EM over Binary Decision Diagrams for Probabilistic Logic Programs

Elena Bellodi Fabrizio Riguzzi*
ENDIF-Dipartimento di Ingegneria, Università di Ferrara
Via Saragat 1, 44122 Ferrara, Italy
fabrizio.riguzzi@unife.it
elena.bellodi@unife.it

April 18, 2011

Abstract

Recently much work in Machine Learning has concentrated on using expressive representation languages that combine aspects of logic and probability. A whole field has emerged, called Statistical Relational Learning, rich of successful applications in a variety of domains. In this paper we present a Machine Learning technique targeted to Probabilistic Logic Programs, a family of formalisms where uncertainty is represented using Logic Programming tools. Among various proposals for Probabilistic Logic Programming, the one based on the distribution semantics is gaining popularity and is the basis for a number of languages, such as ICL, PRISM, ProbLog and Logic Programs with Annotated Disjunctions. This paper proposes a technique for learning parameters of these languages. Since their equivalent Bayesian networks contain hidden variables, an EM algorithm is adopted. In order to speed the computation up, expectations are computed directly on the Binary Decision Diagrams structures that are built for inference. The resulting system, called Emblem for "EM over BDDs for probabilistic Logic programs Efficient Mining", has been applied to a number of datasets and showed good performances both in terms of speed and memory usage. In particular its speed allows the execution of a high number of restarts, resulting in better solutions quality.

Keywords Statistical Relational Learning, Probabilistic Logic Programs, LPAD, EM

1 Introduction

Machine Learning has seen the development of the field of Statistical Relational Learning, where logical-statistical languages are used in order to effectively learn

*Corresponding author, Tel. +390532974836, Fax +390532974870

in complex domains involving relations and uncertainty. These techniques have been successfully applied in social networks analysis, entity recognition, collective classification and information extraction, to name a few.

Similarly, in the field of Logic Programming, a large number of works have started to appear that combine logic and probability. Among these, many share a common approach to defining the semantics of the proposed languages: the distribution semantics [31]. It underlies for example Probabilistic Logic Programs [2], Probabilistic Horn Abduction [21], PRISM [31], the Independent Choice Logic [22], pD [7], Logic Programs with Annotated Disjunctions (LPADs) [40], ProbLog [5] and CP-logic [38]. The approach is particularly appealing for its intuitiveness and because efficient inference algorithms have started to appear [5, 26, 28, 14, 19]. Most of these techniques use Binary Decision Diagrams (BDD) for inference: explanations for the query are found and the probability of the query is computed by building a BDD.

In this paper we present the *Emblem* system for "EM over BDDs for probabilistic Logic programs Efficient Mining" that learns parameters of probabilistic logic programs under the distribution semantics by using an EM algorithm. The system exploits the fact that the translation of these programs into graphical models generates models with hidden variables and therefore an EM approach is necessary. Its main characteristic is that it computes the values of expectations using BDDs. *Emblem* is developed for the language of LPADs and tested on the IMDB, Cora and UW-CSE datasets and compared with RIB [30], LeProbLog [5], Alchemy [23] and CEM, an implementation of EM based on the `cplint` interpreter [26].

The paper is organized as follows. Section 2 presents Probabilistic Logic Programming, concentrating on LPADs. Section 3 describes *Emblem* together with an example of its execution. In Section 4 the results of the experiments performed are presented. Section 5 discusses related works and Section 6 concludes the paper.

2 Probabilistic Logic Programming

Many languages have been proposed that integrate logic programming with probability theory. One of the most interesting approaches to the integration is the distribution semantics [31], which was introduced for the PRISM language but is shared by many other languages. A program in one of these languages defines a probability distribution over normal logic programs called *worlds*. This distribution is then extended to queries and the probability of a query is obtained by marginalizing the joint distribution of the query and the programs.

The distribution semantics has been defined both for programs that do not contain function symbols, and thus have a finite set of worlds, and for programs that contain them, that have an infinite set of worlds. We review here the first case for the sake of simplicity. Let us call $P(W)$ the distribution over worlds. The probability of a query Q given a world w is $P(Q|w) = 1$ if $w \models Q$ and 0 otherwise, where $\models$ is truth in the well-founded model [37]. Thus the probability

of a query Q is given by

$$P(Q) = \sum_{w \in W} P(Q, w) = \sum_{w \in W} P(Q|w)P(w) = \sum_{w \in W : w \models Q} P(w) \qquad (1)$$

The languages following the distribution semantics differ in the way they define the distribution over logic programs. Each language allows probabilistic choices among atoms in clauses: Probabilistic Logic Programs, PHA, ICL, PRISM, and ProbLog allow probability distributions over facts, while LPADs allow probability distributions over the heads of disjunctive clauses. All these languages have the same expressive power: there are transformations with linear complexity that can convert each one into the others [39, 4]. In this paper we will use LPADs for their general syntax.

In Logic Programs with Annotated Disjunctions the alternatives are encoded in the head of clauses in the form of a disjunction in which each atom is annotated with a probability. Each grounding of an annotated disjunctive clause represents a probabilistic choice between a number of ground normal clauses. By choosing a head atom for each grounding of each clause we get a *world*. The probability of the world is given by the product of the annotations of the atoms selected.

Formally a *Logic Program with Annotated Disjunctions* [40] consists of a finite set of annotated disjunctive clauses. An annotated disjunctive clause C_i is of the form $h_{i1} : \Pi_{i1}; \ldots; h_{in_i} : \Pi_{in_i} : -b_{i1}, \ldots, b_{im_i}$. In such a clause $h_{i1}, \ldots h_{in_i}$ are logical atoms and $b_{i1}, \ldots, b_{im_i}$ are logical literals, $\{\Pi_{i1}, \ldots, \Pi_{in_i}\}$ are real numbers in the interval $[0, 1]$ such that $\sum_{k=1}^{n_i} \Pi_{ik} \leq 1$. $b_{i1}, \ldots, b_{im_i}$ is called the *body* and is indicated with $body(C_i)$. Note that if $n_i = 1$ and $\Pi_{i1} = 1$ the clause corresponds to a non-disjunctive clause. If $\sum_{k=1}^{n_i} \Pi_{ik} < 1$ 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 $1 - \sum_{k=1}^{n_i} \Pi_{ik}$. We denote by $ground(T)$ the grounding of an LPAD T.

An *atomic choice* is a triple (C_i, θ_j, k) where $C_i \in T$, θ_j is a substitution that grounds C_i and $k \in \{1, \ldots, n_i\}$. (C_i, θ_j, k) means that, for the ground clause $C_i\theta_j$, the head h_{ik} was chosen. In practice $C_i\theta_j$ corresponds to a random variable X_{ij} and an atomic choice (C_i, θ_j, k) to an assignment $X_{ij} = k$. A set of atomic choices κ is *consistent* if $(C, \theta, i) \in \kappa, (C, \theta, j) \in \kappa \Rightarrow i = j$, i.e., only one head is selected for a ground clause. A *composite choice* κ is a consistent set of atomic choices.

The *probability* $P(\kappa)$ *of a composite choice* κ is the product of the probabilities of the individual atomic choices, i.e. $P(\kappa) = \prod_{(C_i, \theta_j, k) \in \kappa} \Pi_{ik}$.

A *selection* σ is a composite choice that, for each clause $C_i\theta_j$ in $ground(T)$, contains an atomic choice (C_i, θ_j, k). We denote the set of all selections σ of a program T by $\mathcal{S}_T$. A selection σ identifies a normal logic program w_σ defined as $w_\sigma = \{(h_{ik} \leftarrow body(C_i))\theta_j | (C_i, \theta_j, k) \in \sigma\}$. w_σ is called a *world* of T. Since selections are composite choices, we can assign a probability to possible worlds: $P(w_\sigma) = P(\sigma) = \prod_{(C_i, \theta_j, k) \in \sigma} \Pi_{ik}$.

We consider only *sound* LPADs in which every possible world has a total

well-founded model. Subsequently we will write $w_\sigma \models Q$ to mean that the query Q is true in the well-founded model of the program w_σ.

The probability of a query Q according to an LPAD T is given by

$$P(Q) = \sum_{\sigma \in E(Q)} P(\sigma) \tag{2}$$

where we define $E(Q) = \{\sigma \in \mathcal{S}_T, w_\sigma \models Q\}$ the set of selections corresponding to worlds where the query is true.

Sometimes a simplification of this semantics can be used to reduce the computational cost of answering queries. In this simplified semantics random variables are directly associated to clauses in the programs rather than to their ground instantiations. So, for a clause C_i, possibly non ground, there is a random variable X_i. In this way the number of random variables may be significantly reduced and atomic choices take the form (C_i, k), meaning that head h_{ik} is selected from program clause C_i, i.e., that $X_i = k$. In some of the experiments in Section 4 we use this simplification to contain the computations cost.

Example 1 *The following LPAD T encodes a very simple model of the development of an epidemic or pandemic:*

$C_1 = epidemic : 0.6; pandemic : 0.3 : -flu(X), cold.$
$C_2 = cold : 0.7.$
$C_2 = flu(david).$
$C_3 = flu(robert).$

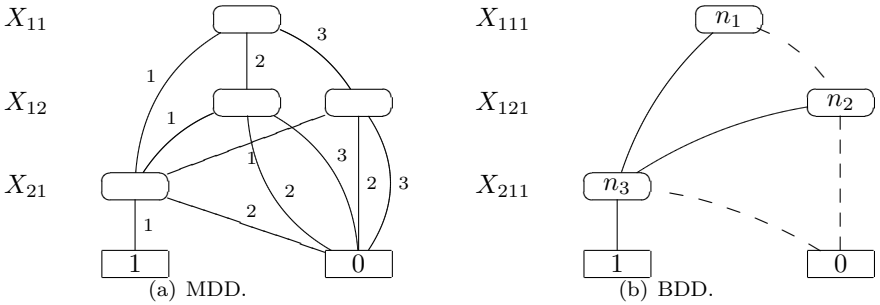
This program models the fact that if somebody has the flu and the climate is cold, there is the possibility that an epidemic or a pandemic arises. We are uncertain about whether the climate is cold but we know for sure that David and Robert have the flu.

Clause C_1 has two groundings, $C_1\theta_1$ with $\theta_1 = \{X/david\}$ and $C_1\theta_2$ with $\theta_2 = \{X/robert\}$ so there are two random variables X_{11} and X_{12}.

T has 18 instances, the query epidemic is true in 5 of them and its probability is $P(epidemic) = 0.6{\cdot}0.6{\cdot}0.7{+}0.6{\cdot}0.3{\cdot}0.7{+}0.6{\cdot}0.1{\cdot}0.7{+}0.3{\cdot}0.6{\cdot}0.7{+}0.1{\cdot}0.6{\cdot}0.7 = 0.588$

In the simplified semantics C_1 is associated to a single random variable X_1. In this case T has 6 instances, the query epidemic is true in 1 of them and its probability is $P(epidemic) = 0.6 \cdot 0.7 = 0.42$.

The possible worlds in which a query is true can be represented using a Multivalued Decision Diagram (MDD) [35]. An MDD represents a function $f(\mathbf{X})$ taking Boolean values on a set of multivalued variables $\mathbf{X}$ by means of a rooted graph that has one level for each variable. Each node is associated to the variable of its level and has one child for each possible value of the variable. The leaves store either 0 or 1. Given values for all the variables $\mathbf{X}$, we can compute the value of $f(\mathbf{X})$ by traversing the graph starting from the root and returning the value associated to the leaf that is reached. An MDD can be used to represent the set $E(Q)$ by considering the multivalued variable X_{ij} associated



(a) MDD. (b) BDD.

Figure 1: Decision diagrams for Example 1.

to $C_i\theta_j$ of $ground(T)$. X_{ij} has values $\{1, \ldots, n_i\}$ and atomic choice (C_i, θ_j, k) corresponds to the propositional equation $X_{ij} = k$. If we represent with the MDD the function $f(\mathbf{X}) = \bigvee_{\sigma \in E(Q)} \bigwedge_{(C_i,\theta_j,k)\in\sigma} X_{ij} = k$ then the MDD will have a path to a 1-leaf for each possible world where Q is true. MDDs can be built by combining simpler MDDs using Boolean operators. While building MDDs, simplification operations can be applied that delete or merge nodes. Merging is performed when the diagram contains two identical sub-diagrams, while deletion is performed when all arcs from a node point to the same node. In this way a reduced MDD is obtained, that often has a much smaller number of nodes with respect to a Multivalued Decision Tree (MDT), i.e., an MDD in which every node has a single parent and all the children belong to the level immediately below.

For example, the reduced MDD corresponding to the query *epidemic* from Example 1 is shown in Figure 1(a). The labels on the edges represent the values of the variable associated to the node.

It is often unfeasible to find all the instances where the query is true so inference algorithms find instead *explanations* for the query, i.e. composite choices such that the query is true in all the worlds whose selections are a superset of them. Explanations however, differently from possible worlds, are not necessarily mutually exclusive with respect to each other, so the probability of the query can not be computed by a summation as in Formula 2. The explanations have first to be made disjoint so that a summation can be computed. Since MDDs split paths on the basis of the values of a variable, the branches are mutually disjoint so a dynamic programming algorithm can be applied for computing the probability.

Most packages for the manipulation of a decision diagram are however restricted to work on Binary Decision Diagrams (BDD), i.e., decision diagrams where all the variables are Boolean. These packages offer Boolean operators between BDDs and apply simplification rules to the result of operations in order to reduce as much as possible the size of the BDD, obtaining a reduced BDD. Usually reduced BDDs have a much smaller number of nodes than the equivalent Binary Decision Tree (BDT).

A node n in a BDD has two children: the 1-child, also indicated with $child_1(n)$, and the 0-child, also indicated with $child_0(n)$. When drawing BDDs, rather than using edge labels, the 0-branch, the one going to the 0-child, is distinguished from the 1-branch by drawing it with a dashed line.

To work on MDDs with a BDD package we must represent multivalued variables by means of binary variables. Various options are possible, we found that the following, proposed in [4], gives the best performance. For a multivalued variable X_{ij}, corresponding to ground clause $C_i\theta_j$, having n_i values, we use $n_i - 1$ Boolean variables $X_{ij1}, \ldots, X_{ijn_i-1}$ and we represent the equation $X_{ij} = k$ for $k = 1, \ldots n_i - 1$ by means of the conjunction $\overline{X_{ij1}} \wedge \overline{X_{ij2}} \wedge \ldots \wedge \overline{X_{ijk-1}} \wedge X_{ijk}$, and the equation $X_{ij} = n_i$ by means of the conjunction $\overline{X_{ij1}} \wedge \overline{X_{ij2}} \wedge \ldots \wedge \overline{X_{ijn_i-1}}$. BDDs obtained in this way can be used as well for computing the probability of queries by associating to each Boolean variable X_{ijk} a parameter π_{ik} that represents $P(X_{ijk} = 1)$. If we define $g(i) = \{j|\theta_j$ is a substitution grounding $C_i\}$ then $\pi_{ik} = P(X_{ijk} = 1)$ for all $j \in g(i)$. The parameters are obtained from those of multivalued variables in this way:

$$\begin{aligned} \pi_{i1} &= \Pi_{i1} \\ &\ldots \\ \pi_{ik} &= \frac{\Pi_{ik}}{\prod_{j=1}^{k-1}(1-\pi_{ij})} \\ &\ldots \end{aligned}$$

up to $k = n_i - 1$.

3 Emblem

Emblem applies the algorithm for performing EM over BDDs proposed in [36, 12, 13, 11] to the problem of learning the parameters of an LPAD. *Emblem* takes as input a number of goals that represent the examples. For each goal it generates the BDD encoding its explanations. The typical input for *Emblem* will be a set of interpretations, i.e., sets of ground facts, each describing a portion of the domain of interest. Among the predicates for the input facts the user has to indicate which are target predicates: the facts for these predicates will then form the queries for which the BDDs are built. The predicates can be treated as closed-world or open-world. In the first case the body of clauses with a target predicate in the head is resolved only with facts in the interpretation. In the second case, the body of clauses with a target predicate in the head is resolved both with facts in the interpretation and with clauses in the theory. If the last option is set and the theory is cyclic, we use a depth bound on SLD-derivations to avoid going into infinite loops, as proposed by [9]. Given the program constituted by clauses C_1 and C_2 from Example 1 and the interpretation $\{epidemic, flu(david), flu(robert)\}$, we obtain the BDD in Figure 1(b) that represents the query *epidemic*.

Then *Emblem* enters the EM cycle, in which the steps of expectation and maximization are repeated until the log-likelihood of the examples reaches a local maximum.

Let us now present the formulas for the expectation and maximization phases for the case of a single example Q:

- Expectation: computes $\mathbf{E}[c_{ik0}|Q]$ and $\mathbf{E}[c_{ik1}|Q]$ for all rules C_i and $k = 1, \ldots, n_i - 1$, where c_{ikx} is the number of times a variable X_{ijk} takes value x for $x \in \{0, 1\}$, with j in $g(i)$. $\mathbf{E}[c_{ikx}|Q]$ is given by $\sum_{j \in g(i)} P(X_{ijk} = x|Q)$.
- Maximization: computes π_{ik} for all rules C_i and $k = 1, \ldots, n_i - 1$.

$$\pi_{ik} = \frac{\mathbf{E}[c_{ik1}|Q]}{\mathbf{E}[c_{ik0}|Q] + \mathbf{E}[c_{ik1}|Q]}$$

If we have more than one example the contributions of each example simply sum up when computing $\mathbf{E}[c_{ijx}]$.

$P(X_{ijk} = x|Q)$ is given by $P(X_{ijk} = x|Q) = \frac{P(X_{ijk}=x,Q)}{P(Q)}$ with

$$\begin{aligned} P(X_{ijk} = x, Q) &= \sum_{\sigma \in E(Q)} P(Q, X_{ijk} = x, \sigma) \\ &= \sum_{\sigma \in E(Q)} P(Q|\sigma) P(X_{ijk} = x|\sigma) P(\sigma) \\ &= \sum_{\sigma \in E(Q)} P(X_{ijk} = x|\sigma) P(\sigma) \end{aligned}$$

where $P(X_{ijk} = 1|\sigma) = 1$ if $(C_i, \theta_j, k) \in \sigma$ for $k = 1, \ldots, n_i - 1$ and 0 otherwise and $P(X_{ijk} = 0|\sigma) = 1$ if $(C_i, \theta_j, k') \in \sigma$ with $k < k'$, for $k' = 1, \ldots, n_i - 1$ or if $(C_i, \theta_j, n_i) \in \sigma$ if $k = n_i$ and 0 otherwise.

Since there is a one to one correspondence between the possible worlds where Q is true and the paths to a 1 leaf in a BDT,

$$P(X_{ijk} = x, Q) = \sum_{\rho \in R(Q)} P(X_{ijk} = x|\rho) \prod_{d \in \rho} \pi(d)$$

where σ corresponds to ρ ($P(X_{ijk} = x|\sigma) = P(X_{ijk} = x|\rho)$), $R(Q)$ is the set of paths in the BDD for query Q that lead to a 1 leaf, d is an edge of ρ and $\pi(d)$ is the probability associated to the edge: if d is the 1-branch from a node associated to a variable X_{ijk}, then $\pi(d) = \pi_{ik}$, if d is the 0-branch from a node associated to a variable X_{ijk}, then $\pi(d) = 1 - \pi_{ik}$.

Now consider a BDT in which only the merge rule is applied, fusing together identical sub-diagrams. For example, by applying only the merge rule in Example 1 the diagram in Figure 2 is obtained. The resulting diagram, that we call Complete Binary Decision Diagram (CBDD), is such that every path contains a node for every level.

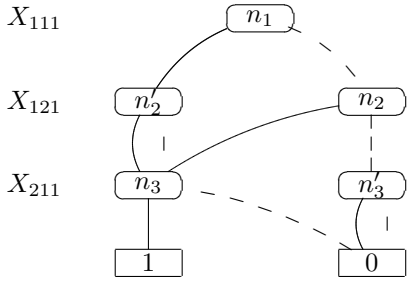


Figure 2: Decision diagram after applying the merge rule for Example 1.

For a CBDD $P(X_{ijk} = x, Q)$ can be further expanded as

$$P(X_{ijk} = x, Q) = \sum_{\rho \in R(Q), (X_{ijk}=x) \in \rho} \prod_{d \in \rho} \pi(d)$$

where $(X_{ijk} = x) \in \rho$ means that ρ contains an x-edge from a node associated to X_{ijk}. We can then write

$$P(X_{ijk} = x, Q) = \sum_{n \in N(Q), v(n)=X_{ijk}, \rho_n \in R_n(Q), \rho^n \in R^n(Q,x)} \prod_{d \in \rho^n} \pi(d) \prod_{d \in \rho_n} \pi(d)$$

where $N(Q)$ is the set of nodes of the BDD, $v(n)$ is the variable associated to node n, $R_n(Q)$ is the set containing the paths from the root to n and $R^n(Q, x)$ is the set of paths from n to the 1 leaf through its x-child.

$$\begin{aligned} P(X_{ijk} = x, Q) &= \sum_{n \in N(Q), v(n)=X_{ijk}} \sum_{\rho_n \in R_n(Q)} \sum_{\rho^n \in R^n(Q,x)} \prod_{d \in \rho^n} \pi(d) \prod_{d \in \rho_n} \pi(d) \\ &= \sum_{n \in N(Q), v(n)=X_{ijk}} \sum_{\rho_n \in R_n(Q)} \prod_{d \in \rho_n} \pi(d) \sum_{\rho^n \in R^n(Q,x)} \prod_{d \in \rho^n} \pi(d) \\ &= \sum_{n \in N(Q), v(n)=X_{ijk}} F(n) B(child_x(n)) \pi_{ikx} \end{aligned}$$

where π_{ikx} is π_{ik} if x=1 and $(1 - \pi_{ik})$ if x=0. $F(n)$ is the *forward probability* [13], the probability mass of the paths from the root to n, while $B(n)$ is the *backward probability* [13], the probability mass of paths from n to the 1 leaf. If *root* is the root of a tree for a query Q then $B(root) = P(Q)$.

The expression $F(n)B(child_x(n))\pi_{ikx}$ represents the sum of the probabilities of all the paths passing through the x-edge of node n. We indicate with $e^x(n)$ such an expression. Thus

$$P(X_{ijk} = x, Q) = \sum_{n \in N(Q), v(n)=X_{ijk}} e^x(n) \tag{3}$$

For the case of a BDD, i.e., a diagram obtained by applying also the deletion rule, Formula 3 is no longer valid since also paths where there is no node associated to X_{ijk} can contribute to $P(X_{ijk} = x, Q)$. In fact, it is necessary to consider also the deleted paths: suppose that a node n associated to variable Y has a level higher than variable X_{ijk} and suppose that $child_0(n)$ is associated to variable W that has a level lower than variable X_{ijk}. The nodes associated to variable X_{ijk} have been deleted from the paths from n to $child_0(n)$. One can imagine that the current BDD has been obtained from a BDD having a node m associated to variable X_{ijk} that is a descendant of n along the 0-branch and whose outgoing edges both point to $child_0(n)$. The original BDD can be reobtained by applying a deletion operation that merges the two paths passing through m. The probability mass of the two paths that were merged was $e^0(n)(1-\pi_{ik})$ and $e^0(n)\pi_{ik}$ for the paths passing through the 0-child and 1-child of m respectively.

Formally, let $Del^x(X)$ be the set of nodes n such that the level of X is below that of n and is above that of $child_x(n)$, i.e., X is deleted between n and $child_x(n)$. For the BDD in Figure 1(b), for example, $Del^1(X_{121}) = \{n_1\}$, $Del^0(X_{121}) = \{\}$, $Del^1(X_{221}) = \{\}$, $Del^0(X_{221}) = \{n_3\}$. Then

$$
\begin{aligned}
P(X_{ijk} = 0, Q) &= \sum_{n \in N(Q), v(n) = X_{ijk}} e^x(n) + \\
&\quad (1 - \pi_{ik}) \left(\sum_{n \in Del^0(X_{ijk})} e^0(n) + \sum_{n \in Del^1(X_{ijk})} e^1(n) \right) \\
P(X_{ijk} = 1, Q) &= \sum_{n \in N(Q), v(n) = X_{ijk}} e^x(n) + \\
&\quad \pi_{ik} \left(\sum_{n \in Del^0(X_{ijk})} e^0(n) + \sum_{n \in Del^1(X_{ijk})} e^1(n) \right)
\end{aligned}
$$

Having shown how to compute the probabilities, we now describe *Emblem* in detail.

Emblem's main procedure, shown in Algorithm 1, consists of a cycle in which the procedures Expectation and Maximization are repeatedly called. Procedure Expectation returns the log likelihood of the data that is used in the stopping criterion: *Emblem* stops when the difference between the log likelihood of this iteration and the one of the previous iteration drops below a threshold ϵ or when this difference is below a fraction δ of the current log likelihood.

Procedure Expectation, shown in Algorithm 2, takes as input a list of BDDs, one for each example, and computes the expectation for each one, i.e. $P(Q, X_{ijk} = x)$ for all variables X_{ijk} in the BDD. In the procedure we use $\eta^x(i,k)$ to indicate $\sum_{j \in g(i)} P(Q, X_{ijk} = x)$. Expectation first calls GetForward and GetBackward that compute the forward, the backward probability of nodes and $\eta^x(i,k)$ for non-deleted paths only. Then it updates $\eta^x(i,k)$ to take into account deleted paths.

Algorithm 1 Procedure EMBLEM

```
1: function EMBLEM(ϵ,δ)
2:     Build BDDs
3:     LL = −inf
4:     repeat
5:         LL_0 = LL
6:         LL = Expectation(BDDs)
7:         Maximization
8:     until LL − LL_0 < ϵ ∨ LL − LL_0 < −LL · δ
9:     return LL, π_ik for all i, k
10: end function
```

Procedure Maximization (Algorithm 3) computes the parameters values for the next EM iteration.

Algorithm 2 Procedure Expectation

```
1: function Expectation(BDDs)
2:     LL = 0
3:     for all BDD ∈ BDDs do
4:         for all i ∈ Rules do
5:             for k = 1 to n_i − 1 do
6:                 η^0(i,k) = 0;  η^1(i,k) = 0
7:             end for
8:         end for
9:         for all variables X do
10:            ς(X) = 0
11:        end for
12:        GetForward(root(BDD))
13:        Prob=GetBackward(root(BDD))
14:        T = 0
15:        for l = 1 to levels(BDD) do
16:            Let X_ijk be the variable associated to level l
17:            T = T + ς(X_ijk)
18:            η^0(i,k) = η^0(i,k) + T × (1 − π_ik)
19:            η^1(i,k) = η^1(i,k) + T × π_ik
20:        end for
21:        for all i ∈ Rules do
22:            for k = 1 to n_i − 1 do
23:                E[c_ik0] = E[c_ik0] + η^0(i,k)/Prob
24:                E[c_ik1] = E[c_ik1] + η^1(i,k)/Prob
25:            end for
26:        end for
27:        LL = LL + log(Prob)
28:    end for
29:    return LL
30: end function
```

Procedure GetForward, shown in Algorithm 4, computes the value of the forward probabilities. It traverses the diagram one level at a time starting from the root level. For each level it considers each node n and computes its contribution to the forward probabilities of its children. Then the forward probabilities of its children, stored in table F, are updated.

Function GetBackward, shown in Algorithm 5, computes the backward probability of nodes by traversing recursively the tree from the root to the leaves. When the calls of GetBackward for both children of a node n return, we have all the information that is needed to compute the e^x values and the value of $\eta^x(i,k)$ for non-deleted paths. Thus these computations are included in GetBackward, rather than being included in GetOutsideExpe as in [13].

Algorithm 3 Procedure Maximization

```
1: procedure MAXIMIZATION
2:     for all i ∈ Rules do
3:         for k = 1 to n_i − 1 do
4:             π(ik) = E[c_ik1] / (E[c_ik0] + E[c_ik1])
5:         end for
6:     end for
7: end procedure
```

Algorithm 4 Procedure GetForward: computation of the forward probability

```
1: procedure GETFORWARD(root)
2:     F(root) = 1
3:     F(n) = 0 for all nodes
4:     for l = 1 to levels do            ▷ levels is the number of levels of the BDD rooted at root
5:         Nodes(l) = ∅
6:     end for
7:     Nodes(1) = {root}
8:     for l = 1 to levels do
9:         for all node ∈ Nodes(l) do
10:            Let X_ijk be v(node), the variable associated to node
11:            if child_0(node) is not terminal then
12:                F(child_0(node)) = F(child_0(node)) + F(node) · (1 − π_ik)
13:                Add child_0(node) to Nodes(level(child_0(node)))  ▷ level(node) returns the level of node
14:            end if
15:            if child_1(node) is not terminal then
16:                F(child_1(node)) = F(child_1(node)) + F(node) · π_ik
17:                Add child_1(node) to Nodes(level(child_1(node)))
18:            end if
19:        end for
20:    end for
21: end procedure
```

The array ς stores for every level-variable l an algebraic sum of $e^x(n)$: those for nodes in upper levels that do not have a descendant in level l minus those for nodes in upper levels that have a descendant in level l. In this way it is possible to add the contributions of the deleted paths by starting from the root level and accumulating $\varsigma(l)$ for the various levels in a variable T: an $e^x(n)$ value which is added to the accumulator T for level l means that n is an ancestor for nodes in this level. When the x-branch from n reaches a node in a level $l' \leq l$, $e^x(n)$ is subtracted from the accumulator, as it is not relative to a deleted node on the path anymore.

Let us see an example of execution. Suppose you have the program of Example 1 and you have the single example *epidemic*. The BDD of Figure 1(b) (also shown in Figure 3) is built and passed to EXPECTATION in the form of a pointer to its root node n_1. After initializing the η counters to 0, GETFORWARD is called with argument n_1. The F table for n_1 is set to 1 since this is the root. The F is computed for the 0-child, n_2, as $0 + 1 \cdot 0.4 = 0.4$ and n_2 is added to $Nodes(2)$, the set of nodes for the second level. Then F is computed for the 1-child, n_3, as $0 + 1 \cdot 0.6 = 0.6$, and n_3 is added to $Nodes(3)$. At the next iteration of the cycle level 2 is considered and node n_2 is fetched from $Nodes(2)$. The 0-child is a terminal so it is skipped, while the 1-child is n_3 and its F value is updated as $0.6 + 0.4 \cdot 0.6 = 0.84$. In the third iteration node n_3 is

Algorithm 5 Procedure GetBackward: computation of the backward probability, updating of η and of ς

```
1: function GetBackward(node)
2:     if node is a terminal then
3:         return value(node)
4:     else
5:         Let X_ijk be v(node)
6:         B(child_0(node)) =GetBackward(child_0(node))
7:         B(child_1(node)) =GetBackward(child_1(node))
8:         e^0(node) = F(node) · B(child_0(node)) · (1 − π_ik)
9:         e^1(node) = F(node) · B(child_1(node)) · π_ik
10:        η^0(i,k) = η_t^0(i,k) + e^0(node)
11:        η^1(i,k) = η_t^1(i,k) + e^1(node)
12:        VSucc = succ(v(node))          ▷ succ(X) returns the variable following X in the order
13:        ς(VSucc) = ς(VSucc) + e^0(node) + e^1(node)
14:        ς(v(child_0(node))) = ς(v(child_0(node))) − e^0(node)
15:        ς(v(child_1(node))) = ς(v(child_1(node))) − e^1(node)
16:        return B(child_0(node)) · (1 − π_ik) + B(child_1(node)) · π_ik
17:    end if
18: end function
```

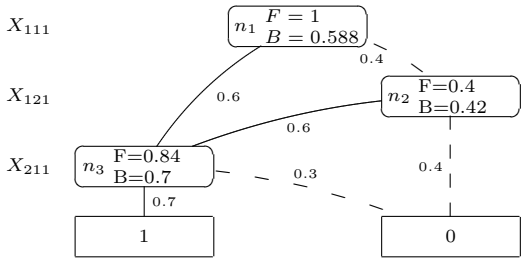


Figure 3: Forward and backward probabilities. F indicates the forward probability and B the backward probability of each node.

fetched but since its children are leaves F is not updated. The resulting forward probabilities are shown in Figure 3.

Then GetBackward is called on n_1.The function calls GetBackward(n_2) that in turn calls GetBackward(0). The latter call returns 0 because it is a terminal node. Then GetBackward(n_2) calls GetBackward(n_3) that in turn calls GetBackward(1) and GetBackward(0), returning respectively 1 and 0. Then GetBackward(n_3) computes $e^0(n_3)$ and $e^1(n_3)$ in the following way:

$e^0(n_3) = F(n_3) \cdot B(0) \cdot (1 - \pi_{21}) = 0.84 \cdot 0 \cdot 0.3 = 0$

$e^1(n_3) = F(n_3) \cdot B(1) \cdot (\pi_{21}) = 0.84 \cdot 1 \cdot 0.7 = 0.588$

where $B(n)$ and $F(n)$ are respectively the backward and forward probabilities of node n. Now the counters for clause C_2 are updated:

$\eta^0(2,1) = 0$

$\eta^1(2,1) = 0.588$

while we do not show the update of ς since its value for the level of the leaves is not used afterwards. GetBackward(n_3) now returns the backward probability of n_3 $B(n_3) = 1 \cdot 0.7 + 0 \cdot 0.3 = 0.7$. GetBackward($n_2$) can proceed to compute

$e^0(n_2) = F(n_2) \cdot B(0) \cdot (1 - \pi_{11}) = 0.4 \cdot 0.0 \cdot 0.4 = 0$

$e^1(n_2) = F(n_2) \cdot B(n_3) \cdot (\pi_{11}) = 0.4 \cdot 0.7 \cdot 0.6 = 0.168$

and $\eta^0(1,1) = 0$, $\eta^1(1,1) = 0.168$. The variable following X_{121} is X_{211} so $\varsigma(X_{211}) = e^0(n_2) + e^1(n_2) = 0 + 0.168 = 0.168$. Since X_{121} is also associated to the 1-child n_2, then $\varsigma(X_{211}) = \varsigma(X_{211}) - e^1(n_2) = 0$. The 0-child is leaf so we do not show the update of ς .

GetBackward(n_2) then returns $B(n_2) = 0.7 \cdot 0.6 + 0 \cdot 0.4 = 0.42$ to GetBackward(n_1) that computes $e^0(n_1)$ and $e^1(n_1)$ as

$e^0(n_1) = F(n_1) \cdot B(n_2) \cdot (1 - \pi_{11}) = 1 \cdot 0.42 \cdot 0.4 = 0.168$

$e^1(n_1) = F(n_1) \cdot B(n_3) \cdot (\pi_{11}) = 1 \cdot 0.7 \cdot 0.6 = 0.42$

and updates the η counters as $\eta^0(1,1) = 0.168$, $\eta^1(1,1) = 0.168 + 0.42 = 0.588$.

Finally ς is updated:

$\varsigma(X_{121}) = e^0(n_1) + e^1(n_1) = 0.168 + 0.42 = 0.588$

$\varsigma(X_{121}) = \varsigma(X_{121}) - e^0(n_1) = 0.42$

$\varsigma(X_{211}) = \varsigma(X_{211}) - e^1(n_1) = -0.42$

GetBackward(n_1) returns $B(n_1) = 0.7 \cdot 0.6 + 0.42 \cdot 0.4 = 0.588$ to Expectation, that adds the contribution of deleted nodes by cycling over the BDD levels and updating T. Initially T is set to 0, then for variable X_{111} T is updated to $T = \varsigma(X_{111}) = 0$ which implies no modification of $\eta^0(1,1)$ and $\eta^1(1,1)$. For variable X_{121} T is updated to $T = 0 + \varsigma(X_{121}) = 0.42$ and the η table is modified as

$\eta^0(1,1) = 0.168 + 0.42 \cdot 0.4 = 0.336$

$\eta^1(1,1) = 0.588 + 0.42 \cdot 0.6 = 0.84$

For variable X_{211} T becomes $0.42 + \varsigma(X_{211}) = 0$ so $\eta^0(2,1)$ and $\eta^0(2,1)$ are not updated. At this point the expected counts for the two rules can be computed:

$\mathbf{E}[c_{110}] = 0 + 0.336/0.588 = 0.5714285714$

$\mathbf{E}[c_{111}] = 0 + 0.84/0.588 = 1.4285714286$

$\mathbf{E}[c_{120}] = 0$

$\mathbf{E}[c_{121}] = 0$

$\mathbf{E}[c_{210}] = 0 + 0/0.588 = 0$

$\mathbf{E}[c_{211}] = 0 + 0.588/0.588 = 1$

4 Experiments

Emblem has been tested over three real world datasets: IMDB[1][20], UW-CSE[2] [33] and Cora[3] [33].

We implemented *Emblem* in Yap Prolog[4] and we compared it with RIB [30]; CEM, an implementation of EM based on the `cplint` inference library [26, 29]; LeProblog [8, 9] and Alchemy [23]. All experiments were performed on Linux machines with an Intel Core 2 Duo E6550 (2333 MHz) processor and 4 GB of RAM.

[1] `http://alchemy.cs.washington.edu/data/imdb`

[2] `http://alchemy.cs.washington.edu/data/uw-cse`

[3] `http://alchemy.cs.washington.edu/data/cora`

[4] `http://www.dcc.fc.up.pt/~vsc/Yap/`

To compare our results with LeProbLog we exploited the translation of LPADs into ProbLog proposed in [4], in which a disjunctive clause with k head atoms and vector of variables $\vec{X}$ is modeled with k ProbLog clauses and $k-1$ probabilistic facts with variables $\vec{X}$.

To compare our results with Alchemy we exploited the translation between LPADs and MLN used in [30] and inspired by the translation between ProbLog and MLN proposed in [9]. An MLN clause is translated into an LPAD clause in which the head atoms of the LPAD clause are the *null* atom plus the positive literals of the MLN clause while the body atoms are the negative literals.

For the probabilistic logic programming systems (*Emblem*, RIB, CEM and LeProbLog) we consider various options. The first consists in choosing between associating a distinct random variable to each grounding of a probabilistic clause or a single random variable to a non-ground probabilistic clause expressing whether the clause is used or not. The latter case makes the problem easier, as stated previously. The second option is concerned with putting a limit on the depth of derivations as done in [9], thus eliminating explanations associated to derivations exceeding the depth limit. This is necessary for problems that contain cyclic clauses, such as transitive closure clauses. The third option involves setting the number of restarts for EM based algorithms.

All experiments for probabilistic logic programming systems have been performed using open-world predicates, meaning that, when resolving a literal, both facts in the database and rules are used to prove it.

IMDB regards movies, actors, directors and movie genres. It is divided into five mega-examples, each containing all the information regarding four movies. We performed training on four mega-examples and test on the remaining one. Then we drew a Precision-Recall curve and computed the Area Under the Curve (AUCPR) using the method reported in [3]. We defined 4 different LPADs, two for predicting the target predicate `sameperson/2`, and two for predicting `samemovie/2`. We had one positive example for each fact that is true in the data, while we sampled from the complete set of false facts three times the number of true instances in order to generate negative examples.

For predicting `sameperson/2` we used the same LPAD of [30]:

```
sameperson(X,Y):p:- movie(M,X),movie(M,Y).
sameperson(X,Y):p:- actor(X),actor(Y),workedunder(X,Z),
                    workedunder(Y,Z).
sameperson(X,Y):p:- gender(X,Z),gender(Y,Z).
sameperson(X,Y):p:- director(X),director(Y),genre(X,Z),
genre(Y,Z).
```

where `p` is a tunable parameter. We ran *Emblem* on it with the following settings: no depth bound, random variables associated to instantiations of the clauses and a number of restarts chosen to match the execution time of *Emblem* with that of the fastest other algorithm.

The queries that LeProbLog take as input are obtained by annotating with 1.0 each positive example for `sameperson/2` and with 0.0 each negative example

for sameperson/2 obtained by random sampling. We ran LeProbLog for a maximum of 100 iterations or until the difference in Mean Squared Error (MSE) between two iterations got smaller than 10^{-5}.

For Alchemy we used the preconditioned rescaled conjugate gradient discriminative algorithm [16] and we specified sameperson/2 as the only non-evidence predicate.

A second LPAD has been created to evaluate the performance of the algorithms when some atoms are unseen:

```
sameperson_pos(X,Y):p:- movie(M,X),movie(M,Y).
sameperson_pos(X,Y):p:- actor(X),actor(Y),
                        workedunder(X,Z),workedunder(Y,Z).
sameperson_pos(X,Y):p:- director(X),director(Y),genre(X,Z),
            genre(Y,Z).
sameperson_neg(X,Y):p:- movie(M,X),movie(M,Y).
sameperson_neg(X,Y):p:- actor(X),actor(Y),
                        workedunder(X,Z),workedunder(Y,Z).
sameperson_neg(X,Y):p:- director(X),director(Y),genre(X,Z),
                              genre(Y,Z).
sameperson(X,Y):p:-\+ sameperson_pos(X,Y),sameperson_neg(X,Y).
sameperson(X,Y):p:-\+ sameperson_pos(X,Y),\+ sameperson_neg(X,Y).
sameperson(X,Y):p:-sameperson_pos(X,Y),sameperson_neg(X,Y).
sameperson(X,Y):p:-sameperson_pos(X,Y),\+ sameperson_neg(X,Y).
```

The sameperson_pos/2 and sameperson_neg/2 predicates are unseen in the data. Settings are the same as the previous LPAD. In this experiment Alchemy was run with the −withEM option that turns on EM learning. The other parameters for Alchemy and for LeProblog are set as before.

Table 1 shows the AUCPR averaged over the five folds for *Emblem*, RIB, LeProbLog, CEM and Alchemy. Results for the two LPADs are shown respectively in the IMDB-SP and IMDBu-SP rows. Table 2 shows the learning times in hours.

For predicting samemovie/2 we used the LPAD:

```
samemovie(X,Y):p:-movie(X,M),movie(Y,M),actor(M).
samemovie(X,Y):p:-movie(X,M),movie(Y,M),director(M).
samemovie(X,Y):p:-movie(X,A),movie(Y,B),actor(A),director(B),
                  workedunder(A,B).
samemovie(X,Y):p:-movie(X,A),movie(Y,B),director(A),director(B),
                  genre(A,G),genre(B,G).
```

To test the behaviour when unseen predicates are present, we transformed the program for samemovie/2 as we did for sameperson/2, thus introducing the unseen predicates samemovie_pos/2 and samemovie_neg/2. We ran *Emblem* on them with no depth bound, one variable for each instantiation of a rule and one random restart. With regard to LeProbLog and Alchemy, we ran them with the same settings as IMDB-SP and IMDBu-SP, by replacing sameperson with samemovie.

Table 1 shows, in the IMDB-SM and IMDBu-SM rows, the average AUCPR for *Emblem*, LeProblog and Alchemy. For RIB and CEM we obtained a lack of memory error (indicated with "me"); table 2 shows the learning times in hours.

The Cora database contains citations to computer science research papers. For each citation we know the title, the authors and the venue, and the words that appear in them. The task is to determine which citations are referring to the same paper, by predicting the predicate `samebib(cit1,cit2)`. The database contains facts for the predicates `sameauthor(aut1,aut2)`, `sametitle(tit1,tit2)`, `samevenue(ven1,ven2)`, `haswordauthor(author,word)`, `haswordtitle(title,word)`, `haswordvenue(venue,word)`.

From the MLN proposed in [34][5] we obtained two LPADs. The first contains 559 rules and differs from the direct translation of the MLN because rules involving words are instantiated with the different constants, only positive literals for the `hasword` predicates are used and transitive rules are not included:

```
samebib(B,C):p:- author(B,D),author(C,E),sameauthor(D,E).
samebib(B,C):p:- title(B,D),title(C,E),sametitle(D,E).
samebib(B,C):p:- venue(B,D),venue(C,E),samevenue(D,E).
samevenue(B,C):p:-haswordvenue(B,word_06),
haswordvenue(C,word_06).
...
sametitle(B,C):p:-haswordtitle(B,word_10),
haswordtitle(C,word_10).
....
sameauthor(B,C):p:-haswordauthor(B,word_a),
haswordauthor(C,word_a).
.....
```

The dots stand for the rules for all the possible words. The Cora dataset comprises five mega-examples each containing facts for the four predicates `samebib/2`, `samevenue/2`, `sametitle/2` and `sameauthor/2`, which have been set as target predicates. We ran *Emblem* on this LPAD with no depth bound, a single variable for each instantiation of a rule and a number of restarts chosen to match the execution time of *Emblem* with that of the fastest other algorithm.

The second LPAD adds to the previous one four transitive rules:

```
samebib(A,B):p :- samebib(A,C), samebib(C,B).
sameauthor(A,B):p :- sameauthor(A,C), sameauthor(C,B).
sametitle(A,B):p :- sametitle(A,C), sametitle(C,B).
samevenue(A,B):p :- samevenue(A,C), samevenue(C,B).
```

for a total of 563 rules. In this case we had to run *Emblem* with a depth bound equal to two and a single variable for each non-ground rule; the number of restarts was one. As for LeProbLog, we separately learned the four predicates because learning the whole theory at once would give a lack of memory error.

[5] Available at `http://alchemy.cs.washington.edu/mlns/er/`.

We annotated with 1.0 each positive example for `samebib/2`, `sameauthor/2`, `sametitle/2`, `samevenue/2` and with 0.0 the negative examples for the same predicates, which were contained in the dataset provided with the MLN. We ran it for a maximum of 100 iterations or until the difference in Mean Squared Error (MSE) between two iterations got smaller than 10^{-5}. As for Alchemy, we used the preconditioned rescaled conjugate gradient discriminative training algorithm [16] to learn weights, by specifying the four predicates as the non-evidence predicates. Table 1 shows in the Cora and CoraT (Cora transitive) rows the average AUCPR obtained by training on four mega-examples and testing on the remaining one. CEM and Alchemy on CoraT gave a memory error while RIB was not applicable because it was not possible to split the input examples into smaller independent interpretations as required by RIB.

The UW-CSE dataset contains information about the computer science department of the University of Washington. It contains 22 different predicates, such as `yearsInProgram/2`, `advisedBy/2`, `taughtBy/3` and so on. The predicates are typed, where possible types are for instance person, course, publication, etc. The database is split into five mega-examples, each containing facts for a particular area of the CS department: AI, graphics, programming languages, systems and theory. The goal here is to predict the `advisedBy/2` predicate, namely the fact that a person is advised by another person: this was our target predicate.

The theory used was obtained from the MLN of [33][6]. It contains 86 rules, such as for instance:

```
advisedby(S, P) :p :- courselevel(C,level_500),taughtby(C,P,Q),
          ta(C, S, Q).
tempadvisedby(S, P) :p :- courselevel(C,level_500),
         taughtby(C, P, Q),  ta(C, S, Q).
professor(P) :p :- courselevel(C,level_500),taughtby(C,P,Q).
```

We ran *Emblem* on it with a single variable for each instantiation of a rule, a depth bound of two and one random restart.

The annotated queries that LeProbLog takes as input have been created by annotating with 1.0 each positive example for `advisedby/2` and with 0.0 the negative examples for `advisedby/2`, obtained by random sampling. The negative examples have been generated by considering all couple of persons `(a,b)` appearing in an `advisedby` fact in the data and adding a negative example `advisedby(a,b)` if it is not in the data. We ran LeProbLog for a maximum of 100 iterations or until the difference in MSE between two iterations got smaller than 10^{-5}. As for Alchemy, we used the preconditioned rescaled conjugate gradient discriminative training algorithm [16] to learn weights, by specifying `advisedby/2` as the only non-evidence predicate. Table 1 shows the area under the precision-recall curve averaged over the five departments for all the algorithms.

[6] Available at `http://alchemy.cs.washington.edu/mlns/uw-cse`.

Table 1: Results of the experiments on all datasets. IMDBu refers to the IMDB dataset with the theory containing unseen predicates. CoraT refers to the theory containing transitive rules. Numbers in parenthesis followed by r mean the number of random restarts (when different from one) to reach the area specified. "me" means memory error during learning. AUCPR is the area under the precision-recall curve averaged over the five folds. R is RIB, L is LeProbLog, C is CEM, A is Alchemy.

Dataset	AUCPR				
	Emblem	R	L	C	A
IMDB-SP	0.202(500r)	0.199	0.096	0.202	0.107
IMDBu-SP	0.175(40r)	0.166	0.134	0.120	0.020
IMDB-SM	1.000	me	0.933	0.537	0.820
IMDBu-SM	1.000	me	0.933	0.515	0.338
Cora	0.995(120r)	0.939	0.905	0.995	0.469
CoraT	0.991	no	0.970	me	me
UW-CSE	0.883	0.588	0.270	0.644	0.294

Table 2: Execution time in hours of the experiments on all datasets. R is RIB, L is LeProbLog, C is CEM and A is Alchemy.

Dataset	Time(h)				
	Emblem	R	L	C	A
IMDB-SP	0.01	0.016	0.35	0.01	1.54
IMDBu-SP	0.01	0.0098	0.23	0.012	1.54
IMDB-SM	0.00036	me	0.005	0.0051	0.0026
IMDBu-SM	3.22	me	0.0121	0.0467	0.0108
Cora	2.48	2.49	13.25	11.95	1.30
CoraT	0.38	no	4.61	me	me
UW-CSE	2.81	0.56	1.49	0.53	1.95

Table 3 shows the p-value of a paired two-tailed t-test at the 5% significance level of the difference in AUCPR between *Emblem* and RIB/LeProbLog/CEM/ Alchemy (significant differences in bold).

From the results we can observe that over IMDB *Emblem* has comparable performances with CEM for IMDB-SP, with similar execution time. On IMDBu-SP it has better performances than all other systems, with a learning time equal to the fastest other algorithm. On IMDB-SM it reaches the highest area value in less time (only one restart is needed). On IMDBu-SM it still reaches the highest area with one restart but with a longer execution time.

Over Cora it has comparable performances with the best other system CEM but in significant lower time and over CoraT is one of the few systems to be able to complete learning, with better performances in terms of area and time.

Over UW-CSE it has significant better performances with respect to all the algorithms.

Table 3: Results of t-test on all datasets. p is the p-value of a paired two-tailed t-test (significant differences at the 5% level in bold) between *Emblem* and all the others. R is RIB, L is LeProbLog, C is CEM, A is Alchemy.

Dataset	p			
	Emblem-R	Emblem-L	Emblem-C	Emblem-A
IMDB-SP	0.2167	**0.0126**	0.3739	**0.0134**
IMDBu-SP	0.1276	0.1995	**0.001**	**4.5234e-005**
IMDB-SM	me	0.3739	**0.0241**	0.1790
IMDBu-SM	me	0.3739	0.2780	**2.2270e-004**
Cora	**0.011**	0.0729	1	**0.0068**
CoraT	no	**0.0464**	me	me
UW-CSE	**0.0054**	**1.5017e-004**	**0.0088**	**4.9921e-004**

5 Related Works

Our work has close connection with various other works. [12, 13] proposed an EM algorithm for learning the parameters of Boolean random variables given observations of a Boolean function over them, represented by a BDD. *Emblem* is an application of that algorithm to probabilistic logic programs. Independently, also [36] proposed an EM algorithm which computes expectations over decision diagrams. The algorithm learns parameters for the CPT-L language, a simple probabilistic logic language for describing sequences of relational states, that is less expressive than LPADs. [11] applies the algorithm of [12, 13] to the problem of computing the probabilistic parameters of abductive explanations. [10] recently presented the CoPrEM algorithm that performs EM for the ProbLog language. We differ from this work in the construction of BDDs: while they build a BDD for an interpretation that represents the application of the whole theory to the interpretation, we focus on a target predicate, the one for which we want to obtain good predictions, and we build BDDs starting from atoms for the target predicate. Moreover, while we compute the contributions of deleted paths with the ς table, CoPrEM treats missing nodes as if they were there and updates the counts accordingly.

Other approaches for learning probabilistic logic programs can be classified into three categories: those that employ constraint techniques, those that use EM and those that adopt gradient descent. In the first class, [24, 25, 27] learn a subclass of ground programs by first finding a large set of clauses satisfying certain constraints and then applying mixed integer linear programming to identify a subset of the clauses that form a solution.

Among the approaches that use EM, [1, 17, 18] first proposed to use the EM algorithm to induce parameters and the Structural EM algorithm to induce ground LPADs structures. Their EM algorithm however works on the underlying Bayesian network.

RIB [30] performs parameter learning using the information bottleneck approach, which is an extension of EM targeted especially towards hidden variables. However, it works better when interpretations have the same Herbrand

base, which is not always the case.

The PRISM system [31, 32] is one of the first learning algorithms based on EM. It exploits Logic Programming techniques for computing expectations but imposes restrictions on the language.

In [15] the authors use EM to learn the structure of first-order rules with associated probabilistic uncertainty parameters. Their approach involves generating the underlying graphical model using a Knowledge-Based Model Construction approach. EM is then applied on the graphical model.

Among the works that use a gradient descent technique, LeProbLog [8, 9] starts from a set of queries annotated with a probability and from a ProbLog program. It tries to find the values of the parameters of the program that minimize the mean squared error of the queries probability. LeProbLog uses Binary Decision Diagrams that represent the queries to compute the gradient.

Alchemy [23] is a state of the art SRL system that offers various tools for inference, weight learning and structure learning of Markov Logic Networks (MLN). [16] discusses how to perform weight learning by applying gradient descent of the conditional likelihood of queries for target predicates. MLN significantly differ from the languages under the distribution semantics since they extends first-order logic by attaching weights to logical formulas, reflecting "how strong" they are. MLN allow the use of logical formulas without syntactic restrictions, but does not allow to exploit logic programming techniques.

6 Conclusions

We have proposed a technique which applies EM algorithm to BDD for learning the parameters of Logic Programs with Annotated Disjunctions. The problem we have faced is, given an LPAD for a domain, efficiently learning correct probabilities (the parameters) for the disjunctive heads of the LPAD clauses. The resulting algorithm *Emblem* returns the parameters that best describe data and can be applied to all languages that are based on the distribution semantics. It exploits the BDDs that are built during inference to efficiently compute the expectation for hidden variables.

We executed the algorithm over the real datasets IMDB, UW-CSE and Cora, and evaluated its performances - together with those of four other probabilistic systems - through the AUCPR. These results show that *Emblem* uses less memory than RIB, CEM and Alchemy, allowing it to solve larger problems. Moreover its speed allows to perform a high number of restarts making it escape local maxima.

In the future we plan to extend *Emblem* for learning the structure of LPADs by using the techniques presented in [6].

References

[1] H. Blockeel and W. Meert. Towards learning non-recursive LPADs by transforming them into Bayesian networks. In Hendrik Blockeel, Jan Ramon, Jude W. Shavlik, and Prasad Tadepalli, editors, *Proceedings of the 17th International Conference on Inductive Logic Programming*, volume 4894 of *LNCS*, pages 94–108. Springer, 2007.

[2] E. Dantsin. Probabilistic logic programs and their semantics. In Andrei Voronkov, editor, *Proceedings of the Russian Conference on Logic Programming*, volume 592 of *LNCS*, pages 152–164. Springer, 1991.

[3] J. Davis and M. Goadrich. The relationship between Precision-Recall and ROC curves. In William W. Cohen and Andrew Moore, editors, *Proceedings of the 23rd International Conference on Machine Learning*, volume 148 of *ACM International Conference Proceeding Series*, pages 233–240. ACM, 2006.

[4] L. De Raedt, B. Demoen, D. Fierens, B. Gutmann, G. Janssens, A. Kimmig, N. Landwehr, T. Mantadelis, W. Meert, R. Rocha, V. Santos Costa, I. Thon, and J. Vennekens. Towards digesting the alphabet-soup of statistical relational learning. In Daniel Roy, John Winn, David McAllester, Vikash Mansinghka, and Joshua Tenenbaum, editors, *Proceedings of the 1st Workshop on Probabilistic Programming: Universal Languages, Systems and Applications, in NIPS*, 2008.

[5] L. De Raedt, A. Kimmig, and H. Toivonen. ProbLog: A probabilistic prolog and its application in link discovery. In Manuela M. Veloso, editor, *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 2462–2467. AAAI Press, 2007.

[6] N. Friedman. The Bayesian structural EM algorithm. In Gregory F. Cooper and Serafín Moral, editors, *Proceedings of the 14th Conference on Uncertainty in Artificial Intelligence*, pages 129–138. Morgan Kaufmann, 1998.

[7] N. Fuhr. Probabilistic datalog: Implementing logical information retrieval for advanced applications. *Journal of the American Society for Information Science*, 51(2):95–110, 2000.

[8] B. Gutmann, A. Kimmig, K. Kersting, and L. De Raedt. Parameter learning in probabilistic databases: A least squares approach. In Walter Daelemans, Bart Goethals, and Katharina Morik, editors, *Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases*, volume 5211 of *LNCS*, pages 473–488. Springer, 2008.

[9] B. Gutmann, A. Kimmig, K. Kersting, and L.De Raedt. Parameter estimation in ProbLog from annotated queries. Technical Report CW 583, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, 2010.

[10] B. Gutmann, I. Thon, and L. De Raedt. Learning the parameters of probabilistic logic programs from interpretations. Technical Report CW 584, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, June 2010.

[11] K. Inoue, T. Sato, M. Ishihata, Y. Kameya, and H. Nabeshima. Evaluating abductive hypotheses using an em algorithm on bdds. In Craig Boutilier, editor, *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI)*, pages 810–815. Morgan Kaufmann Publishers Inc., 2009.

[12] M. Ishihata, Y. Kameya, T. Sato, and S. Minato. Propositionalizing the em algorithm by bdds. In F. Zelezn and N. Lavra, editors, *Late Breaking Papers of the 18th International Conference on Inductive Logic Programming*, pages 44–49, 2008.

[13] M. Ishihata, Y. Kameya, T. Sato, and S. Minato. Propositionalizing the em algorithm by bdds. Technical Report TR08-0004, Dept. of Computer Science, Tokyo Institute of Technology, 2008.

[14] A. Kimmig, V. Santos Costa, R. Rocha, B. Demoen, and L. De Raedt. On the efficient execution of ProbLog programs. In *Proceedings of the 24th International Conference on Logic Programming*, volume 5366 of *LNCS*, pages 175–189. Springer-Verlag, 2008.

[15] D. Koller and A. Pfeffer. Learning probabilities for noisy first-order rules. In *Proceedings of the 15th International Joint Conference on Artifical Intelligence*, volume 2, pages 1316–1321. Morgan Kaufmann, 1997.

[16] D. Lowd and P. Domingos. Efficient weight learning for Markov logic networks. In Joost N. Kok, Jacek Koronacki, Ramon López de Mántaras, Stan Matwin, Dunja Mladenic, and Andrzej Skowron, editors, *Proceedings of the 18th European Conference on Machine Learning*, volume 4702 of *LNCS*, pages 200–211. Springer, 2007.

[17] W. Meert, J. Struyf, and H. Blockeel. Learning ground CP-logic theories by means of bayesian network techniques. In D. Malerba, A. Appice, and M. Ceci, editors, *Proceedings of the 6th International Workshop on Multi-Relational Data Mining*, pages 93–104, 2007.

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

[19] W. Meert, J. Struyf, and H. Blockeel. CP-Logic theory inference with contextual variable elimination and comparison to BDD based inference methods. In *Proceedings of the 19th international conference on Inductive logic programming*, ILP'09, pages 96–109. Springer-Verlag, 2009.

[20] L. Mihalkova and R. J. Mooney. Bottom-up learning of Markov logic network structure. In Zoubin Ghahramani, editor, *Proceedings of the 24th International Conference on Machine Learning*, volume 227 of *ACM International Conference Proceeding Series*, pages 625–632. ACM, 2007.

[21] D. Poole. Logic programming, abduction and probability - a top-down anytime algorithm for estimating prior and posterior probabilities. *New Generation Computing*, 11(3-4):377–400, 1993.

[22] D. Poole. The Independent Choice Logic for modelling multiple agents under uncertainty. *Artificial Intelligence*, 94(1-2):7–56, 1997.

[23] M. Richardson and P. Domingos. Markov logic networks. *Machine Learning*, 62(1-2):107–136, 2006.

[24] F. Riguzzi. Learning logic programs with annotated disjunctions. In Rui Camacho, Ross D. King, and Ashwin Srinivasan, editors, *Proceedings of the 14th International Conference on Inductive Logic Programming*, volume 3194 of *LNCS*, pages 270–287. Springer, 2004.

[25] F. Riguzzi. ALLPAD: Approximate learning of logic programs with annotated disjunctions. In Stephen Muggleton, Ramón P. Otero, and Alireza Tamaddoni-Nezhad, editors, *Proceedings of the 16th International Conference on Inductive Logic Programming*, volume 4455 of *LNCS*, pages 43–45. Springer, 2007.

[26] F. Riguzzi. A top-down interpreter for LPAD and CP-Logic. In Roberto Basili and Maria Teresa Pazienza, editors, *Proceedings of the 10th Congress of the Italian Association for Artificial Intelligence*, volume 4733 of *LNCS*, pages 109–120. Springer, 2007.

[27] F. Riguzzi. ALLPAD: approximate learning of logic programs with annotated disjunctions. *Machine Learning*, 70(2-3):207–223, 2008.

[28] F. Riguzzi. Inference with logic programs with annotated disjunctions under the well-founded semantics. In Maria Garcia de la Banda and Enrico Pontelli, editors, *Proceedings of the 24th International Conference on Logic Programming*, volume 5366 of *LNCS*, pages 667–771. Springer, 2008.

[29] F. Riguzzi. Extended semantics and inference for the Independent Choice Logic. *Logic Journal of the IGPL*, 17(6):589–629, 2009.

[30] F. Riguzzi and N. Di Mauro. Applying the information bottleneck to statistical relational learning. *Machine Learning*, 2011. To appear.

[31] T. Sato. A statistical learning method for logic programs with distribution semantics. In Leon Sterling, editor, *Proceedings of the 12th International Conference on Logic Programming*, pages 715–729. MIT Press, 1995.

[32] T. Sato and Y. Kameya. Parameter learning of logic programs for symbolic-statistical modeling. *Journal of Artificial Intelligence Research*, 15:391–454, 2001.

[33] P. Singla and P. Domingos. Discriminative training of Markov logic networks. In Manuela M. Veloso and Subbarao Kambhampati, editors, *Proceedings of the 20th National Conference on Artificial Intelligence and the 17th Innovative Applications of Artificial Intelligence Conference*, pages 868–873. AAAI Press/The MIT Press, 2005.

[34] P. Singla and P. Domingos. Entity resolution with Markov logic. In *Proceedings of the 6th IEEE International Conference on Data Mining*, pages 572–582. IEEE Computer Society, 2006.

[35] A. Thayse, M. Davio, and J. P. Deschamps. Optimization of multivalued decision algorithms. In *International Symposium on Multiple-Valued Logic*, pages 171–178. IEEE Computer Society Press, 1978.

[36] I. Thon, N. Landwehr, and L. De Raedt. A simple model for sequences of relational state descriptions. In Walter Daelemans, Bart Goethals, and Katharina Morik, editors, *Proceedings of the European conference on Machine Learning and Knowledge Discovery in Databases (ECML/PKDD 2008)- Part II*, volume 5212 of *Lecture Notes in Computer Science*, pages 506–521. Springer-Verlag, 2008.

[37] A. Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.

[38] J. Vennekens, M. Denecker, and M. Bruynooghe. Cp-logic: A language of causal probabilistic events and its relation to logic programming. *Theory and Practice of Logic Programming*, 9(3):245–308, 2009.

[39] J. Vennekens and S. Verbaeten. Logic programs with annotated disjunctions. Technical Report CW386, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, 2003.

[40] J. Vennekens, S. Verbaeten, and M. Bruynooghe. Logic programs with annotated disjunctions. In Bart Demoen and Vladimir Lifschitz, editors, *Proceedings of the 20th International Conference on Logic Programming*, volume 3131 of *LNCS*, pages 195–209. Springer, 2004.