Parameter Learning in Probabilistic Databases: A Least Squares Approach

Bernd Gutmann[1], Angelika Kimmig[1], Kristian Kersting[2], and Luc De Raedt[1]

[1] Dept. of Computer Science, Katholieke Universiteit Leuven, Celestijnenlaan 200A, POBox 2402, BE-3001 Heverlee, Belgium
{firstname.lastname}@cs.kuleuven.be
[2] Fraunhofer IAIS, Schloß Birlinghoven, 53754 Sankt Augustin, Germany
kristian.kersting@iais.fraunhofer.de

Abstract. We introduce the problem of learning the parameters of the probabilistic database ProbLog. Given the observed success probabilities of a set of queries, we compute the probabilities attached to facts that have a low approximation error on the training examples as well as on unseen examples. Assuming Gaussian error terms on the observed success probabilities, this naturally leads to a least squares optimization problem. Our approach, called LeProbLog, is able to learn both from queries and from proofs and even from both simultaneously. This makes it flexible and allows faster training in domains where the proofs are available. Experiments on real world data show the usefulness and effectiveness of this least squares calibration of probabilistic databases.

1 Introduction

Many real-world application today depend on managing enormous volumes of uncertain data. Such "dirty" databases arise for example when integrating data from various sources, when analyzing social, biological, and chemical networks, within privacy-preserving data mining where only aggregated data is available, and within pervasive computing. These are only some of the many real-world applications showing the abundance of uncertain data and the need for probabilistic databases, i.e., generalizations of traditional relational databases that can deal with uncertainty.

Over the last years, the statistical relational learning community has devoted a lot of attention to learning both the structure and parameters of probabilistic logics, cf. [1,2], but so far seems to have devoted little attention to the learning of probabilistic database formalisms. Probabilistic databases [3,4] associate probabilities to facts, indicating the probabilities with which these facts hold. This information is then used to define and compute the success probability of queries or derived facts or tuples, which are defined using background knowledge (in the form of predicate definitions). As one example, imagine a life scientist mining a large network of biological entities in an interactive querying session. The biological network is a probabilistic graph, in which the edges are represented by probabilistic facts about the biological entities [4]. Interesting questions can

W. Daelemans et al. (Eds.): ECML PKDD 2008, Part I, LNAI 5211, pp. 473–488, 2008.

then be asked about the probability of the existence of *a* connection between two nodes, or the most reliable path between them.

The key contribution of the present paper is the introduction of a novel setting for learning the parameters of a probabilistic database from examples together with their target probability. The task then is to find those parameters that minimize the least squared error w.r.t. these examples. The examples themselves can either be queries or proofs, where a proof is a conjunction of all facts in the database needed to proof a query by SLD-resolution. This learning setting is then incorporated in the probabilistic database ProbLog [4], though it can easily be integrated in other probabilistic databases as well. This yields the second key contribution of the paper, namely an effective learning algorithm called *LeProbLog*[1]. It performs gradient-based optimization utilizing advanced data-structures for efficiently computing the gradient. This efficient computation of the gradient allows us to estimate a ProbLog program from a large real-world network of biological entities in our experiments, which can then be used for example by a life scientist in interactive querying sessions.

We proceed as follows. After reviewing related work in Section 2 and ProbLog in Section 3, we will formally introduce the parameter estimation problem for probabilistic databases in Section 4. Section 5 will then present our least-squares approach LeProbLog for solving it. Before concluding, we will present the results of an extensive set of experiments on a real-world data set.

2 Related Work

The probabilistic database setting differs from the usual statistical relational learning approach in that there is no underlying generative model. Indeed, consider for instance the learning of stochastic logic programs (SLPs) [5], PRISM programs [6], probabilistic relational models (PRMs) [7] or Bayesian logic programs (BLPs) [8]. In all these approaches, a generative model is assumed. For SLPs (and stochastic context-free grammars) as well as for PRISM, the learning procedure assumes that ground atoms for a *single* predicate (or in the grammar case, sentences belonging to the language) are sampled and that the sum of the probabilities of all different atoms obtainable in this way is at most 1. Recently, Chen *et al.* [9] also proposed a learning setting similar to ours. The probabilities associated with examples, however, are viewed as specifying the degree of being sampled from some distribution specified by a generative model, which does not hold in our case. Furthermore, they only provide an algorithm for learning from probabilistic facts and not queries and proofs as we do. PRMs and BLPs are relational extensions of Bayesian networks using entity relationship models or logic programming respectively. In both frameworks, possible worlds, i.e. interpretations, are sampled, and the sum of the probabilities of such worlds is 1. Consider now learning in the probabilistic network sketched above. It is unclear how different paths could be sampled and, clearly, the sum of the probabilities

[1] French for Least square estimation for ProbLog.

of such paths need not be equal to 1. These difficulties explain – in part – why so far only few learning techniques for probabilistic databases have been developed.

The learning setting, however, is in line with the general theory of probabilistic logic learning [10] and inductive logic programming. From an inductive logic programming perspective, a query corresponds to a formula that is entailed by the database, and hence, queries correspond to well-known learning from entailment setting. On the other hand, a proof does not only show *what* was proven but also *how* this was realized. An analogy with a probabilistic context-free grammar is useful here. One can learn the parameters of such a grammar starting from sentences belonging to the grammar (learning from entailment / from queries), or alternatively, one could learn it from parse-trees (learning from proofs), cf. the work on tree-bank grammars [11,12]. The former setting is typically a lot harder than the later one because one query may have multiple proofs, which introduces hidden parameters into the learning setting, which are not present when learning from parse-trees. In the present paper, both types of examples can be combined, and as far as the authors are aware, it is the first time within relational learning and inductive logic programming that learning from proofs is integrated with learning from entailment.

Within the probabilistic database community, parameter estimation has received surprisingly few attention. Nottelmann and Fuhr [13] consider learning probabilistic Datalog rules in a similar setting where the underlying distribution semantics is similar to ProbLog. However, their setting and approach also significantly differ from ours. First, a single probabilistic target predicate only is estimated whereas we consider estimating the probabilities attached to definitions of multiple predicates. Second, their approach employs the training probabilities only. Specifically, they generate training examples labeled with 0/1 randomly according to the observed probabilities whereas we use the observed probabilities directly. Finally, whereas LeProbLog follows a principled gradient approach employing (in principle) all combinations of proofs or explanations, they follow a two-steps bootstrapping approach first estimating parameters as empirical frequencies among matching rules and then selecting the subset of rules with the lowest expected quadratic loss on an hold-out validation set. Gupta and Sarawagi [14] also consider a closely related learning setting but only extract probabilistic facts from data.

Finally, the new setting and algorithm compromise a natural and interesting addition to the existing learning algorithms for ProbLog. It is most closely related to the theory compression setting of [15]. There the task was to remove all but the k best facts from the database (that is to set the probability of such facts to 0), which realizes an elementary form of theory revision. The present task extends the compression setting in that parameters of *all* facts can now be *tuned* starting from evidence. This realizes a more general form of theory revision [16], albeit that only the parameters are changed and not the structure.

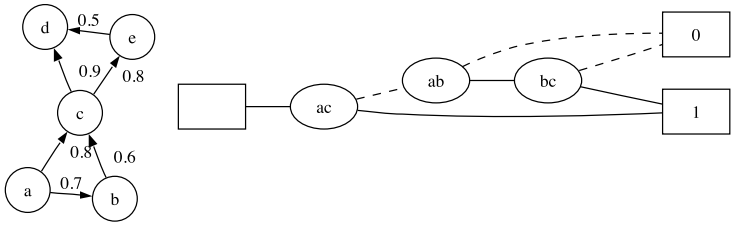

Fig. 1. (a) Example of a probabilistic graph, where edge labels indicate the probability that the edge is part of the graph. (b) Binary Decision Diagram encoding the DNF formula $ac \vee (ab \wedge bc)$, corresponding to the two proofs of query *path(a,c)* in the graph. An internal node labeled xy represents the Boolean variable for the edge between x and y, solid/dashed edges correspond to values true/false.

3 ProbLog

As one example of a probabilistic database, we employ ProbLog, a simple probabilistic extension of Prolog introduced in [4]. Alternatively, the database formalism of [3] or [13] could be used. A ProbLog program consists – as Prolog – of a set of definite clauses. However, in ProbLog every fact c_i is labeled with the probability p_i that its instances $c_i\theta$ are true. It is also assumed that the probabilities of each ground instance $c_i\theta$ (that is, each instance not containing variables) are assumed to be mutually independent. In the following we repeat the main ideas of ProbLog, see [4] for a more detailed explanation.

For ease of illustration, we will consider probabilistic graphs encoded in ProbLog, but the entire discussion carries over to arbitrary ProbLog programs. Figure 1(a) shows a small example that can be encoded in ProbLog as follows:

$0.8 : \texttt{edge(a,c)}.$ $\quad 0.7 : \texttt{edge(a,b)}.$ $\quad 0.8 : \texttt{edge(c,e)}.$
$0.6 : \texttt{edge(b,c)}.$ $\quad 0.9 : \texttt{edge(c,d)}.$ $\quad 0.5 : \texttt{edge(e,d)}.$

It is straightforward to sample subgraphs of a probabilistic graph by tossing a biased coin for each edge. The corresponding ProbLog program $T = \{p_1 : c_1, \cdots, p_n : c_n\}$ therefore defines a probability distribution over subgraphs $L \subseteq L_T = \{c_1, \cdots, c_n\}$ in the following way:

$$P(L|T) = \prod_{c_i \in L} p_i \prod_{c_i \in L_T \setminus L} (1 - p_i).$$

It is straightforward to add background knowledge in the form of Prolog clauses, say, the definition of a path by combining edges. We can then ask for the probability that there exists e.g. a path between nodes a and c in our probabilistic graph, i.e. the probability that a randomly sampled subgraph contains the edge from a to c, or the path from a to c via b (or both of them). Formally, the *success probability* $P_s(q|T)$ of a query q in a ProbLog program T is defined as

$$P_s(q|T) = \sum_{L \subseteq L_T} P(q|L) \cdot P(L|T)\,, \tag{1}$$

where $P(q|L) = 1$ if there exists a θ such that $L \models q\theta$, and $P(q|L) = 0$ otherwise. In other words, the success probability of query q corresponds to the probability that the query q is *provable* in a randomly sampled logic program.

As a consequence, the probability of a *specific* proof, also called explanation, corresponds to that of sampling a logic program L that contains all the facts needed in that explanation or proof. The *explanation probability* $P_x(q|T)$ is then defined as the probability of the most likely explanation or proof of the query q:

$$P_x(q|T) = \max_{e \in E(q)} P(e|T) = \max_{e \in E(q)} \prod_{c_i \in e} p_i \tag{2}$$

where $E(q)$ is the set of all explanations for query q [17].

For our example graph and query *path(a,c)*, the set of all explanations contains the edge from a to c (with probability 0.8) as well as the path consisting of the edges from a to b and from b to c (with probability $0.7 \cdot 0.6 = 0.42$). Thus, $P_x(path(a,c)|T) = 0.8$.

Calculating the explanation probability can easily be realized using a best-first search – guided by the probability of the current derivation – through standard logic programming techniques based on the SLD-tree [18]. On the other hand, evaluating the success probability of ProbLog queries is computationally hard, as different proofs of a query are not independent in general. As shown in [4], the problem can be tackled by reducing the problem to that of computing the probability of a monotone DNF formula, an NP-complete problem.

$$P_s(q|T) = P\Big(\bigvee_{e \in E(q)} \bigwedge_{a_i \in cl(e)} a_i\Big) \tag{3}$$

This DNF formula describes each proof in $E(q)$ as a conjunction of Boolean variables, and the entire set as disjunction of these conjunctions. The formula corresponding to our example query *path(a,c)* is $ac \vee (ab \wedge bc)$, where we use xy as Boolean variable representing edge(x,y). To effectively calculate the probability of such a monotone DNF formula, we employ Binary Decision Diagrams (BDDs) [19], an efficient graphical representation of a Boolean function over a set of variables, see Section 6 for more details.

As the size of the DNF formula grows with the number of proofs, its evaluation can become expensive. For instance, when searching for paths in graphs or networks, even in small networks with a few dozen edges there are easily $O(10^6)$ possible paths between two nodes. In [4], an approximation algorithm is proposed that computes both an upper and a lower bound on the probability of a query and searches for more explanations until the difference between the upper and the lower bound becomes sufficiently small.

When learning parameters, we will have to repeatedly evaluate BDDs for all examples. In this context, using a fixed number of proofs allows better control of the overall complexity. We therefore introduce the k-probability $P_k(q|T)$, which approximates the success probability by using the k best (that is, most likely) explanations instead of all proofs when building the DNF formula used in Equation 3:

$$P_k(q|T) = P\Big(\bigvee_{e \in E_k(q)} \bigwedge_{a_i \in cl(e)} a_i\Big) \tag{4}$$

where $E_k(q) = \{e \in E(q) | P_x(e) \geq P_x(e_k)\}$ with e_k the kth element of $E(q)$ sorted by non-increasing probability. Setting $k = \infty$ and $k = 1$ leads to the success and the explanation probability respectively. Using $k = 1$ in parameter learning has also been called Viterbi learning. Finding the k best proofs can be realized using a simple branch-and-bound approach (cf. also [20]).

To illustrate k-probability, we consider again our example graph, but this time with query *path(a,d)*. This query has four proofs, represented by the conjunctions $ac \wedge cd$, $ab \wedge bc \wedge cd$, $ac \wedge ce \wedge ed$ and $ab \wedge bc \wedge ce \wedge ed$, with probabilities 0.72, 0.378, 0.32 and 0.168 respectively. As P_1 corresponds to the explanation probability P_x, we obtain $P_1(path(a,d)) = 0.72$. For $k = 2$, overlap between the best two proofs has to be taken into account: the second proof only adds information if the first one is disconnected. As they share edge *cd*, this means that edge *ac* has to be missing, leading to $P_2(path(a,d)) = P((ac \wedge cd) \vee (\neg ac \wedge ab \wedge bc \wedge cd)) = 0.72 + (1 - 0.8) \cdot 0.378 = 0.7956$. Similarly, we obtain $P_3(path(a,d)) = 0.8276$ and $P_k(path(a,d)) = 0.83096$ for $k \geq 4$. For reasons of memory-efficiency, the implementation used in our experiments below employs iterative deepening for the calculation of lower and upper bounds as well as for P_k with finite k.

4 Parameter Learning in Probabilistic Databases

When new data is added to a database, there is often uncertainty about the data. Text extraction algorithms return the confidence, experimental data is averaged over several runs and so on. Consider for instance populating a probabilistic database of genes from MEDLINE [2] abstracts using off-the-shelves information extraction tools. For example, we could extract that gene *a* is located in region *b* and interacting with *c*. State-of-the art extraction tools, however, often additionally provide a sound probability distribution over the possible outcomes. Hence, we should deal with weighted examples such as 0.6:locatedIn(a,b) and 0.7:interacting(a,c) as already argued e.g. by Gupta and Sarawagi [14] and Chen *et al.* [9]. The situation fits the general learning setting stated in [21]:

> Given is a set of examples E, a probabilistic coverage relation $P(e|D)$ that denotes the probability that the database D covers the example $e \in E$, a theory T in a probabilistic logic, and a scoring function *score*. The goal is to find parameters of T such that the *score* function yields an optimal value.

Concretely instantiating this definition to ProbLog requires us to determine what the examples will be, which probabilistic coverage relation shall be employed, and also determining the scoring function to be optimized. We shall address each of these in turn.

In probabilistic inductive logic programming, examples can be available in different forms. [21] show how one can learn from entailment, from proofs and from interpretations. When learning from entailment, examples are atoms or

[2] `http://medline.cos.com/`

clauses that are logically entailed by a theory, and in the case of a probabilistic logic, assigned a non-zero probability value. Transforming this setting to ProbLog leads to examples that are logical queries. When learning from proofs, examples are proofs, which correspond to concrete explanations in the ProbLog setting. Learning from interpretation in the ProbLog setting is less natural because it requires interpretations containing all the facts that logically follow from the theory. On the other hand, the former two settings can easily be incorporated and actually integrated in ProbLog. The reason is that the logical form of the example will be translated to the monotone DNF formula and it is this last form that is employed by the learning algorithm anyway. The key difference between learning from entailment and learning from proofs in ProbLog is that the DNF formula is a conjunction when learning from proofs and a more general DNF formula when learning from queries. So, using the query *path(a,c)* as example results in $ac \vee (ab \wedge bc)$, whereas the explanation *edge(a,b),edge(b,c)* results in $ab \wedge bc$ only. To the best of our knowledge, this is the first time that an integrated learning from proofs / entailment setting is considered within (probabilistic) inductive logic programming.

Before determining the scoring function and learning setting, it is important to realize that there is also a major difference between probabilistic databases and alternative probabilistic logics, such as PRISM [6] and SLPs [5], even though the probabilistic database semantics seems closely related at first sight. To see this, assume that we now want to estimate the parameters of a ProbLog program starting from example queries, possibly together with their target probability. Continuing our illustration, assume that we are given a number of path queries together with their true probabilities. It is important to observe that the probabilistic database model does not provide a generative model for sampling such queries because the sum of the probabilities of all path queries is not equal to 1 (and in general will be a lot higher). Therefore, we cannot directly apply standard maximum likelihood techniques for parameter estimation based on the EM algorithm as is usually done for statistical relational learning models [1]. The learning mechanisms developed for both PRISM and SLPs assume that there is a generative model from which the examples (ground atoms for a single predicate) can be sampled and the probability mass associated to the set of all examples is maximum 1. This observation explains also why we consider a different setting for probabilistic databases, in which parameter learning is viewed as a function optimization problem. The problem then is that we seek a set of parameters that approximates the actual query probabilities as close as possible, which in turn explains why rather than maximizing the likelihood of the data, we shall minimize the least squared error between the target probabilities of the examples and the probability of the model, but see below.

Finally, let us remark that the choice of probabilistic coverage relation is open, and that therefore, within ProbLog we choose the k-probability as this allows for maximal flexibility. By now we can instantiate the above definition to obtain the problem-setting tackled in this paper:

Definition 1 (Parameter Learning in Probabilistic Databases). ***Given*** *a set of training examples* $\{q_i, \tilde{p}_i\}_{i=1}^{M}$*,* $M > 0$*, where each* $q_i \in \mathcal{H}$ *is a query or proof and* $\tilde{p}_i$ *is the k-probability of* q_i*,* ***find*** *a function* $h : \mathcal{H} \to [0, 1]$ *with low approximation error on the training examples as well as on unseen examples.* $\mathcal{H}$ *comprises all parameter assignments for a given database* T*.*

This framework allows to naturally combine *learning from entailment* and *learning from proofs*, two learning settings that so far have been considered separately. In ProbLog, proofs correspond to conjunctions of probabilistic facts, and can be seen as a conjunction of queries. Therefore, a learning algorithm can use examples of both forms, (atomic) queries and proofs, at the same time. To realize *learning from interpretations*, probability estimates could be obtained using simple counting. However, this is infeasible for domains where interpretations contain high fractions of facts assigned value true. Finally, the *error function* that we want to minimize is the mean squared error:

$$MSE(T) = \frac{1}{M} \sum\nolimits_{1 \leq i \leq M} \left(P_s(q_i|T) - \tilde{p}_i \right)^2 . \tag{5}$$

It is easy to see that minimizing the squared error corresponds to finding a maximum likelihood hypothesis, provided that each training example $(q_i, \tilde{p}_i)$ is disturbed by a Gaussian error $\tilde{p}_i$, i.e. $\tilde{p}_i = p(q_i) + e_i$, with $p(q_i)$ the actual probability of query q_i and e_i drawn independently from a zero-mean Gaussian. See [22, Chapter 6.4] for a detailed derivation.

Gradient descent is a standard way of minimizing a given error function. The tunable parameters are initialized randomly. Then, as long as the error did not converge, the gradient of the error function is calculated, scaled by the learning rate η, and subtracted from the current parameters. In the following sections, we derive the gradient of the MSE and show how it can be computed efficiently.

5 Gradient of the Mean Squared Error

Applying the sum and chain rule to Eq. (5) yields the partial derivative

$$\frac{\partial MSE(T)}{\partial p_j} = \frac{2}{M} \sum_{1 \leq i \leq M} \underbrace{\left(P_s(q_i|T) - \tilde{p}_i \right)}_{\text{Part 1}} \cdot \underbrace{\frac{\partial\ P_s(q_i|T)}{\partial p_j}}_{\text{Part 2}} . \tag{6}$$

Part 1 can be calculated by a ProbLog inference call computing (1). It does not depend on j and has to be calculated only once in every iteration of a gradient descent algorithm. Part 2 can be calculated as following

$$\frac{\partial P_s(q_i|T)}{\partial p_j} = \sum_{\substack{S \subseteq L_T \\ S \models q_i}} \delta_{jS} \prod_{\substack{c_x \in S \\ x \neq j}} p_x \prod_{\substack{c_x \in L_T \setminus S \\ x \neq j}} (1 - p_x) , \tag{7}$$

where $\delta_{jS} := 1$ if $c_j \in S$ and $\delta_{jS} := -1$ if $c_j \in L_T \setminus S$. It is derived by first deriving the gradient $\partial P(S|T)/\partial p_j$ for a fixed subset $S \subseteq L_T$ of facts, which is straight-forward, and then summing over all subsets S where q_i can be proven.

Algorithm 1. Evaluating the gradient of a query efficiently by traversing the corresponding BDD, calculating partial sums, and adding only relevant ones

function GRADIENT(BDD b, fact to derive for n_j)
  $(val, seen) =$ GRADIENTEVAL$(root(b), n_j)$
  If $seen = 1$ return $val \cdot \sigma(a_j) \cdot (1 - \sigma(a_j))$
  Else return 0
function GRADIENTEVAL(node n, target node n_j)
  If n is the 1-terminal return $(1, 0)$
  If n is the 0-terminal return $(0, 0)$
  Let h and l be the high and low children of n
  $(val(h), seen(h)) =$ GRADIENTEVAL(h, n_j)
  $(val(l), seen(l)) =$ GRADIENTEVAL(l, n_j)
  If $n = n_j$ return $(val(h) - val(l), 1)$
  ElseIf $seen(h) = seen(l)$ return $(\sigma(a_n) \cdot val(h) + (1 - \sigma(a_n)) \cdot val(l), seen(h)))$
  ElseIf $seen(h) = 1$ return $(\sigma(a_n) \cdot val(h), 1)$
  ElseIf $seen(l) = 1$ return $((1 - \sigma(a_n)) \cdot val(l), 1)$

To ensure that all p_j stay probabilities during gradient descent, we reparameterize the search space and express each $p_j \in]0, 1[$ in terms of the sigmoid function[3] $p_j = \sigma(a_j) := 1/(1 + \exp(-a_j))$ applied to $a_j \in \mathbb{R}$. This technique has been used for Bayesian networks and in particular for sigmoid belief networks [23]. We derive the partial derivative $\partial P_s(q_i|T)/\partial a_j$ in the same way as (7) but we have to apply the chain rule one more time due to the σ function

$$\sigma(a_j) \cdot (1 - \sigma(a_j)) \cdot \sum_{\substack{S \subseteq L_T \\ L \models q_i}} \delta_{jS} \prod_{\substack{c_x \in S \\ x \neq j}} \sigma(a_x) \prod_{\substack{c_x \in L_T \setminus S \\ x \neq j}} (1 - \sigma(a_x)).$$

We also have to replace every p_j in Eq. (1) by $\sigma(p_j)$. Going over all subprograms S in the last equation is infeasible. But there is an efficient algorithm to compute $P_s(q_i|T)$ relying on BDDs [4]. In the following section we update this towards the gradient and introduce LeProbLog, the gradient descent algorithm for ProbLog.

6 LeProbLog

To compute the success probability P_∞ for a query q efficiently, De Raedt *et al.* [4] collect all proofs and compactly represent them in a Binary Decision Diagram (BDD) [19]. BDDs are one of the best understood data structures today. They have been used to solve a wide variety of computer science problems. Given a fixed variable ordering, a Boolean function f can be represented as a full Boolean decision tree where each node on the ith level is labeled with the ith variable

[3] The sigmoid function can induce plateaus which might slow down a gradient-based search. However, it is unlikely that a plateau will spread out over several dimensions and we did not observe such a behavior in our experiments. If it happens though, one can take standard counter measures like simulated annealing or random restarts.

Algorithm 2. LeProbLog, the algorithm takes a program without probabilities as input, minimizes the MSE on the training set by gradient descent and returns a ProbLog program with probabilities

Require: a ProbLog program without probabilities L_T
training set $q_j, \tilde{p}_j$ $1 \leq j \leq M$
learning rate η
k, the number of proofs used to generate the BDDs
Ensure: parameters p_i $1 \leq i \leq n$
initialize all a_j randomly
while not converged **do**
$\Delta\mathbf{a} := \mathbf{0}$
for $1 \leq i \leq M$ **do**
find k best proofs and generate BDD_i for q_i
$y := \frac{2}{M} \cdot \left(P_s(q_i|T) - \tilde{p}_i\right)$
for $1 \leq j \leq n$ **do**
$\Delta a_j := \Delta a_j + y \cdot \frac{\partial P_s(q_i|T)}{\partial a_j}$ {call GRADIENT$(BDD_i, node_j)$}
$\mathbf{a} := \mathbf{a} - \eta \cdot \Delta\mathbf{a}$
return T, that is $\{\sigma(a_j) : c_j \mid c_j \in L_T\}$ {A ProbLog program with probabilities}

and has two children called low and high. Leaves are labeled by the outcome of f for the variable assignment corresponding to the path to the leaf, where in each node labeled x, the branch to the low (high) child is taken if variable x is assigned 0 (1). Starting from such a tree, one obtains a BDD by merging isomorphic subgraphs and deleting redundant nodes until no further reduction is possible. A node is redundant if the subgraphs rooted at its children are isomorphic. Figure 1(b) shows the BDD for the existence of a path between a and c in our earlier example.

The algorithm of De Raedt *et al.* [4] calculates the probability of a Boolean formula by traversing the BDD bottom-up, in each node summing the probability of the high and low child, weighted by the probability of the node's variable being assigned true and false respectively. We extended this to the computation of the gradient (7). Both algorithms have a time and space complexity of O(number of node in the BDD) when intermediate results are cached.

Let us first consider a full decision tree instead of a BDD. Each branch in the tree represents a product $n_1 \cdot n_2 \cdot \ldots \cdot n_i$, where the n_i are the probabilities associated to the corresponding variable assignment of nodes on the branch. The gradient of such a branch b with respect to n_j is $g_b = n_1 \cdot n_2 \cdot \ldots n_{j-1} \cdot n_{j+1} \cdot \ldots \cdot n_i$ if n_j is true, and $-g_b$ if n_j is false in b. As all branches in a full decision tree are mutually exclusive, the gradient w.r.t. n_j can be obtained by simply summing the gradients of all branches ending in a leaf labeled 1. In BDDs however, isomorphic sub-parts are merged, and obsolete parts are left out. This implies that some paths from the root to the 1-terminal may not contain n_j, therefore having a gradient of 0. So, when calculating the gradient on the BDD, we have to keep track of whether n_j appeared on a path or not. Given that the variable order is the same on all paths, we can easily propagate this information in our bottom-up algorithm. This is exactly what is described in Algorithm 1. Specifically,

GradientEval(n, n_j) calculates the gradient w.r.t. n_j in the sub-BDD rooted at n. It returns two values: the gradient on the sub-BDD and a Boolean indicating whether or not the target node n_j appears in the sub-BDD. When at some node n the indicator values for the two children differ, we know that n_j does not appear above the current node, and we can drop the partial result from the child with indicator 0. The indicator variable is also used on the top level: Gradient returns the value calculated by the bottom-up algorithm if n_j occurred in the BDD and 0 otherwise.

LeProbLog combines the BDD-based gradient calculation with a standard gradient descent search. Starting from parameters $\mathbf{a} = a_1, \ldots, a_n$ initialized randomly, the gradient $\Delta\mathbf{a} = \Delta a_1, \ldots, \Delta a_n$ is calculated, parameters are updated by subtracting the gradient, and updating is repeated until convergence. When using the k-probability with finite k, the set of k best proofs may change due to parameter updates. After each update, we therefore recompute the set of proofs and the corresponding BDD. Algorithm 2 shows the pseudocode of LeProbLog.

7 Experiments

We set up experiments to investigate the following questions:

Q1 Does our approach reduce the mean squared error on training and test data?
Q2 Is our approach able to recover the original parameters?

Answering these first questions will serve as a sanity check for the algorithm and our implementation.

Q3 Is it necessary to update the set of k best proofs in each iteration?

As building BDDs for all examples is expensive, building BDDs once and using them during the entire learning process can save significant amounts of resources and time. We are therefore interested in the effects this strategy has on the results.

Q4 Can we obtain good results approximating P_∞ by P_k for finite (small) k?

Given that using BDDs to calculate P_∞ is infeasible for huge sets of proofs, as they occur in our application, where we easily get hundreds of thousands of proofs, we are interested in fast, reliable approximations.

Q5 Do the results improve when parts of the training examples are given as proof?

Here we are interested in exploring the effects of providing more information in the form of proofs, which is one of the main distinguishing features of LeProbLog.

To answer these questions, we extracted graphs around both Alzheimer disease and asthma from a collection of databases. For each disease, we obtained a set of related genes by searching Entrez for human genes with the relevant annotation (AD or asthma); corresponding phenotypes for the diseases are from OMIM. Most of the other information comes from EntrezGene, String, UniProt, HomoloGene, Gene Ontology, and OMIM databases. Weights were assigned to

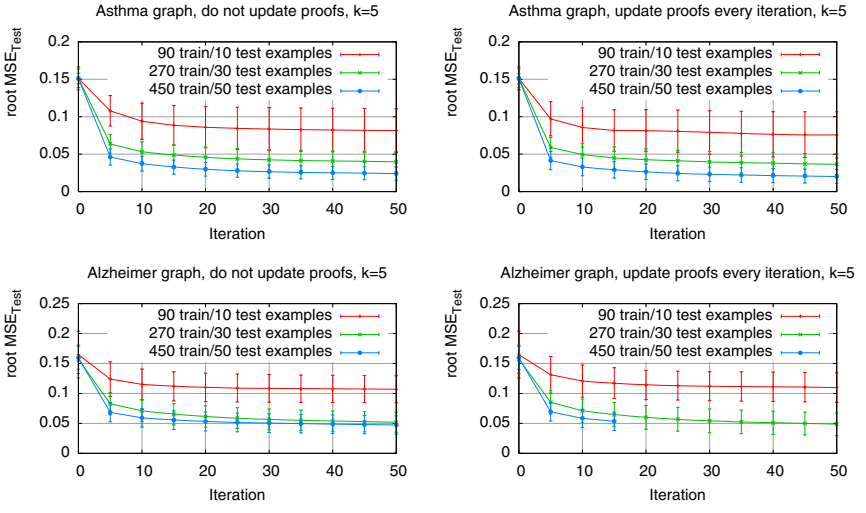

Fig. 2. $\sqrt{MSE_{\text{Test}}}$ for asthma and Alzheimer using the 5 best proofs ($k = 5$); when the BDDs and proofs are not updated (left column); when they are updated every iteration (right column) (**Q2** and **Q3**)

edges as described in [24]. In the experiments below, we used a fixed number of randomly chosen (Alzheimer disease or asthma) genes for graph extraction. Subgraphs were extracted by taking all acyclic paths of no more than length 4, with a probability of at least 0.01, between any given gene and the corresponding phenotype. Some of the genes did not have any such paths to the phenotype and are thus disconnected from the rest of the graph. The resulting graph around Alzheimer contains 122 nodes and 259 edges, that around Asthma 127 nodes and 241 edges. From these graphs we generated 3 sorts of training sets:

1. We sampled 500 random node pairs from the asthma and Alzheimer graph and estimate the query probability for path(a,b) using P_5, the probability of the 5 best proofs. These two sets are used to answer **Q1**, **Q2**, and **Q3**.
2. We sampled 200 random node pairs from the asthma graph and estimated $P_\infty(path(a,b))$ using the lower bound of the approximative inference algorithm [4] with interval width $\delta = 0.01$. This set is used to answer **Q4**.
3. We sampled 300 random node pairs and calculated P_1 for path(a,b), the probability of the best path between a and b. We then build several sets where different fractions of the examples where given as proof, the edges of the best path, instead of the path(a,b) query, and used them to answer **Q5**.

To assess results, we use the root mean squared error on the test data $\sqrt{MSE_{\text{test}}}$, and the mean absolute difference MAD_{facts} between learned p_j and original fact probabilities p_j^{true}: $MAD_{\text{facts}} := n^{-1} \sum_{j=1}^{n} |p_j - p_j^{\text{true}}|$. We always sampled the initial fact parameters uniformly in the interval $[-0.5, 0.5]$. Applying the sigmoid function yields probability values with mean 0.5 ± 0.07. The datasets used, had

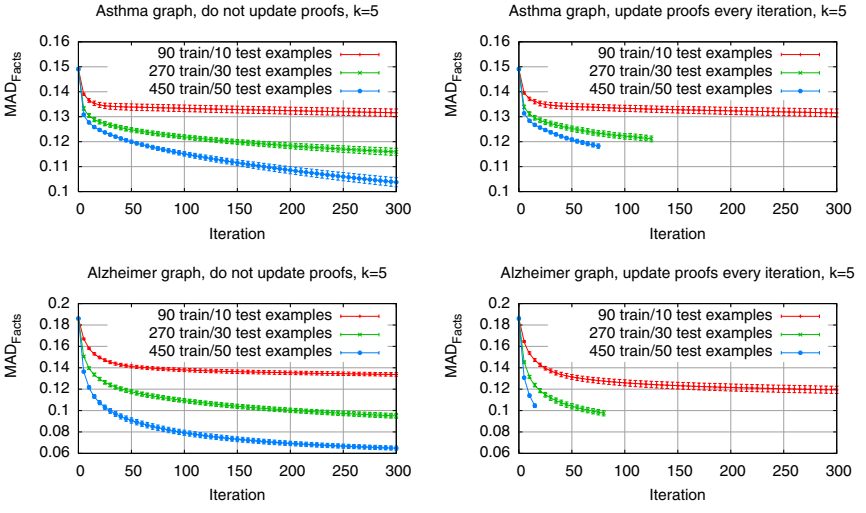

Fig. 3. MAD_{facts} for asthma and Alzheimer using the 5 best proofs ($k = 5$); when the BDDs and proofs are not updated (left column); when they are updated every iteration (right column) (**Q2** and **Q3**)

fact probabilities in this range and we therefore got lower initial errors than by completely random initialization. In general, one can utilize prior knowledge to initialize the parameters. We perform 10-fold crossvalidation in all experiments. The learning rate η was always set to the number of training examples. LeProbLog was implemented in Prolog (Yap-5.1.3) using CUDD for BDD operations.

Q1, Q2: Sanity Check. We attach probabilities to queries in the training set based on the best $k = 5$ proofs. The same approximation is used in the gradient descent algorithm, where the set of proofs to build the BDD is determined anew in every iteration as stated in Algorithm 2. We repeated the experiment using a total of 100, 300, and 500 examples, which we each split in ten folds for crossvalidation. We thus use 90, 270, and 450 training examples. The more training examples are used, the more time each iteration takes. In the same amount of time, the algorithm therefore performs less iterations when using more training examples. The right column of Figure 2 shows the evolvement of the root mean squared error on the test data during learning. The gradient descent algorithm reduces the MSE on both training and test data, with significant differences in all cases (two-tailed t-test, $\alpha = 0.05$). These results affirmatively answer **Q1**.

The MAD_{facts} error is reduced as can be seen in the right column of Figure 3. Again, all differences are significant (two-tailed t-test, $\alpha = 0.05$). Using more training examples results in faster error reduction. This answers **Q2** affirmatively. It should be noted however that in other domains, especially with limited or noisy training examples, minimizing the MSE might not reduce MAD_{facts}, as the MSE is a non-convex non-concave function with local minima.

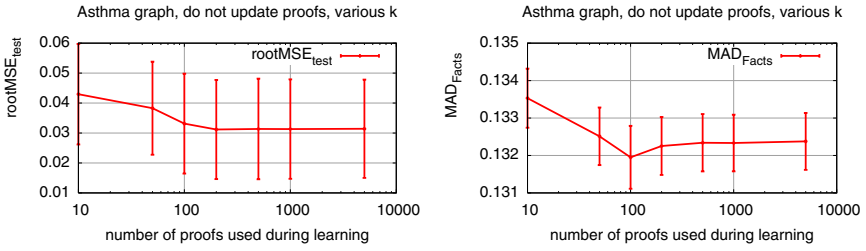

Fig. 4. MAD_{facts} and $\sqrt{MSE_{\text{Test}}}$ after 50 iterations for different k (number of best proofs used) on the asthma graph where training examples carry P_∞ probabilities (**Q4**)

Q3: Error made when the best proofs are not updated. We repeated the same series of experiments, but without updating the set of proofs used for constructing the BDDs. The evolvement of $\sqrt{MSE_{\text{Test}}}$ as well as of MAD_{Facts} is plotted in the left column of Figures 2 and 3 respectively.

The plots for the asthma graph are hardly distinguishable and there is indeed no significant difference (two-tailed t-test, $\alpha = 0.05$). However, the runtime decreases by orders of magnitude, since searching for proofs and building BDDs are expensive operations which had to be done only once in the current experiments. Not updating the BDDs gave a speedup of 10 for the Alzheimer graph. For the Alzheimer graph there is no significant difference for the MSE_{test} (two-tailed t-test, $\alpha = 0.05$), but MAD_{facts} is reduced a little slower (in terms of iterations) when the BDDs are kept constant. However, in terms of time this is not the case. These results indicate that BDDs can safely be kept fixed during learning in this domain which affirmatively answers **Q3**.

Q4: Less proofs, more speed, and still the right results?. In the next experiment, we studied the influence of the number k of best proofs used during learning on the results. We consider the asthma graph with the second dataset, where training example probabilities are lower bounds obtained from the approximation algorithm with interval width 0.01. During learning, P_k is employed to approximate probabilities.

We ran LeProbLog on this dataset and used different values of k between 10 and 5000. We thus aim at learning parameters using an underestimate of the true function, as k best proofs may ignore a potentially large number of proofs used originally. Figure 4 shows the results for this experiment after 50 iterations of gradient descent. As can be seen, the average absolute error per fact (MAD_{facts}) goes down slightly with higher k. The difference is statistically significant for $k = 10$ and $k = 100$ (two-tailed t-test, $\alpha = 0.05$), but using more than 200 proofs has no significant influence on the error. The MSE also decreases significantly (two-tailed t-test, $\alpha = 0.05$) comparing the values for $k = 10$ and $k = 200$, but using more proofs has no significant influence. It takes more time to search for more proofs and to build the corresponding BDDs. These results indicate that using only 100 proofs is a sufficient approximation in this domain and affirmatively answer **Q4**.

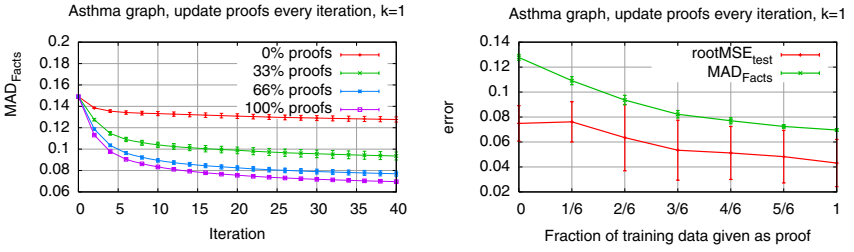

Fig. 5. MAD_{facts} and $\sqrt{MSE_{\text{Test}}}$ after 40 iterations on the asthma graph when different fractions of the data are given as proof (**Q5**)

Q5: Learning from Proofs and Queries. To investigate the effect of using both proofs and queries as examples, we compute the best proof and its probability for 300 examples per graph. For each example, we either use the query or the best proof, both with the probability of the best proof. Learning uses $k = 1$. We use proofs for $0, 50, \ldots, 300$ examples and queries for the remaining ones, and perform stratified 10-fold crossvalidation, that is the ratio of examples given as queries and as proofs was the same in every fold. We updated BDDs in every iteration. Figure 5 shows the results of this experiment. The curve on the left side indicates that the error per fact (MAD_{facts}) goes down faster in terms of iterations when increasing the fraction of proofs. Furthermore, the plot on the right side shows that the root MSE on the test set decreases. These results answer **Q5** affirmatively.

8 Conclusions

We have introduced an approach to learning the parameters of the probabilistic database ProbLog and successfully shown it at work on a real biological application. Interesting directions for future research include conjugate gradient techniques and regularization-based cost functions. Those enable domain experts to successively refine probabilities of a database by stating training examples.

Acknowledgments

Angelika Kimmig and Bernd Gutmann are supported by the Research Foundation-Flanders (FWO-Vlaanderen), Kristian Kersting by a Fraunhofer ATTRACT fellowship. This work is supported by the GOA project 2008/08 Probabilistic Logic Learning and uses HPC resources `http://ludit.kuleuven.be/hpc`.

References

1. Getoor, L., Taskar, B. (eds.): Statistical Relational Learning. MIT Press, Cambridge (2007)
2. De Raedt, L., Frasconi, P., Kersting, K., Muggleton, S. (eds.): Probabilistic Inductive Logic Programming. LNCS (LNAI), vol. 4911, pp. 1–27. Springer, Heidelberg (2008)

3. Dalvi, N.N., Suciu, D.: Efficient query evaluation on probabilistic databases. In: Proceedings of VLDB, pp. 864–875 (2004)
4. De Raedt, L., Kimmig, A., Toivonen, H.: ProbLog: A probabilistic Prolog and its application in link discovery. In: Veloso, M. (ed.) IJCAI, pp. 2462–2467 (2007)
5. Cussens, J.: Parameter estimation in stochastic logic programs. MLJ 44(3), 245–271 (2001)
6. Sato, T., Kameya, Y.: Parameter learning of logic programs for symbolic-statistical modeling. J. Artif. Intell. Res. (JAIR) 15, 391–454 (2001)
7. Friedman, N., Getoor, L., Koller, D., Pfeffer, A.: Learning probabilistic relational models. In: IJCAI, pp. 1300–1309 (1999)
8. Kersting, K., De Raedt, L.: Basic principles of learning bayesian logic programs. In: De Raedt, L., Frasconi, P., Kersting, K., Muggleton, S. (eds.) Probabilistic Inductive Logic Programming. LNCS (LNAI), vol. 4911, pp. 189–221. Springer, Heidelberg (2008)
9. Chen, J., Muggleton, S., Santos, J.: Learning probabilistic logic models from probabilistic examples (extended abstract). In: Blockeel, H., Ramon, J., Shavlik, J., Tadepalli, P. (eds.) ILP 2007. LNCS (LNAI), vol. 4894, pp. 22–23. Springer, Heidelberg (2008)
10. De Raedt, L., Kersting, K.: Probabilistic Logic Learning. ACM-SIGKDD Explorations: Special issue on Multi-Relational Data Mining 5(1), 31–48 (2003)
11. Charniak, E.: Tree-bank grammars. In: AAAI/IAAI, vol. 2, pp. 1031–1036 (1996)
12. De Raedt, L., Kersting, K., Torge, S.: Towards learning stochastic logic programs from proof-banks. In: AAAI, pp. 752–757 (2005)
13. Nottelmann, H., Fuhr, N.: Learning probabilistic datalog rules for information classification and transformation. In: CIKM, pp. 387–394. ACM, New York (2001)
14. Gupta, R., Sarawagi, S.: Creating probabilistic databases from information extraction models. In: VLDB, pp. 965–976 (2006)
15. De Raedt, L., Kersting, K., Kimmig, A., Revoredo, K., Toivonen, H.: Compressing probabilistic prolog programs. Machine Learning 70(2-3), 151–168 (2008)
16. Wrobel, S., Wettschereck, D., Sommer, E., Emde, W.: Extensibility in data mining systems. In: KDD, pp. 214–219 (1996)
17. Kimmig, A., De Raedt, L., Toivonen, H.: Probabilistic explanation based learning. In: Kok, J.N., Koronacki, J., Lopez de Mantaras, R., Matwin, S., Mladenič, D., Skowron, A. (eds.) ECML 2007. LNCS (LNAI), vol. 4701, pp. 176–187. Springer, Heidelberg (2007)
18. Lloyd, J.W.: Foundations of Logic Programming, 2nd edn. Springer, Berlin (1989)
19. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Trans. Computers 35(8), 677–691 (1986)
20. Poole, D.: Logic programming, abduction and probability - a top-down anytime algorithm for estimating prior and posterior probabilities. New Generation Comput. 11(3), 377–400 (1993)
21. De Raedt, L., Kersting, K.: Probabilistic inductive logic programming. In: Ben-David, S., Case, J., Maruoka, A. (eds.) ALT 2004. LNCS (LNAI), vol. 3244, pp. 19–36. Springer, Heidelberg (2004)
22. Mitchell, T.M.: Machine Learnning. McGraw-Hill, New York (1997)
23. Saul, L., Jaakkola, T., Jordan, M.: Mean field theory for sigmoid belief networks. JAIR 4, 61–76 (1996)
24. Sevon, P., Eronen, L., Hintsanen, P., Kulovesi, K., Toivonen, H.: Link discovery in graphs derived from biological databases. In: Leser, U., Naumann, F., Eckman, B. (eds.) DILS 2006. LNCS (LNBI), vol. 4075, pp. 35–49. Springer, Heidelberg (2006)