Parameter Learning in ProbLog with Probabilistic Rules

Francisco H. O. V. de Faria[1] and Arthur C. Gusmão[1] and Glauber De Bona[1]
and Denis D. Mauá[2] and Fabio G. Cozman[1]

[1]Escola Politécnica and [2]Instituto de Matemática e Estatística — Universidade de São Paulo, Brazil

Abstract. Probabilistic logic programs under the distribution semantics offer a flexible language to specify deterministic rules and probabilistic assessments. State-of-art inference and learning algorithms are now available in the freely available ProbLog package. In this paper we describe techniques that speed up the learning algorithms in ProbLog, both for complete and incomplete datasets, by exploring a new approach for likelihood maximization. Our experiments show that our techniques significantly speed up the learning process.

Categories and Subject Descriptors: I.2.6 [**Artificial Intelligence**]: Learning

Keywords: parameter learning, probabilistic logic programming, ProbLog

1. INTRODUCTION

There is a myriad of ways to combine first-order logic and probability, thus allowing reasoning about relational structures while handling uncertainty. Examples are relational Bayesian networks [Jaeger 1997], Markov logic networks [Richardson and Domingos 2006] and a variety of probabilistic logics [Halpern 1990; Ognjanovic and Raškovic 2000]. A well-explored path is to endow logic programs with probabilities. In fact, many are the proposals to extend the logic programing framework of Prolog with probabilistic semantics [Getoor and Taskar 2007; De Raedt et al. 2008]. A particularly popular semantics for probabilistic logic programs is due to Sato, and usually referred to as the *distribution semantics*. The idea is that we have rules, as in logic programming, such as

$$\mathsf{cares}(X,Y) :- \mathsf{person}(X), \mathsf{person}(Y), \mathsf{neighbor}(X,Y). \tag{1}$$

and probabilistic facts such as $0.8 :: \mathsf{neighbor}(X,Y)$, meaning that the probability that any X and Y are neighbors is 0.8.

Usually, one is interested in assigning probabilities not only to facts, but also to rules. For instance, we may want to express that $\mathsf{person}(X)$ and $\mathsf{person}(Y)$ yields a proof for $\mathsf{cares}(X,Y)$ with probability 0.8. Then we could write

$$0.8 :: \mathsf{cares}(X,Y) :- \mathsf{person}(X), \mathsf{person}(Y). \tag{2}$$

The syntax we just used can be found in the ProbLog package, a freely available system that implements state-of-art algorithms for inference and learning in the context of probabilistic logic programming [Fierens et al. 2015]. ProbLog adopts Sato's distribution semantics, together with probabilistic facts and probabilistic rules. Inference in ProbLog is done by model counting, and parameter learning relies on the Expectation-Maximization algorithm (EM). One important feature

F. H. O. V. de Faria is supported by a scholarship from Toshiba Corporation. A. C. Gusmão is supported by a scholarship from CNPq. G. De Bona is supported by Fapesp, Grant 2016/25928-4. F. G. Cozman and D. D. Mauá are partially supported by CNPq.

of ProbLog's EM-like algorithm is that it requires introducing a latent variable for each probabilistic rule in the program of interest. This is a major source of inefficiency, that we fix in this paper.

In this work, we offer an alternative approach to learning parameters in probabilistic program with probabilistic rules. Instead of inserting unobservable variables, we exploit the intrinsic semantics of probabilistic rules to express the likelihood of the observations in function of the parameters, which is a main ingredient of parameter learning. The lesser number of latent variables speeds up the learning task, especially with complete data, when we can dispense with the EM algorithm altogether.

This article is structured as follows: Section 2 presents ProbLog's probabilistic programs; Section 3 brings an overview of how parameter learning is implemented in ProbLog; our approach to parameter learning with probabilistic rules is put forward in Section 4; in Section 5, we present the results of experiments comparing the performance of our algorithm to ProbLog's implementation.

2. PRELIMINARIES

We follow the syntax and semantics of probabilistic logic programs from the ProbLog framework [De Raedt et al. 2007; Fierens et al. 2015].

2.1 Syntax

Consider a vocabulary with logical variables $X, Y, \ldots$, predicate symbols $r, s, \ldots$ and constants $a, b, \ldots$. Each predicate symbol has an associated arity. An *atom* is an expression of the form $r(t_1, t_2, \ldots, t_m)$ where r is a predicate symbol with arity m, and each t_i is a *term*, which is either a constant or a logical variable. An atom is *ground* if it has no logical variables. An atomic proposition is an 0-arity predicate symbol, which is also a ground atom. A *grounding* is a function taking logical variables and returning constants. A *literal* is an atom (A) possibly preceded by **not** (**not** A). A *(deterministic) rule* is an expression of the form $H :- B_1, \ldots, B_n.$, where H is an atom, called the *head*, and each *subgoal* B_i is a literal, with $B_1, \ldots, B_n$ being the rule's *body*. A fact is a rule with empty body ($H :- .$). If an atom is in the head of some rule, it is said to be a *derived* atom. A set of rules is a *logic program*. The grounding of a rule is a *ground rule* obtained by applying the same grounding to each atom. The grounding of a program is the propositional program obtained by applying every possible grounding to each rule and fact, using only the constants in the program.

For a given logic program, the *dependency graph* contains its ground atoms as nodes and arcs $\langle A, B\rangle$ only if there is a ground rule with A in the body, possibly negated, and B in the head. A logic program is *acyclic* if its dependency graph is acyclic.

ProbLog programs are formed by standard logic programs together with *probabilistic facts*, which have the form $p :: F.$, where $p \in [0, 1]$ is a real number and F is an atom, called *probabilistic*. Similarly to rules, a probabilistic fact can be grounded to form a set of ground probabilistic facts. Formally, we define a *probabilistic program* as a pair $\langle \mathbf{P}, \mathbf{PF}\rangle$, where $\mathbf{P}$ is a logic program, $\mathbf{PF}$ is a set of probabilistic facts and probabilistic atoms are not derived in $\mathbf{P}$.

2.2 Semantics

The semantics of a relational probabilistic program is simply defined through the semantics of its grounding, so we focus on the propositional case. ProbLog's semantics for probabilistic programs is based on the standard semantics of Prolog. It is convenient to view a ground atom A in a program as random variables taking values in $\{0, 1\}$ (false and true), and we write $P \models A = 1$ ($P \models A = 0$) iff A ($\mathbf{A}$) is entailed by the logic program P.

Let $T = \langle \mathbf{P}, \mathbf{PF}\rangle$ be a probabilistic program, and let $\{p_i :: F_i. | 1 \leq i \leq n\}$ be the grounding of $\mathbf{PF}$, for $p_1, \ldots, p_n \in [0, 1]$ and a set of ground atoms $F = \{F_1, \ldots, F_n\}$. T implicitly defines a probability

distribution over the logic programs, $P_{\mathbf{PR}}(\mathbf{P} \cup F') = \prod_{F_i \in F'} p_i \prod_{F_i \in F \setminus F'} (1 - p_i)$, where F' is a subset of F. For the given $T = \langle \mathbf{P}, \mathbf{PF} \rangle$, we can define the probability P_T of a given set of ground atoms $Q = \{Q_1, \ldots, Q_n\}$ have the truth value $q = \langle q_1, \ldots, q_n \rangle \in \{0,1\}^n$ $(Q = q)$, by employing $P_{\mathbf{PR}}$:

$$P_T(Q = q) = \sum \{P_{\mathbf{PF}}(P \cup F') \mid P \cup F' \models Q = q\}. \tag{3}$$

Given some evidence $E = e$, which is a set of ground atoms (E) and their observed values (e), the probability of $Q = q$ becomes the conditional probability $P_T(Q = q | E = e) = P_T(Q = q, E = e)/P_T(E = e)$, as usual. If Q is a set of random variables, $P(Q)$ denotes its probability distribution.

EXAMPLE 2.1. *Consider the following probabilistic program T:*

```
0.2 :: Burglary.  0.3 :: Fire.  Alarm :- Burglary.  Alarm :- Fire.
```

Suppose we want to compute $P_T(\mathsf{Alarm} = 1)$. With two probabilistic facts, there are four total choices $F' \subseteq \{\mathsf{Burglary}, \mathsf{Fire}\}$. Taking $\mathbf{P} = \{\mathsf{Alarm} :- \mathsf{Burglary}., \mathsf{Alarm} :- \mathsf{Fire}.\}$, $\mathbf{P} \cup F' \models \mathsf{Alarm} = 1$ for any non-empty F'. Hence, $P_T(\mathsf{Alarm} = 1) = 0.2 \times 0.3 + 0.2 \times 0.7 + 0.8 \times 0.3 = 0.44$.

2.3 Adding Probabilistic Rules

We can augment the syntax and semantics of probabilistic programs to allow for probabilistic rules, like $p :: H :- B_1, \ldots, B_n.$, where probabilities are annotated to deterministic rules with non-empty bodies. In this case, a probabilistic program would be a pair $T = \langle \mathbf{P}, \mathbf{PR} \rangle$, where $\mathbf{P}$ is a logic program and $\mathbf{PR}$ is a set of probabilistic rules. Grounding the probabilistic rules, one would have a set $\{p_i :: R_i. | 1 \leq i \leq n\}$, where $R = \{R_1, \ldots, R_n\}$ is a set of ground (deterministic) rules. Then the probability of a total choice $R' \subseteq R$ entails the probability of a logic program $P_{\mathbf{PR}}((\mathbf{P} \cup R')) = \prod_{R_i \in R'} p_i \prod_{R_i \in R \setminus R'} (1 - p_i)$, which analogously defines a probability of a set of ground atoms Q have truth value q:

$$P_T(Q = q) = \sum \{P_{\mathbf{PR}}(P \cup F') \mid P \cup F' \models Q = q\}. \tag{4}$$

Using only probabilistic facts though, one can simulate probabilistic rules as well. Each probabilistic rule $p :: H :- B_1, \ldots, B_n.$ is equivalent to a pair formed by a deterministic rule $H :- B_1, \ldots, B_n, prob(id).$ and a probabilistic fact $p :: prob(id).$, where id is an identifier corresponding to this rule, and $prob(id)$ does not occur anywhere else [Fierens et al. 2015]. Due to this equivalence, ProbLog internally works only with probabilistic facts, not probabilistic rules, implicitly transforming each $p :: H :- B_1, \ldots, B_n.$ into $H :- B_1, \ldots, B_n, prob(id).$ and $p :: prob(id).$. When we refer to a probabilistic program, we mean the more economic definition notion without probabilistic rules, unless stated otherwise, knowing that probabilistic rules can be simply seen as syntactical sugar.

3. PARAMETER LEARNING IN PROBLOG

Here we take parameter learning to be the the task of, given some training data, finding the maximum-likelihood probabilities for a probabilistic program $\langle \mathbf{P}, \mathbf{PF} \rangle$, where both $\mathbf{P}$ and the atoms of $\mathbf{PF}$ (the program *structure*) are fixed. Formally, let $\mathbf{At}(T)$ be the set of all ground atoms from a probabilistic program $T = \langle \mathbf{P}, \mathbf{PF} \rangle$. An *observation* $E = e$ denotes a set of ground atoms $E \subseteq \mathbf{At}(T)$ together with their truth value $e \in \{0,1\}^{|E|}$, and a *dataset* is a set of observations. The parameter learning problem is defined via its input and output:

—**Input:** (i) a logic program $\mathbf{P}$ and a set of facts $\{F_1, \ldots, F_n\}$ (the structure of a probabilistic program $T_p = \langle \mathbf{P}, \mathbf{PF} \rangle$ where the set of probabilistic facts is $\mathbf{PF} = \{p_i :: F_i, 1 \leq i \leq n\}$ and the parameters are $p = \langle p_1, \ldots, p_n \rangle$); (ii) a dataset $D = \{E_1 = e_1, \ldots, E_m = e_m\}$ (the training examples);

—**Output:** the maximum-likelihood probabilities $\hat{p} = \langle \hat{p}_1, \ldots, \hat{p}_n \rangle$, where

$$\hat{p} = \mathsf{argmax}_p P_{T_p}(D) = \mathsf{argmax}_p \prod_{i=1}^{m} P_{T_p}(E_i = e_i)\,. \tag{5}$$

In the following, we briefly sketch the algorithms implemented in ProbLog to tackle the parameter learning problem, which are implemented in its function LFI (after "Learn From Interpretations") [Fierens et al. 2015].

3.1 Complete Data (Full Observability)

When an observation $E = e$ is such that $E = \mathbf{At}(T_p)$, we say it is *complete*. A dataset D is complete if each of its observation is so. In such case, the maximum-likelihood parameters can be computed straightforwardly via counting.

Consider a probabilistic program $T = \langle \mathbf{P}, \mathbf{PF} \rangle$, with a probabilistic fact $p_i :: F_i \in \mathbf{PF}$ that can be grounded to form $p_i :: F_{ij}.$. By definition, probabilistic atoms appearing in $\mathbf{PF}$ are not derived in $\mathbf{P}$. Consequently, the probability of the ground probabilistic atom F_{ij} being true, $P_T(F_{ij} = 1)$, is exactly the probability associated to the probabilistic fact $p_i :: F_i. \in \mathbf{PF}$ – that is, $P_T(F_{ij} = 1) = p_i$. Hence, the parameters $\hat{p}$ that maximize likelihood for a given complete dataset can be computed simply through the ratio of the groundings of the probabilistic fact $p_i :: F_i$ observed to be true.

Formally, let $F = \{F_1, \ldots, F_n\}$ be the set of probabilistic facts, and let $\{F_{ij} \mid 1 \leq j \leq Z_i\}$ be the set of possible groundings of F_i, for each $1 \leq i \leq n$. For a complete dataset $D = \{E_1 = e_1, \ldots, E_m = e_m\}$, the maximum-likelihood parameters $\hat{p} = \langle \hat{p}_1, \ldots, \hat{p}_n \rangle$ are given by, for $1 \leq i \leq n$:

$$\hat{p}_i = \frac{1}{mZ_i} \sum_{k=1}^{m} \sum_{j=1}^{Z_i} \delta_{i,j}^k, \text{ where } \delta_{i,j}^k = \begin{cases} 1 & \text{if } F_{ij} \in E_k; \\ 0 & \text{otherwise.} \end{cases} \tag{6}$$

The normalization factor mZ_i is the number of groundings of the probabilistic atom F_i observed through the whole dataset D. In the propositional case, $Z_i = 1$ for every $1 \leq i \leq n$. Typically, learning parameters for a relational probabilistic program is possible with a single observation $D = \{E = e\}$, as a large $Z_i >> 1$ guarantees many observable groundings for the same probabilistic fact $p_i :: F_i$.

3.2 Incomplete Data (Partial Observability)

In practice, learning with incomplete data is a common scenario. As the direct counting approach from the previous section is not an option when there is unobserved ground probabilistic atoms[1], the Expectation-Maximization (EM) algorithm [Dempster et al. 1977] has been the main tool to learn parameters in this situation. The idea behind the EM algorithm implemented in ProbLog is: (E-step) for each observation E_k, use a set of parameters p^t to compute the probability of each unobserved ground fact F_{ij} being true given E_k, which is the expected value of $\delta_{i,j}^k$ (from Equation (6)); (the M-step) employs these expected values and the observed $\delta_{i,j}^k$ to obtain new parameters p^{t+1} via Equation (6).

Formally, ProbLog's parameter learning function takes as input a logic program $\mathbf{P}$, a set of facts $\{F_1, \ldots, F_n\}$ and a dataset $D = \{E_1 = e_1, \ldots, E_m = e_m\}$. Let $F_{i1}, \ldots, F_{iZ_i}$ denote the groundings of the fact F_i, for each i, and let $T_p = \langle \mathbf{P}, \{p_1 :: F_1, \ldots, p_n :: F_n\} \rangle$ denote the probabilistic program determined by the parameters $\langle p_1, \ldots, p_n \rangle$. After setting the value of each $p_i \in [0, 1]$ randomly, the LFI-ProbLog algorithm ([Fierens et al. 2015]) iterates between the two steps below, until the likelihood ($P_{T_p}(D)$) increment is less than a threshold:

[1] With deterministic rules, observing the probabilistic atoms determine the truth value of derived atoms.

—(E-step): for each $\langle i, j, k\rangle$ s.t. $1 \leq i \leq n, 1 \leq j \leq Z_i, 1 \leq j \leq m$, $\delta^k_{i,j} = P_{T_p}(F_{ij} = 1|E_k = e_k)$;

—(M-step): for each $1 \leq i \leq n$, $p^t_i = \frac{1}{mZ_i} \sum_{k=1}^{m} \sum_{j=1}^{Z_i} \delta^k_{i,j}$.

When F_{ij} is in E_k, $P_{T_p}(F_{ij} = 1|E_k = e_k)$ in the E-step is simply 1 or 0, and there is no need for performing inference. In all other cases ProbLog must perform an inference for $P_{T_p}(F_{ij} = 1|E_k = e_k)$, although some optimizations are possible. For instance, ProbLog can detect when F_{ij} is independent from E_k, yielding $P_{T_p}(F_{ij} = 1|E_k = e_k) = p_i$. For more optimization details, see [Fierens et al. 2015].

3.3 Handling Probabilistic Rules

If the probabilistic program T_p, whose parameters we want to learn from a dataset D, has a probabilistic rule $p :: H :- B_1, \ldots, B_n.$, ProbLog interprets it as $H :- B_1, \ldots, B_n, prob(id).$ and $p :: prob(id).$. Hence, the introduction of the new probabilistic atom $prob(id)$ makes any observation incomplete. In other words, a probabilistic rule $p :: H :- B_1, \ldots, B_n.$ is inserted in the probabilistic program whose parameters are to be learned, ProbLog applies the algorithm sketched above no matter the dataset D, due to the fresh, unobserved atoms $prob(id)$. Each ground probabilistic rule is responsible for one of such new atoms and the larger the number of latent atoms, the slower is the convergence.

To circumvent that, the user himself could input probabilistic rules $p :: H :- B_1, \ldots, B_n.$ already translated into $H :- B_1, \ldots, B_n, prob(id).$ and $p :: prob(id).$, giving also the truth value of each $prob(id)$ within the input observations. Nonetheless, this approach is inviable mainly due to the fact that these atoms are essentially not observable. For instance, if B_1 is false in an observation, H is also false and there is no way to tell the value of $prob(id)$. Furthermore, it may the case that two probabilistic rules, $p_1 :: H :- B_1.$ and $p_2 :: H :- B_2$, share the same head, and if H, B_1, B_2 are all true, then there is no means to tell which $prob(id)$ is true.

4. OUR APPROACH

In short: there is no need for inserting an auxiliary atom for each probabilistic rule to perform parameter learning. Given a probabilistic program $T_p = \langle \mathbf{P}, \mathbf{PR}\rangle$ with probabilistic rules $\mathbf{PR} = \{p_i :: R_i. \mid 1 \leq i \leq n\}$, we can adopt the augmented semantics from Section 2.3. Thus, one can compute the likelihood of T_p for an observation $E = e$ directly through the expression for $P_{T_p}(E = e)$ given in Equation (4). This expression is a function of the parameters p_i, and its maximum yields the solution to the parameter learning problem. Henceforth, we adapt the learning problem to allow probabilistic rules: its input is a logic program $\mathbf{P}$, a set of rules $\{R_1, \ldots, R_n\}$ and a dataset $D = \{E_i = e_i \mid 1 \leq i \leq m\}$; and its output is the parameter vector $\langle p_1, \ldots, p_m\rangle$ yielding the probabilistic program $T_p = \langle \mathbf{P}, \mathbf{PR}\rangle$, with $\mathbf{PR} = \{p_i :: R_i \mid 1 \leq i \leq n\}$, that maximizes $P_{T_p}(D)$ (the likelihood).

4.1 Complete Data

When data are complete, we can write down the (log-)likelihood and maximize it directly. Let $\{A_1 = a_1, \ldots, A_o = a_o\}$ $(E = e)$ be a complete observation. Using the dependency graph, we can factor $P_{T_p}(E)$ in the usual way, employing the Markov condition: $P_{T_p}(E = e) = \prod_{i=1}^{o} P_{T_P}(A_i = a_i|Pa(A_i))$, where $Pa(A_i) \subseteq E$ is the set of ground atoms that are parents of A_i in the dependency graph – $\{A_i\} \cup Pa(A_i)$ is A_i's family. When considering a dataset $D = \{E_1 = e_q, \ldots, E_m = e_m\}$, we have that $P_{T_p}(D) = \prod_{k=1}^{m} P_{T_p}(E_k = e_k)$, and each $P_{T_p}(E_k = e_k)$ can be factored in this way.

Each of the terms $P_{T_p}(A_i = a_i|Pa(A_i))$ in $P_{T_p}(D)$ is a function depending only on the parameters attached to rules sharing as head a same ground atom A_i. Besides that, if A_j and A_i are groundings

of the same atom, $P_{T_p}(A_i|Pa(A_i))$ and $P_{T_p}(A_j|Pa(A_j))$ share the same parameters. If a set of ground atoms $A = \{A_1, \ldots, A_n\}$ share the same predicate symbol r, we call the set $A \cup \{Pa(A_i) \mid A_i \in A\}$ the *(predicate) family* of r. Using predicate families, we can partition the factors $P_{T_p}(A_i = a_i|Pa(A_i))$ of $P_{T_p}(D)$, and each partition can be maximized independently due to the disjoint parameters.

For instance, consider a probabilistic program T_p where the predicate $r_0(\cdot)$ appears in the head of exactly two rules, $p_1 :: r_0(X) :- r_1(X), \mathbf{not}\, r_2(X).$ and $p_2 :: r_0(X) :- \mathbf{not}\, r_1(X), r_2(X).$, and a complete observation $E = e$. The family of $r_0(\cdot)$ include all groundings of $r_0(\cdot)$, $r_1(\cdot)$ and $r_2(\cdot)$. For each possible instantiation of X, let $r(X) = \langle r_0(X), r_1(X), r_2(X)\rangle$ denote a set of ground atoms. Suppose that, for a_0 different values of X, $r(X) = \langle 1, 1, 0\rangle \in E$, for b_1 values of X, $r(X) = \langle 0, 1, 0\rangle \in E$, for b_0 values of X, $r(X) = \langle 1, 0, 1\rangle \in E$ and for b_1 values of X, $r(X) = \langle 0, 0, 1\rangle \in E$. The likelihood $P_{T_p}(E = e)$ then has a factor $p_1^{a_0}(1-p_1)^{a_1}p_2^{b_0}(1-p_2)^{b_1}$ (from r_0's family), maximized at $p_1 = a_0/(a_0+a_1)$ and $p_2 = b_0/(b_0+b_1)$ – and these are part of the $\langle p_1, \ldots, p_n\rangle$ maximizing $P_{T_p}(E = e)$.

We noticed that several other combinations of rules sharing the same head lead to likelihood factors whose maximizations have exact solutions. In our implementations we used exact solutions to find maximum likelihood parameters whenever possible. When the likelihood expressions cannot be maximized in a closed form, we resort to a gradient-based numerical optimization to find the maximum-likelihood parameters $\langle p_1, \ldots, p_n\rangle$.

In comparison to ProbLog's method, our approach dispenses with the EM-like algorithm when the data are complete, even with probabilistic rules. Consequently, the whole learning problem is solved in a single numerical optimization for each predicate family.

4.2 Incomplete Data

When the data are incomplete, we employ an EM algorithm, although we can also avoid inserting an auxiliary variable for each ground probabilistic rule. Suppose we have a probabilistic program $T_p = \langle \mathbf{P}, \mathbf{PR}\rangle$ and an incomplete observation $E = e$. Let $Z = \mathbf{At}(T_p) \setminus E$ be the set of unobserved ground atoms. For each complete observation $E = e, Z = z$, we can express the (log-)likelihood in terms of the parameters $p = \langle p_1, \ldots, p_n\rangle$, as explained in the section above. Thus, a probability distribution over Z yields an expected value for the log-likelihood. As an observation $E = e$ determines a conditional probability distribution for Z, it also determines an expected log-likelihood. When considering a dataset $D = \{E_1 = e_1, \ldots, E_m = e_m\}$, the expected log-likelihood can be computed by summing over the observations. Setting initially $p_i = 0.5$ for each parameter p_i, our implementation of the EM algorithm iterates between the following steps until some convergence criterion is met:

—(E-step): Given a set of parameters p, for each $E_k = e_k \in D$, compute $P_{T_p}(Z = z|E_k = e_k)$ for each $z \in \{0,1\}^{|Z|}$;

—(M-step): Find the set of parameters maximizing the expected log-likelihood:

$$\mathsf{argmax}_{p'} \sum_{k=1}^{m} \sum_{z \in \{0,1\}^{|Z|}} P_{T_p}(Z = z|E_k = e_k) \ln(P_{T_{p'}}(E_k = e_k, Z = z)). \tag{7}$$

To compute the terms $P_{T_p}(Z = z|E_k = e_k)$, we employ ProbLog's inference. In principle, for each $E_k = e_k$, each z would yield an inference in the E-step, but we can do much better. Let $\mathcal{F}(A_i)$ denote the family of a ground atom A_i. If $c \in \{0,1\}^{|Q|}$ is a vector of truth values associated to a set Q of ground atoms $(Q = c)$, we use $c_{Q'} \in \{0,1\}^{|Q'|}$ to denote the vector of truth values corresponding to the subset $Q' \subseteq Q$. Now the expected log-likelihood, for a each $E_k = e_k$, can be rewritten as

$$\sum_{A_i \in \mathbf{At}(T_p)} \sum_{c \in \{0,1\}^{|\mathcal{F}(A_i)|}} P_{T_p}(\mathcal{F}(A_i) = c|E_k = e_k) \ln(P_{T_{p'}}(A_i = c_{A_i}|Pa(A_i) = c_{Pa(A_i)})). \tag{8}$$

Thus, assuming the observations are consistent with the model, we only need to make inferences $P_{T_p}(\mathcal{F}(A_i) = c|E_k = e_k)$ for those c such that $P_{T_{p'}}(A_i = c_{A_i}|Pa(A_i) = c_{Pa(A_i)})$ is a (non-constant) function of the parameters, for the others can be ignored during the maximization. For instance, suppose that the ground atom H is the head of a single ground probabilistic rule $p :: H :- B_1, B_2..$ There are eight possible values for c regarding the H's family $\{H, B_1, B_2\}$. Nonetheless, we have to consider only two of them, $c = \langle 0, 1, 1\rangle$ and $c = \langle 1, 1, 1\rangle$, for the other values of c either yield a constant $P_{T_{p'}}(H = c_H|Pa(H) = c_{Pa(H)}) = 1$ or $P_{T_p}(\mathcal{F}(H) = c|E_k = e_k) = 0$.

5. EXPERIMENTS AND RESULTS

The goal of the experiments is to compare our approach with LFI-ProbLog algorithm for varying missing data rates and dataset sizes. To accomplish this, we used the following program:

0.3 :: fire(X) :− person(X).	0.9 :: alarm(X) :− burglary(X).
0.4 :: burglary(X) :− person(X).	0.8 :: cares(X, Y) :− person(X), person(Y).
0.7 :: alarm(X) :− fire(X).	0.8 :: calls(X, Y) :− cares(X, Y), alarm(Y), **not** samePerson(X, Y).

To generate the datasets we sampled from the model above for a given number of constants (dataset size). For each constant c deterministic facts person(c) and samePerson(c, c) were added to the model. To impose a missing rate we discarded part of the generated observations using a pseudo random function. All tests were run four times, each time with a new independently sampled dataset. Each datapoint in Table 1 therefore corresponds to the average computation time among these four runs.

ProbLog's stopping criteria is defined over the convergence of the log-likelihood values observed. Notice, however, as ProbLog's iterative process differs substantially from our algorithm's, using the same stopping criteria does not guarantee similar log-likelihood values are reached. In order to ensure a valid comparison basis, we set a fixed Problog's stopping criteria—1×10^{-3}, which means it stops when the log-likelihood variation between subsequent iterations is smaller than 1×10^{-3}—and set our algorithm to stop whenever it reaches an equal or better log-likelihood value than ProbLog for the same dataset.

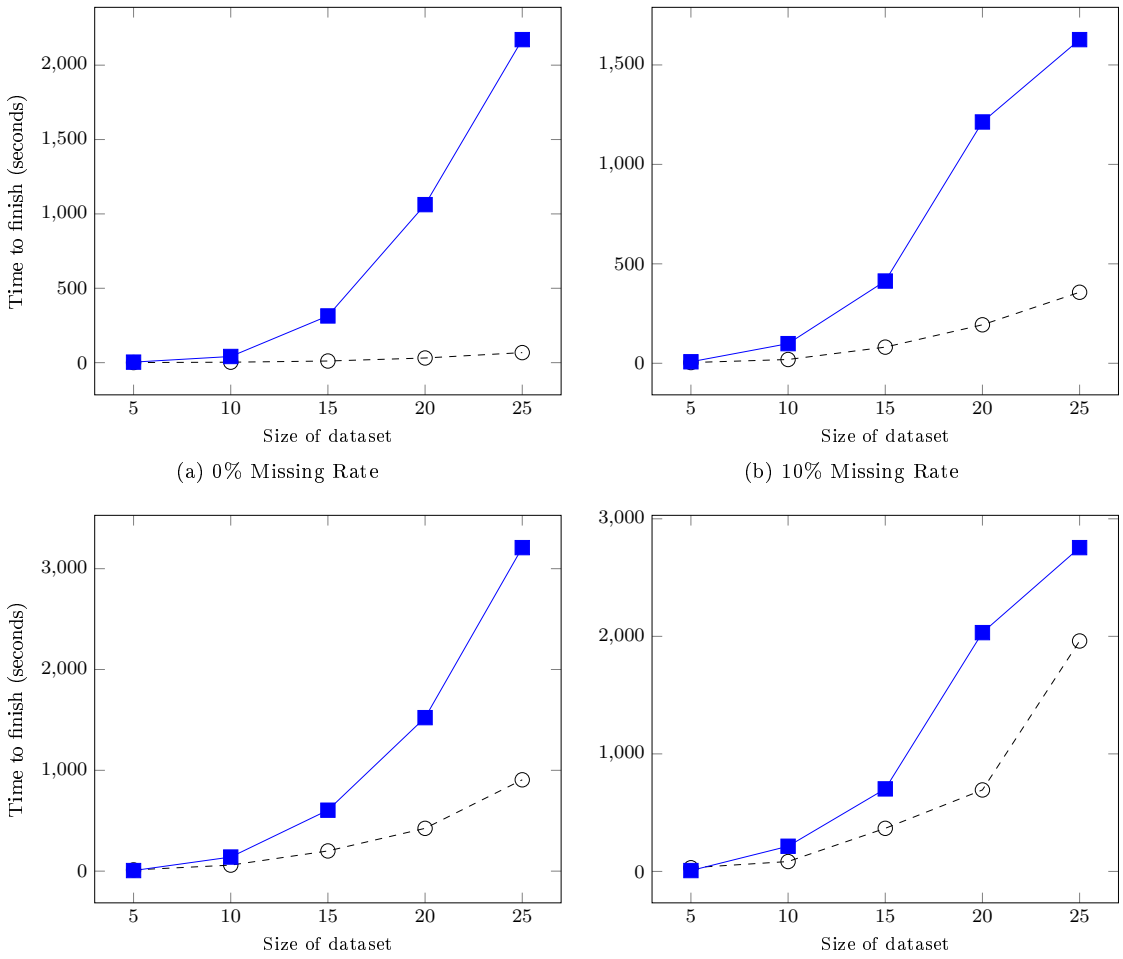
All experiments were performed in parallel on a dedicated machine with the following specifications: 8 vCPU, 2900 MHz, 15 GiB RAM.

From the results we can see that our algorithm outperforms ProbLog in the vast majority of cases, only losing for datasets with 5 constants and missing rate above or equal to 20%. It is also worth noting that, when the size of the datasets increases, the ratio between our approach and ProbLog's computation time tends to decrease.

Dataset size is limited to 25 constants because for larger datasets ProbLog approximated the likelihood values to zero, returning an error when trying to calculate its log.

6. CONCLUSION

We have presented a new approach to learn the parameters for a probabilistic program with probabilistic rules. We have shown how one can avoid the insertion of latent variables for probabilistic rules. In particular, this avoids the need for using the EM-algorithm when the data are complete. Experiments indicates significant gains in time, when comparing to ProbLog, even when there is missing data. Future work includes applying these techniques to perform structure learning; that is, learning the rules of a probabilistic program.



(a) 0% Missing Rate (b) 10% Missing Rate

(c) 20% Missing Rate (d) 30% Missing Rate

Fig. 1: Time to learn parameters from incomplete relational data. Size of dataset is the the number of constants in the program. Solid squares were generated by ProbLog; empty circles were generated by our algorithm.

REFERENCES

De Raedt, L., Frasconi, P., Kersting, K., and Muggleton, S. H. *Probabilistic inductive logic programming*. Springer, 2008.

De Raedt, L., Kimmig, A., and Toivonen, H. Problog: A probabilistic prolog and its application in link discovery, 2007.

Dempster, A. P., Laird, N. M., and Rubin, D. B. Maximum likelihood from incomplete data via the em algorithm. *Journal of the royal statistical society. Series B (methodological)*, 1977.

Fierens, D., Van den Broeck, G., Renkens, J., Shterionov, D., Gutmann, B., Thon, I., Janssens, G., and De Raedt, L. Inference and learning in probabilistic logic programs using weighted boolean formulas. *Theory and Practice of Logic Programming* 15 (3): 358–401, 2015.

Getoor, L. and Taskar, B. *Introduction to statistical relational learning*. MIT press, 2007.

Halpern, J. Y. An analysis of first-order logics of probability. *Artificial intelligence* 46 (3): 311–350, 1990.

Jaeger, M. Relational bayesian networks. In *Proceedings of the Thirteenth conference on Uncertainty in artificial intelligence*. Morgan Kaufmann Publishers Inc., pp. 266–273, 1997.

Ognjanovic, Z. and Rašković, M. Some first-order probability logics. *Theoretical Computer Science* 247 (1): 191–212, 2000.

Richardson, M. and Domingos, P. Markov logic networks. *Machine learning* 62 (1): 107–136, 2006.