differently from Liftable PLP, FOPL and BLP allow multiple layers of rules. Koller and Pfeffer (1997) and Kersting and De Raedt (2002) also present learning algorithms: the first discusses an EM algorithm for parameter learning and the latter EM and gradient descent parameter learning algorithms together with a structure learning algorithm. The learning problems are similar to the ones considered in this paper. Additionally, Kersting and De Raedt (2002) consider the

case where non-target atoms may be unobserved in the data. Both articles derive formulas for updating the parameters but, given the generality of the settings considered (non-Boolean domains, multiple layers of rules, different combining rules, incompleteness of the data), the formulas involve quantities to be computed by inference in the Bayesian network, while our formula depend on the parameters only.

SCOOBY, the structure learning algorithm of Kersting and De Raedt (2002), is similar to LIFTCOVER in the sense that it performs a greedy search in the space of programs evaluating each hypothesis by performing parameter learning. SCOOBY performs a local search by applying theory revisions to an initial hypothesis, while LIFTCOVER performs a beam search in the space of clauses followed by search in the space of theories by parameter learning.

The First-Order Conditional Influence Language (FOCIL) (Natarajan et al. 2009), like FOPL and BLP, considers probabilistic rules compactly encoding probabilistic dependencies. FOCIL is more similar to liftable PLP because it allows only one layer of rules. FOCIL uses different combining rules with respect to liftable PLP: the contributions of different groundings of the same rule with the same random variable in the head are combined by taking the mean and the contributions of different rules are combined either with a weighted mean or with a noisy-OR combining rule. Liftable PLP instead uses noisy-OR for both types of contributions. Natarajan et al. (2009) also present parameter learning algorithms for optimizing the mean squared errors or the log likelihood using gradient descent or EM both for weighted mean and with noisy-OR. While the derivation of the update formulas for the weights are similar, they differ because we don't use the mean combining function.

7 Experiments

LIFTCOVER[1] has been implemented in SWI-Prolog (Wielemaker et al. 2012) using a porting[2] of YAP-LBFGS,[3] a foreign language interface to libLBFGS.[4]

LIFTCOVER has been tested on the following 12 real world datasets: the 4 classic benchmarks UW-CSE (Kok and Domingos 2005), Mutagenesis (Srinivasan et al. 1996), Carcinogenesis (Srinivasan et al. 1997), Mondial (Schulte and Khosravi 2012); the 4 datasets Bupa, Nba, Pyrimidine, Triazine from https://relational.fit.cvut.cz/, and the 4 datasets Financial, Sisya, Sisyb and Yeast from Struyf et al. (2006).[5]

Statistics on all the domains are reported in Table 1. In all datasets the mega-examples are defined only by facts, there are no background non-probabilistic rules.

We would like to test the hypothesis that LIFTCOVER allows fast learning without a significant degradation of the quality of the solution with respect to SLIPCOVER: In order to compare the two systems fairly, in all datasets the language bias for SLIPCOVER allows only one atom in the head and only input predicates in the body, so the space of allowed clauses is the same for the two algorithms.

To evaluate the performance, we drew Precision-Recall curves and Receiver Operating Characteristics curves, computing the Area Under the Curve (AUC-PR and AUC-ROC

[1] The code of the systems and the datasets are available at https://bitbucket.org/machinelearningunife/liftcover.

[2] https://github.com/friguzzi/lbfgs.

[3] Developed by Bernd Gutmann.

[4] http://www.chokkan.org/software/liblbfgs/.

[5] https://dtai.cs.kuleuven.be/ACE/doc/.

Table 1 Characteristics of the datasets for the experiments: number of predicates (P), of tuples (T) (i.e., ground atoms), of positive (PEx) and negative (NEx) examples for target predicate(s), of folds (F)

Dataset	P	T	PEx	NEx	F
Financial	9	92, 658	34	223	10
Bupa	12	2781	145	200	5
Mondial	11	10, 985	572	616	5
Mutagen.	20	15, 249	125	126	10
Sisyb	9	354, 507	3705	9229	10
Sisya	9	358, 839	10, 723	6544	10
Pyrimidine	29	2037	20	20	4
Yeast	12	53, 988	1299	5456	10
Nba	4	1218	15	15	5
Triazine	62	10, 079	20	20	4
UW-CSE	15	2673	113	20, 680	5
Carcinogen.	36	24, 533	182	155	1

The number of tuples includes the target positive examples

respectively) with the methods reported in Davis and Goadrich (2006) and Fawcett (2006). AUC was used to measure the quality of the learned models as classifiers for predicting the truth values of atoms for target predicates, larger areas means better classifiers.

SLIPCOVER was compared with Aleph (Srinivasan 2007), SLIPCASE (Bellodi and Riguzzi 2012), SEM-CP-logic (Meert et al. 2008) LSM (Kok and Domingos 2010), ALEPH++ExactL1 (Huynh and Mooney 2008), LEMUR (Mauro et al. 2015), BUSL (Mihalkova and Mooney 2007), MLN-BC, MLN-BT (Khot et al. 2011) RDN-B (Natarajan et al. 2012), SLS (Hoos and Stützle 2004) and RRR (Železný et al. 2002, 2006) in Bellodi and Riguzzi (2015) and Mauro et al. (2015) on 8 datasets. In almost all datasets SLIPCOVER was among the top 4 systems in terms of AUC-PR, thus showing that it is among the state of the art of StarAI. Thus comparing LIFTCOVER with SLIPCOVER will also provide an evaluation of its performance in the general context of StarAI.

LIFTCOVER-EM was run with the following parameters for EM: $restarts = 1$, $max\_iter = 10$, $\epsilon = 10^{-4}$ and $\delta = 10^{-5}$. The default parameters have been used for libLBFGS. The parameters controlling structure learning are: the number *NInt* of mega-examples on which to build the bottom clauses, the number *NA* of bottom clauses to be built for each mega-example, the number *NS* of saturation steps (for building the bottom clauses), the maximum number *NI* of clause search iterations, the size *NB* of the beam, the maximum number *NV* of variables in a rule, the threshold for the rule parameter *WMin* under which the rule is removed and the maximum numbers *NIS* of iterations of structure search of SLIPCOVER. Table 2 shows the values we have used. *WMin* was set to 0 in all dataset in order to perform the simplest pruning type, that of rules that don't influence at all the prediction. The other parameters for UW-CSE, Mutagenesis, Carcinogenesis, Mondial have been set as in Mauro et al. (2015). For the other datasets they have been set by a random search with the objective of keeping the computation time of both algorithms within a few hundred seconds.

All experiments were performed on GNU/Linux machines with an Intel Xeon Haswell E5-2630 v3 (2.40GHz) CPU with 8GB of memory allocated to the job.

Figures 2, 3 and 4 show histograms of the average over the folds of AUC-ROC, AUC-PR and the execution time respectively of LIFTCOVER-EM, LIFTCOVER-LBFGS and SLIPCOVER. Tables 3, 4 and 5 show the same data in a tabular way.

Table 2 Parameters controlling structure search for LIFTCOVER and SLIPCOVER

Dataset	*NB*	*NI*	*NInt*	*NS*	*NA*	*NV*	*WMin*	*NIS*
Financial	100	20	1	1	1	4	0	50
Bupa	100	20	1	1	1	4	0	50
Mondial	1000	10	1	2	6	5	0	10, 000
Mutagen.	100	10	1	1	1	4	0	500
Sisb	100	20	1	1	1	50	0	40
Sisya	100	20	1	1	1	4	0	40
Pyrimidine	100	20	1	1	1	100	0	50
Yeast	100	20	1	1	1	4	0	50
Nba	100	20	1	1	1	100	0	50
Triazine	100	20	1	1	1	4	0	50
UW-CSE	20	60	1	1	1	4	0	500
Carcinogen.	100	60	1	2	1	3	0	50

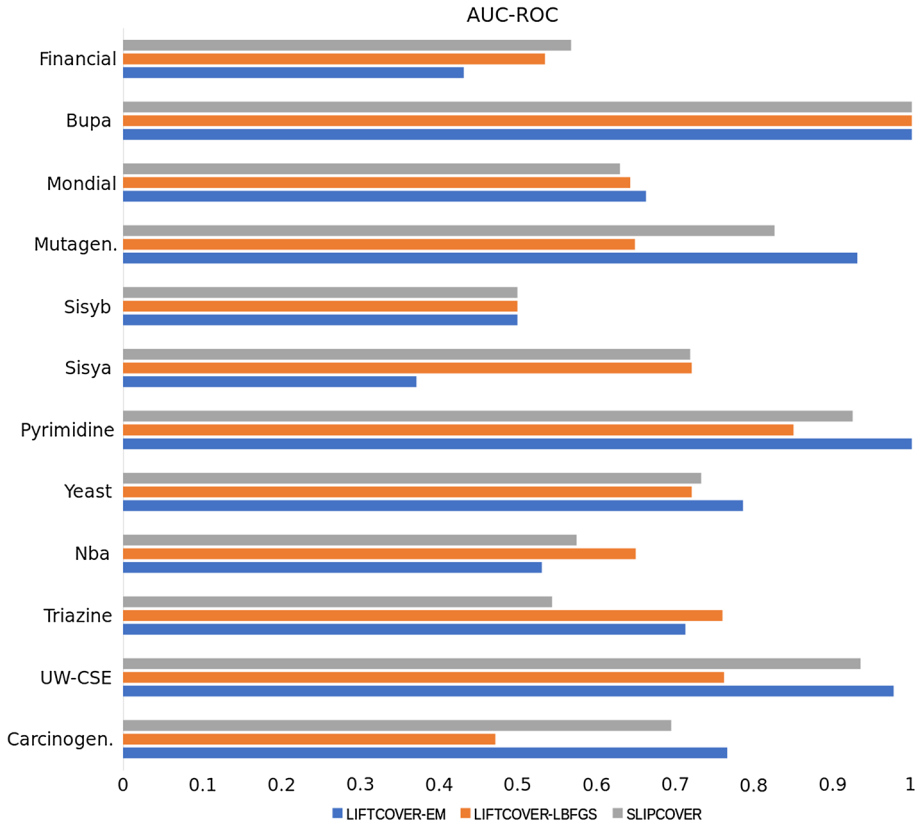

Fig. 2 Histograms of average AUC-ROC

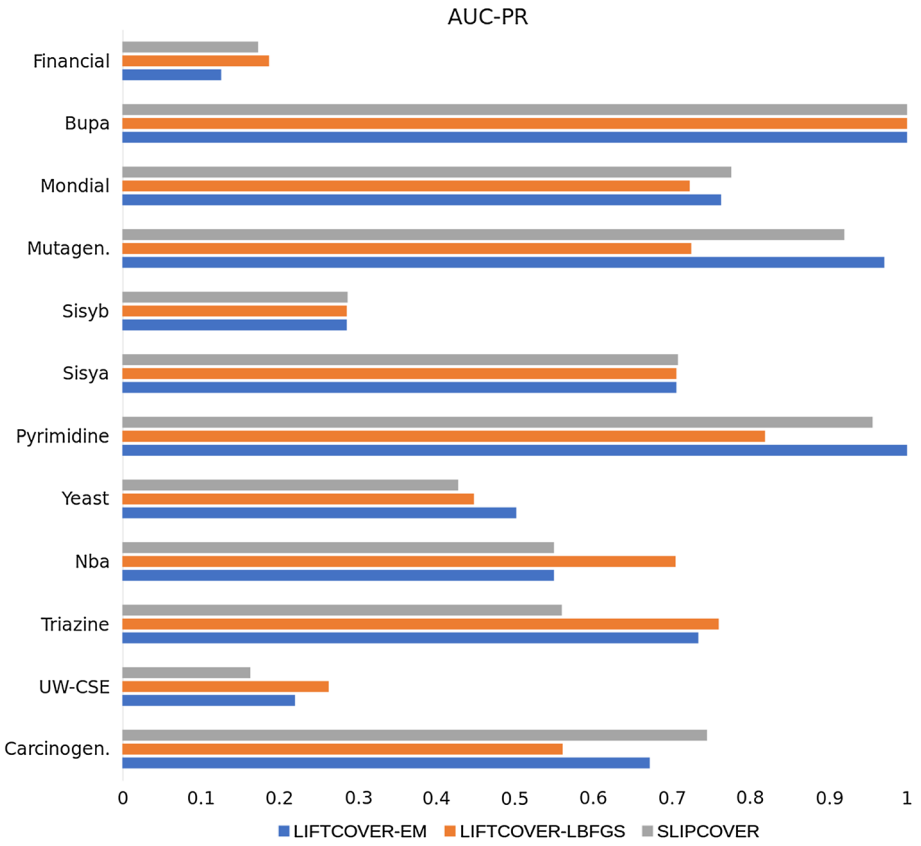

Fig. 3 Histograms of average AUC-PR

LIFTCOVER-EM beats SLIPCOVER 8 times and ties twice in terms of AUC-ROC and beats SLIPCOVER 5 times and ties twice in terms of AUC-PR, with two cases (Sisya and Sisyb) where it is nearly as good. In terms of execution time, LIFTCOVER-EM is faster than SLIPCOVER in 9 cases and in the other three cases is nearly as fast. In 7 cases the gap is of one or more orders of magnitude (UW-CSE, Carcinognesis, Nba, Triazine, Sisya, Sisyb, Yeast).

LIFTCOVER-LBFGS beats SLIPCOVER 4 times (in Sisya they are very close) and ties twice in terms of AUC-ROC and beats SLIPCOVER 5 times and ties once in terms of AUC-PR, with two cases where it is nearly as good. In terms of execution time, LIFTCOVER-LBFGS beats SLIPCOVER 9 times, with one case where SLIPCOVER is nearly as fast. In 5 cases the gap is of one or more orders of magnitude (UW- CSE, Carcinogenesis, Nba, Sisya, Sisyb). In Mutagenesis and Pyrimidine LIFTCOVER-LBFGS takes about ten times and twice as much as SLIPCOVER respectively. The reason for these differences may be due to the fact that these are small-medium datasets which are relatively easy for the systems (and also for ILP systems in general (Reutemann et al. 2004)), so SLIPCOVER is able to achieve good performance even with the allowed small search space.

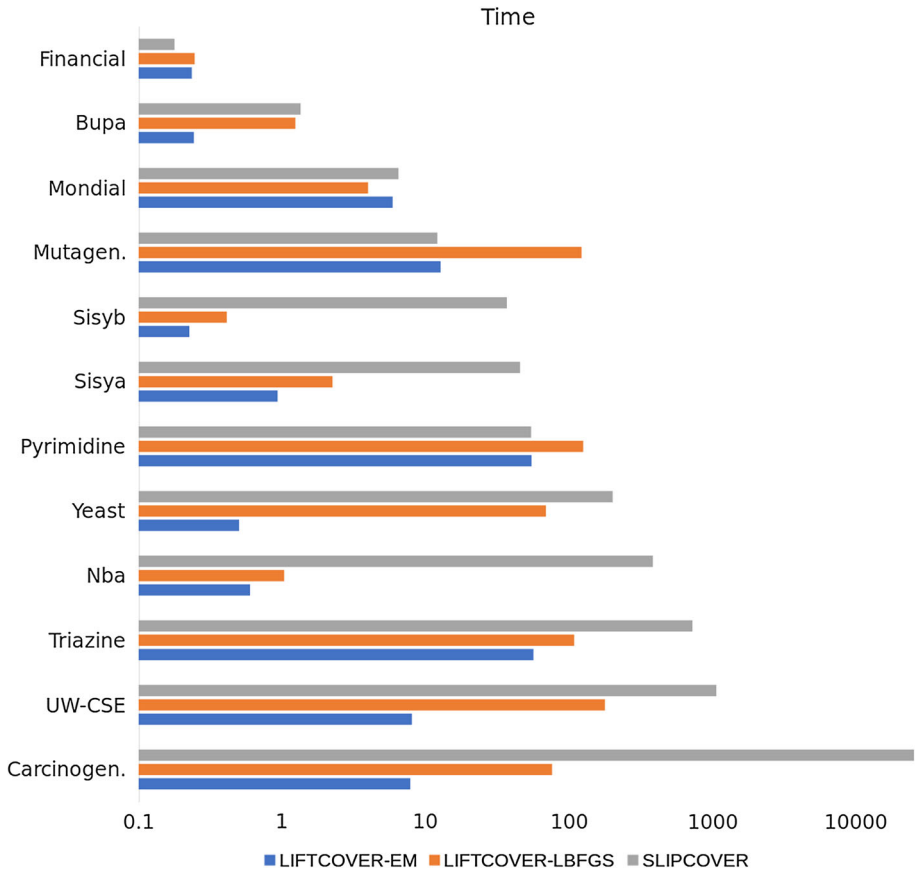

Fig. 4 Histograms of average time in seconds. The scale of the Y axis is logarithmic

Table 3 Average AUC-ROC

AUC-ROC	LIFTCOVER-EM	LIFTCOVER-LBFGS	SLIPCOVER
Financial	0.432	0.535	0.568
Bupa	1.000	1.000	1.000
Mondial	0.663	0.643	0.630
Mutagen.	0.931	0.649	0.826
Sisyb	0.500	0.500	0.500
Sisya	0.372	0.721	0.719
Pyrimidine	1.000	0.850	0.925
Yeast	0.786	0.721	0.733
Nba	0.531	0.650	0.575
Triazine	0.713	0.760	0.544
UW-CSE	0.977	0.762	0.935
Carcinogen.	0.766	0.472	0.695

Table 4 Average AUC-PR

AUC-PR	LIFTCOVER-EM	LIFTCOVER-LBFGS	SLIPCOVER
Financial	0.126	0.187	0.173
Bupa	1.000	1.000	1.000
Mondial	0.763	0.723	0.776
Mutagen.	0.971	0.725	0.920
Sisyb	0.286	0.286	0.287
Sisya	0.706	0.706	0.708
Pyrimidine	1.000	0.819	0.956
Yeast	0.502	0.448	0.428
Nba	0.550	0.705	0.550
Triazine	0.734	0.760	0.560
UW-CSE	0.220	0.263	0.163
Carcinogen.	0.672	0.561	0.745

Table 5 Average time in seconds

Time	LIFTCOVER-EM	LIFTCOVER-LBFGS	SLIPCOVER
Financial	0.235	0.246	0.178
Bupa	0.243	1.239	1.349
Mondial	5.911	3.984	6.490
Mutagen.	12.77	122.8	12.11
Sisyb	0.226	0.412	37.00
Sisya	0.932	2.252	45.75
Pyrimidine	54.99	126.1	54.62
Yeast	0.502	69.30	202.4
Nba	0.599	1.036	386.0
Triazine	56.69	109.1	728.2
UW-CSE	8.054	178.6	1069
Carcinogen.	7.850	76.49	25,568

Overall we see that both lifted algorithms are usually faster, sometimes by a large margin, with respect to SLIPCOVER, especially LIFTCOVER-EM. Moreover, this system often finds better quality solutions, showing that structure search by parameter learning is effective.

Between the two lifted algorithms, the EM version wins 6 times and ties twice with respect to AUC-ROC and wins 5 times and ties 3 times with respect to AUC-PR. In terms of execution times, the EM version wins on all dataset except Mondial, with differences below one order of magnitude except UW-CSE and Yeast. So LIFTCOVER-EM appears to be superior both in terms of solution quality and of computation time. EM seems to be better at escaping local maxima and cheaper than LBFGS, possibly also due to the fact that LBFGS may require a careful tuning of parameters and that it is implemented as a foreign language library.

Therefore, LIFTCOVER represents a valid alternative to SLIPCOVER when learning the definition of a single predicate by using a single layer of rules with a single head.

While the problem of inducing general probabilistic logic program will come to fore soon, learning a restricted language may be a valid alternative in many cases. For example, in Bellodi and Riguzzi (2015) and Mauro et al. (2015) SLIPCOVER was applied to the UW-CSE dataset with a language bias that allowed multiple heads and clauses for non-target predicates. The values of AUCPR obtained there are 0.13 and 0.11 respectively, that are lower than the values obtained by LIFTCOVER and SLIPCOVER shown in Table 4. Therefore restricting the language bias in this case actually improved the performance, probably because it has a regularizing effect.

Moreover, by looking at the characteristics of the datasets, there doesn't seem to exist a clear relationship between the number of predicates/number of tuples and the performance: complex datasets (large number of predicates) may be hard for LIFTCOVER (Carcinogenesis) or easy (Triazine) and large datasets (large number of tuples) may be hard for LIFTCOVER (Financial) or easy (Yeast). We leave for future work further experiments to investigate whether the adoption of a more expressive language can improve the performance for the datasets that are hard for LIFTCOVER such as Sisyb and Financial.

8 Conclusions

We have presented an algorithm that learns the structure of a restricted version of probabilistic logic programs using either EM or LBFGS for parameter learning. The results show that LIFTCOVER-EM and LIFTCOVER-LBFGS are faster than SLIPCOVER and often more accurate, with LIFTCOVER-EM performing slightly better than LIFTCOVER-LBFGS.

Regarding the restriction imposed on the language, the results of SLIPCOVER on the UW-CSE dataset with a more expressive language are inferior to those of the restricted language. In the future we plan to perform further experiments to investigate the impact of the adoption of a more expressive language on the datasets that are hard for LIFTCOVER. Moreover, we plan to compare LIFTCOVER directly with other algorithms designed for scalability such as Nath and Domingos (2015), Huynh and Mooney (2011).

We also plan to test the influence of settings in LBFGS and to add explicit regularization to parameter learning in order to reduce overfitting.

Acknowledgements This work was supported by the "National Group of Computing Science (GNCS-INDAM)" and by Regione Emilia Romagna under the Piano triennale alte competenze—POR FSE 2014/2020 Obiettivo tematico 10.

References

Alberti, M., Bellodi, E., Cota, G., Riguzzi, F., & Zese, R. (2017). `cplint` on SWISH: Probabilistic logical inference with a web browser. *Intelligenza Artificiale*, *11*(1), 47–64. https://doi.org/10.3233/IA-170105.

Bellodi, E., & Riguzzi, F. (2012). Learning the structure of probabilistic logic programs. In S. Muggleton, A. Tamaddoni-Nezhad, & F. Lisi (Eds.) *22nd international conference on inductive logic programming*, Vol. 7207, LNCS. Berlin: Springer, pp 61–75.

Bellodi, E., & Riguzzi, F. (2013). Expectation maximization over binary decision diagrams for probabilistic logic programs. *Intelligent Data Analysis*, *17*(2), 343–363.

Bellodi, E., & Riguzzi, F. (2015). Structure learning of probabilistic logic programs by searching the clause space. *Theory and Practice of Logic Programming*, *15*(2), 169–212. https://doi.org/10.1017/S1471068413000689.

Bellodi, E., Lamma, E., Riguzzi, F., Costa, V. S., & Zese, R. (2014). Lifted variable elimination for probabilistic logic programming. *Theory and Practice of Logic Programming*, *14*(4–5), 681–695. https://doi.org/10.1017/S1471068414000283.

Darwiche, A., & Marquis, P. (2002). A knowledge compilation map. *Journal of Artificial Intelligence Research*, *17*, 229–264.

Davis, J., & Goadrich, M. (2006). The relationship between precision-recall and ROC curves. In *European conference on machine learning* (ECML 2006), ACM, pp. 233–240.

De Raedt, L., & Kimmig, A. (2015). Probabilistic (logic) programming concepts. *Machine Learning*, *100*(1), 5–47.

De Raedt, L., Kimmig, A., & Toivonen, H. (2007). ProbLog: A probabilistic prolog and its application in link discovery. In M.M. Veloso (Ed.), *20th international joint conference on artificial intelligence* (IJCAI 2007), Vol. 7. AAAI Press/IJCAI, pp 2462–2467.

Dempster, A. P., Laird, N. M., & Rubin, D. B. (1977). Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society Series B (Methodological)*, *39*(1), 1–38.

Di Mauro, N., Bellodi, E., & Riguzzi, F. (2015). Bandit-based Monte-Carlo structure learning of probabilistic logic programs. *Machine Learning*, *100*(1), 127–156. https://doi.org/10.1007/s10994-015-5510-3.

Fawcett, T. (2006). An introduction to ROC analysis. *Pattern Recognition Letters*, *27*, 861–874.

Good, I. J. (1961). A causal calculus (I). *The British journal for the philosophy of science*, *11*(44), 305–318.

Gordon, D. M. (1998). A survey of fast exponentiation methods. *Journal of Algorithms*, *27*(1), 129–146. https://doi.org/10.1006/jagm.1997.0913.

Gorlin, A., Ramakrishnan, C. R., & Smolka, S. A. (2012). Model checking with probabilistic tabled logic programming. *Theory and Practice of Logic Programming*, *12*(4–5), 681–700.

Hoos, H. H., & Stützle, T. (2004). *Stochastic local search: Foundations and applications*. New York: Elsevier/Morgan Kaufmann.

Huynh, T. N., & Mooney, R. J. (2008). Discriminative structure and parameter learning for Markov logic networks. In W.W. Cohen, A. McCallum, & S.T. Roweis (Eds.), *Proceedings of the 25th international conference on machine learning*, ACM, pp. 416–423.

Huynh, T. N., & Mooney, R. J. (2011). Online structure learning for Markov logic networks. In D. Gunopulos, T. Hofmann, D. Malerba, & M. Vazirgiannis (Eds.), *European conference on machine learning and principles and practice of knowledge discovery in databases* (ECMLPKDD 2011), Vol. 6912. Lecture Notes in Computer Science. Springer, pp. 81–96. https://doi.org/10.1007/978-3-642-23783-6_6.

Kazemi, S. M., Buchman, D., Kersting, K., Natarajan, S., & Poole, D. (2014). Relational logistic regression. In C. Baral, G. D. Giacomo, & T. Eiter (Eds.), *14th international conference on principles of knowledge representation and reasoning* (KR 2014). AAAI Press.

Kersting, K., & De Raedt, L. (2002). Basic principles of learning Bayesian logic programs. In Institute for Computer Science, University of Freiburg, Citeseer.

Khot, T., Natarajan, S., Kersting, K., & Shavlik, J. W. (2011). Learning Markov logic networks via functional gradient boosting. In *Proceedings of the 11th IEEE international conference on data mining*, IEEE, pp. 320–329.

Kietz, J., & Lübbe, M. (1994). An efficient subsumption algorithm for inductive logic programming. In W.W. Cohen, & H. Hirsh (Eds.), *11th international conference on machine learning*, Morgan Kaufmann, pp. 130–138.

Kisynski, J., & Poole, D. (2009). Lifted aggregation in directed first-order probabilistic models. In C. Boutilier (Ed.), *21st international joint conference on artificial intelligence* (IJCAI 2009), pp. 1922–1929.

Kok, S., & Domingos, P. (2005). Learning the structure of Markov logic networks. In *22nd international conference on machine learning*, ACM, pp. 441–448.

Kok, S., & Domingos, P. (2010). Learning Markov logic networks using structural motifs. In J. Fürnkranz, & T. Joachims (Eds.), *27th international conference on machine learning*, Omnipress, pp. 551–558.

Koller, D., & Friedman, N. (2009). *Probabilistic graphical models: Principles and techniques*. Cambridge, MA: MIT Press.

Koller, D., & Pfeffer, A. (1997). Learning probabilities for noisy first-order rules. In *IJCAI*, pp. 1316–1323.

Meert, W., Struyf, J., & Blockeel, H. (2008). Learning ground CP-Logic theories by leveraging Bayesian network learning techniques. *Fundamenta Informaticae*, *89*(1), 131–160.

Meert, W., Struyf, J., & Blockeel, H. (2010). CP-Logic theory inference with contextual variable elimination and comparison to BDD based inference methods. In L. De Raedt (Ed.), *Inductive logic programming, 19th international conference* (ILP 2009), Vol. 5989, Lecture notes in computer science. Springer, pp. 96–109. https://doi.org/10.1007/978-3-642-13840-9_10.

Mihalkova, L., & Mooney, R. J. (2007). Bottom-up learning of Markov logic network structure. In *Proceedings of the 24th international conference on machine learning*, ACM, pp. 625–632.

Mørk, S., & Holmes, I. (2012). Evaluating bacterial gene-finding hmm structures as probabilistic logic programs. *Bioinformatics*, *28*(5), 636–642.

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

Natarajan, S., Tadepalli, P., Kunapuli, G., & Shavlik, J. (2009). Learning parameters for relational probabilistic models with noisy-or combining rule. In *International conference on machine learning and applications*, 2009. ICMLA'09. IEEE, pp. 141–146.

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

Nath, A., & Domingos, P. (2015). Learning relational sum-product networks. In B. Bonet & S. Koenig (Eds.), *29th national conference on artificial intelligence, AAAI'15* (pp. 2878–2886). Austin, Texas, USA: AAAI Press.

Nguembang Fadja, A., & Riguzzi, F. (2017). Probabilistic logic programming in action. In A. Holzinger, R. Goebel, M. Ferri, & V. Palade (Eds.), *Towards integrative machine learning and knowledge extraction* (Vol. 10344)., Lecture notes in computer science Berlin: Springer. https://doi.org/10.1007/978-3-319-69775-8_5.

Nishino, M., Yamamoto, A., & Nagata, M. (2014). A sparse parameter learning method for probabilistic logic programs. In *Statistical relational artificial intelligence*, Vol. WS-14-13. Papers from the 2014 AAAI workshop, AAAI Press, AAAI Workshops.

Nocedal, J. (1980). Updating quasi-Newton matrices with limited storage. *Mathematics of Computation*, *35*(151), 773–782.

Pearl, J. (1988). *Probabilistic reasoning in intelligent systems: Networks of plausible inference*. Burlington: Morgan Kaufmann.

Poole, D. (1997). The independent choice logic for modelling multiple agents under uncertainty. *Artificial Intelligence*, *94*, 7–56.

Poole, D. (2000). Abducing through negation as failure: Stable models within the independent choice logic. *Journal of Logic Programming*, *44*(1–3), 5–35.

Poole, D. (2003). First-order probabilistic inference. *IJCAI*, *3*, 985–991.

Quinlan, J. R. (1990). Learning logical definitions from relations. *Machine Learning*, *5*, 239–266. https://doi.org/10.1007/BF00117105.

Reutemann, P., Pfahringer, B., & Frank, E. (2004). A toolbox for learning from relational data with propositional and multi-instance learners. In G.I. Webb, & X. Yu (Eds.), *17th Australian joint conference on artificial intelligence* (AI 1994), Vol. 3339. Lecture notes in computer science, Springer, pp. 1017–1023. https://doi.org/10.1007/978-3-540-30549-1_95.

Riguzzi, F. (2014). Speeding up inference for probabilistic logic programs. *The Computer Journal*, *57*(3), 347–363. https://doi.org/10.1093/comjnl/bxt096.

Riguzzi, F. (2016). The distribution semantics for normal programs with function symbols. *International Journal of Approximate Reasoning*, *77*, 1–19. https://doi.org/10.1016/j.ijar.2016.05.005.

Riguzzi, F., & Swift, T. (2011). The PITA system: Tabling and answer subsumption for reasoning under uncertainty. *Theory and Practice of Logic Programming*, *11*(4–5), 433–449. https://doi.org/10.1017/S147106841100010X.

Riguzzi, F., & Swift, T. (2018). Probabilistic logic programming under the distribution semantics. In M. Kifer & Y. A. Liu (Eds.), *Declarative logic programming: Theory, systems, and applications*. Bonita Springs: Association for Computing Machinery and Morgan & Claypool.

Riguzzi, F., Bellodi, E., Lamma, E., Zese, R., & Cota, G. (2016). Probabilistic logic programming on the web. *Software: Practice and Experience*, *46*(10), 1381–1396. https://doi.org/10.1002/spe.2386.

Riguzzi, F., Bellodi, E., Zese, R., Cota, G., & Lamma, E. (2017a). A survey of lifted inference approaches for probabilistic logic programming under the distribution semantics. *International Journal of Approximate Reasoning*, *80*, 313–333. https://doi.org/10.1016/j.ijar.2016.10.002.

Riguzzi, F., Lamma, E., Alberti, M., Bellodi, E., Zese, R., & Cota, G. (2017b). Probabilistic logic programming for natural language processing. In F. Chesani, P. Mello, & M. Milano (Eds.), *Workshop on deep understanding and reasoning*, Vol. 1802, URANIA 2016, Sun SITE Central Europe, CEUR workshop proceedings, pp. 30–37.

Sato, T. (1995). A statistical learning method for logic programs with distribution semantics. In L. Sterling (Ed.), *12th international conference on logic programming* (ICLP 1995), MIT Press, pp. 715–729.

Sato, T., & Kameya, Y. (1997). PRISM: A language for symbolic-statistical modeling. In *15th international joint conference on artificial intelligence* (IJCAI 1997), Vol. 97, pp 1330–1339.

Sato, T., & Kubota, K. (2015). Viterbi training in PRISM. *Theory and Practice of Logic Programming*, *15*(02), 147–168.

Schulte, O., & Khosravi, H. (2012). Learning graphical models for relational data via lattice search. *Machine Learning*, *88*(3), 331–368.

Srinivasan, A. (2007). The aleph manual. http://www.cs.ox.ac.uk/activities/machlearn/Aleph/aleph.html. Accessed April 3, 2018.
Srinivasan, A., Muggleton, S., Sternberg, M. J. E., & King, R. D. (1996). Theories for mutagenicity: A study in first-order and feature-based induction. *Artificial Intelligence*, *85*(1–2), 277–299.
Srinivasan, A., King, R. D., Muggleton, S., & Sternberg, M. J. E. (1997). Carcinogenesis predictions using ILP. In N. Lavrac, & S. Dzeroski (Eds.), *7th international workshop on inductive logic programming*, Vol. 1297. Lecture notes in computer science, Berlin: Springer, pp 273–287.
Struyf, J., Davis, J., & Page, D. (2006). An efficient approximation to lookahead in relational learners. In *European conference on machine learning* (ECML 2006), Lecture notes in computer science. Springer, pp. 775–782. https://doi.org/10.1007/11871842_79.
Taghipour, N., Fierens, D., Davis, J., & Blockeel, H. (2013). Lifted variable elimination: Decoupling the operators from the constraint language. *Journal of Artificial Intelligence Research*, *47*, 393–439.
Valiant, L. G. (1979). The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, *8*(3), 410–421.
Van den Broeck, G., Meert, W., & Darwiche, A. (2014). Skolemization for weighted first-order model counting. In C. Baral, G.D. Giacomo, & T. Eiter (Eds.), *14th international conference on principles of knowledge representation and reasoning* (KR 2014), AAAI Press, pp. 111–120.
Van Gelder, A., Ross, K. A., & Schlipf, J. S. (1991). The well-founded semantics for general logic programs. *Journal of the ACM*, *38*(3), 620–650.
Van Haaren, J., Van den Broeck, G., Meert, W., & Davis, J. (2016). Lifted generative learning of markov logic networks. *Machine Learning*, *103*(1), 27–55. https://doi.org/10.1007/s10994-015-5532-x.
Vennekens, J., & Verbaeten, S. (2003). Logic programs with annotated disjunctions. Tech. Rep. CW386, KU Leuven.
Vennekens, J., Verbaeten, S., & Bruynooghe, M. (2004a). Logic programs with annotated disjunctions. In B. Demoen, & V. Lifschitz (Eds.), *24th international conference on logic programming* (ICLP 2004), Vol. 3131. Lecture notes in computer science, Berlin: Springer, pp. 195–209.
Vennekens, J., Verbaeten, S., & Bruynooghe, M. (2004b). Logic programs with annotated disjunctions. In *24th international conference on logic programming* (ICLP 2004), Vol. 3132. Lecture notes in computer science. Springer, pp. 431–445.
Wang, W. Y., Mazaitis, K., & Cohen, W. W. (2014). Structure learning via parameter learning. In J. Li, X.S. Wang, M.N. Garofalakis, I. Soboroff, T. Suel, & M. Wang (Eds.), *23rd ACM international conference on conference on information and knowledge management*, CIKM 2014, ACM Press, pp. 1199–1208. https://doi.org/10.1145/2661829.2662022.
Wellman, M. P., Breese, J. S., & Goldman, R. P. (1992). From knowledge bases to decision models. *The Knowledge Engineering Review*, *7*(1), 35–53.
Wielemaker, J., Schrijvers, T., Triska, M., & Lager, T. (2012). SWI-Prolog. *Theory and Practice of Logic Programming*, *12*(1–2), 67–96. https://doi.org/10.1017/S1471068411000494.
Železný, F., Srinivasan, A., & Page, C. D. (2002). Lattice-search runtime distributions may be heavy-tailed. In *Proceedings of the 12th international conference on inductive logic programming*. Springer.
Železný, F., Srinivasan, A., & Page, C. D, Jr. (2006). Randomised restarted search in ILP. *Machine Learning*, *64*(1–3), 183–208.
Zhang, N. L., & Poole, D. (1994). A simple approach to Bayesian network computations. In *10th Canadian conference on artificial intelligence*, Canadian AI 1994, pp. 171–178.
Zhang, N. L., & Poole, D. L. (1996). Exploiting causal independence in Bayesian network inference. *Journal of Artificial Intelligence Research*, *5*, 301–328.