Parameter Estimation in ProbLog from Annotated Queries

Bernd Gutmann
Angelika Kimmig
Kristian Kersting
Luc De Raedt

Report CW 583, April 2010

Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Parameter Estimation in ProbLog from Annotated Queries

Bernd Gutmann
Angelika Kimmig
Kristian Kersting
Luc De Raedt

Report CW 583, April 2010

Department of Computer Science, K.U.Leuven

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 unobserved probabilities attached to facts that have a low approximation error on the training examples as well as on unseen examples. The objective function to be minimized is the squared-error between the measured and computed values of the queries. As we will show, our approach 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 proofs are available. Experiments on real world data show the usefulness and effectiveness of this least squares calibration of probabilistic databases.

Keywords : Probabilistic Logic Programming, Statistical Relational Learning, Inductive Querying, Probabilistic Inference, Parameter Estimation.
CR Subject Classification : I.2.6

Parameter Estimation in ProbLog from Annotated Queries

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 unobserved probabilities attached to facts that have a low approximation error on the training examples as well as on unseen examples. The objective function to be minimized is the squared-error between the measured and computed values of the queries. As we will show, our approach 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 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 applications involve managing vast 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 residing in databases. Because traditional databases do not allow one to deal with uncertainty, probabilistic extensions of such databases are necessary for managing as well as for mining such data.

In 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. Getoor and Taskar (2007); De Raedt et al. (2008a), but so far seems to have devoted little attention to learning in a probabilistic database setting, that is, a database setting where probabilities are associated to facts or tuples, indicating the probability with which the tuple is in the database (Dalvi and Suciu, 2004; De Raedt et al., 2007). This information is then used to define and compute the probability of derived facts given background knowledge specifying further relationships or predicates. As one example, consider an image processing system that generates high level relational state descriptions of, for instance, traffic situations. The output of such a system could consists out of a set of

facts holding with particular probabilities (Antanas et al., 2009). These facts might state, for instance, the probability that a certain object in the scene is a pedestrian who is walking in a particular direction. The task then could be to recognize, for instance, certain types of traffic violations. Background knowledge might be used to specify different forms of common sense traffic knowledge. As another example imagine a life scientist mining and exploring a large network of biological entities, such as Biomine (Sevon et al., 2006), in an interactive querying session. The biological network in this case is a probabilistic network, in which the edges are represented by probabilistic facts about the biological entities (De Raedt et al., 2007; Sevon et al., 2006). Interesting questions can then be asked about the probability of the existence of a connection between two nodes, or the most reliable path between them. The answers to these questions should provide the life scientist with better insights into the mutual relationships between the queried entities.

The key contribution of the present paper is the introduction of a novel probabilistic database setting for parameter learning from examples together with their target probability. The task is to find 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 prove a query. This learning setting is then incorporated in the probabilistic logic programming system ProbLog (De Raedt et al., 2007), a simple probabilistic extension of Prolog based on Sato's distribution semantics (Sato, 1995). Although ProbLog is a probabilistic programming language, it can be considered as a generalization of a probabilistic database. As probabilistic databases, ProbLog associates probabilities to facts, but it also generalizes such databases by allowing such facts to be non-ground. Therefore, both the problem setting introduced in this paper and the solution developed, can of course easily be integrated in other probabilistic databases as well. The second contribution of this paper is the introduction of an efficient learning algorithm for this problem. It realizes gradient-based optimization using advanced data-structures for efficiently computing the gradient, thereby allowing us to estimate the parameters of ProbLog programs. This algorithm is illustrated at work in three different domains, namely the above mentioned biological network mining setting as well as the UW-CSE (Richardson and Domingos, 2006) and WebKB (Craven and Slattery, 2001) datasets, two standard benchmarks from the statistical relational learning literature. The experiments performed are competitive with those of Markov Logic.

This paper significantly extends the earlier paper (Gutmann et al., 2008) by including parameter estimation for non-ground probabilistic facts using parameter tying, a correctness proof and additional experiments on standard datasets from the Markov Logic literature, where we use a simple heuristic transformation of Markov logic into ProbLog.

We proceed as follows. After reviewing ProbLog in Section 2, we formally introduce the parameter estimation problem for probabilistic databases in Section 3. Section 4, 5, and 6 then present various aspects of our least-squares

approach for solving it. Before concluding, we present the results of an extensive set of experiments on real-world data sets in Section 7 as well as related work in Section 8.

2 ProbLog

As one example of a probabilistic database, we employ ProbLog, a simple probabilistic extension of Prolog introduced in (De Raedt et al., 2007). Alternatively, the database formalisms of (Dalvi and Suciu, 2004) or (Nottelmann and Fuhr, 2001) could be used. A ProbLog program consists – as Prolog – of a set of definite clauses. However, in ProbLog a fact c_i can be labeled with the probability p_i that its ground instances $c_i\theta$ (that is, instances not containing variables) are true. It is also assumed that the probabilities of all ground instances $c_i\theta$ are mutually independent. In the following we repeat the main ideas of ProbLog, see (De Raedt et al., 2007) 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.

Example 1 (Probabilistic Graph). Figure 1(a) shows a small example that can be encoded in ProbLog as follows:

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

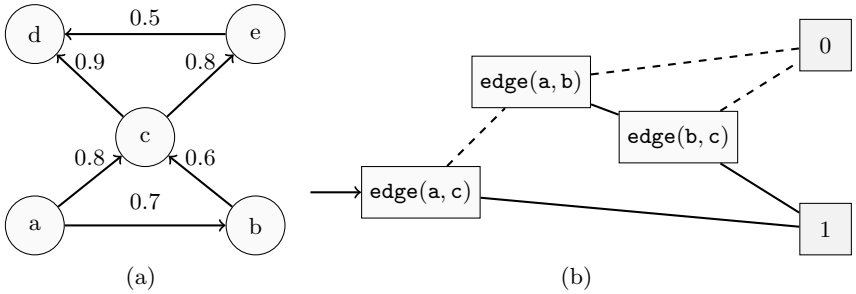

Fig. 1. (a) Example of a probabilistic graph, where edge labels indicate the probability that the edge is part of the graph. (b) Binary Decision Diagram encoding the DNF formula $ac \vee (ab \wedge bc)$, corresponding to the two proofs of query *path(a,c)* in the graph. An internal node labeled xy represents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

It is straightforward to sample subgraphs of a probabilistic graph by tossing a biased coin for each edge. Given a finite set of possible substitutions $\{\theta_{j1}, \ldots \theta_{ji_j}\}$ for each probabilistic fact $p_j :: c_j$, a ProbLog program $T = \{p_1 :: c_1, \cdots, p_n ::$

$c_n\} \cup BK$ defines a probability distribution over ground subprograms $L \subseteq L_T = \{c_1\theta_{11}, \ldots c_1\theta_{1i_1}, \cdots, c_n\theta_{n1}, \ldots, c_n\theta_{ni_n}\}$:

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

Sato has shown how this semantics can be generalized to the infinite case, we refer to (Sato, 1995) for details. For ease of readability, we will restrict ourselves to the finite case here, assuming that the set of possible substitutions is known if the ProbLog program contains non-ground probabilistic facts.

Background knowledge can easily be added 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, which corresponds to 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* using the background knowledge together with a randomly sampled set of ground probabilistic facts.

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 ground probabilistic 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 (Kimmig et al., 2007).

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 (Lloyd, 1989). 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 (De Raedt et al., 2007), the problem can be tackled by reducing the problem to that of computing the probability of the monotone DNF formula

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

where $var(e)$ denotes the set of boolean variables used in the proof e. (Note: Computing this probability is an NP-complete problem.) 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) (Bryant, 1986), an efficient graphical representation of a Boolean function over a set of variables, see Section 5 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 (De Raedt et al., 2007), 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):

$$P_k(q|T) = P\Big(\bigvee\nolimits_{e \in E_k(q)} \bigwedge\nolimits_{a_i \in var(e)} a_i\Big) \tag{4}$$

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. The are two *special cases*: If k is set to ∞, all proofs will be taken into account. And if k is set to 1 the most likely proof – which can be seen as the best explanation – is used. 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 (Poole, 1993)).

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.

Example 2 (Coin toss game). Consider a simple coin game which can be won either by throwing two times heads or by cheating. This game can be modeled by the ProbLog program below. The probability to win the game is then defined by the success probability $P_s(\texttt{win})$. Figure 2 shows the corresponding BDD of

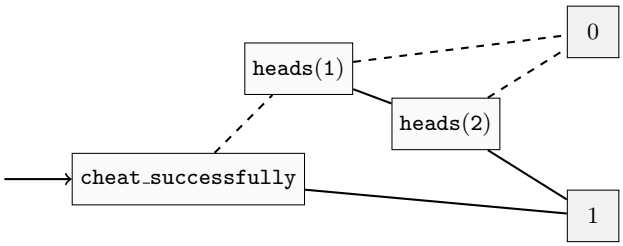

Fig. 2. BDD encoding the DNF `cheat_successfully` ∨ (`heads(1)` ∧ `heads(2)`) corresponding to the possible proofs for `win` in Example 2. The DNF contains two ground instances of the non-ground probabilistic fact `heads(X)` which are treated as different variables by the BDD package but which share the same parameter (fact probability) by parameter tying.

this query.

```
0.5 :: heads(X).                    0.2 :: cheat_succesfully.
win :- cheat_successfully.
win :- heads(1), heads(2).
```

As stated in the beginning of this section, ProbLog is not restricted to *ground* probabilistic facts as used in the graph example, but allows probability labels on *non-ground* facts as well. As an example, consider a sequence of n tosses of the same coin. A probabilistic fact 0.5 :: `heads(X)` can be used to encode that the probability to get heads in a particular round X is always 0.5, where instances for different values of X are independent. Non-ground probabilistic facts thus offer a compact notation for a (potentially infinite) set of ground probabilistic facts sharing the same probability. Note that the semantics is still defined in terms of ground programs, as the success probability of a query with unbound variables asks for the *existence* of some answer and thus depends on the size of the Herbrand universe. From an implementation point of view, we therefore require such facts to be ground on calling, which allows one to dynamically define the Herbrand universe during proving. Internally, a new random variable is created for each grounding used during proving, where parameters are tied. Using non-ground facts, the semantics of ProbLog is still in line with Sato's distribution semantics (Sato, 1995), which also forms the basis of PRISM (Sato and Kameya, 2001) and ICL (Poole, 2008).

Instead of using one non-ground fact, the coin toss game could also be modeled using two ground probabilistic facts (`heads(1)` and `heads(2)`). While this is feasible for a small number of coins, programs quickly grow as the domain size increases, for instance, in sequential domains where the number of time steps is not known beforehand. At the same time, using non-ground probabilistic facts allows for parameter tying between all the ground instances of the fact. This speeds up learning and reduces the number of model parameters.

3 Parameter Learning in Probabilistic Databases

Within probabilistic logical and relational learning (De Raedt and Kersting, 2003; De Raedt, 2008), the problem of parameter estimation can be defined as follows:

Definition 1 (Parameter Learning in Probabilistic Databases). ***Given*** *is a set of examples E, a probabilistic database or probabilistic logic theory D, a probabilistic coverage relation $P(e|D)$ that denotes the probability that the database D covers the example $e \in E$, and a scoring function score. The goal is to* ***find*** *parameters of D such that the scoring is optimal.*

The key difference with logical learning approaches is that the coverage relation becomes probabilistic. Furthermore, within probabilistic logical and relational learning (De Raedt and Kersting, 2003; De Raedt, 2008) three settings are typically considered: learning from entailment, from proofs, and from interpretations. From a database or logic programming perspective, a query corresponds to a formula that is entailed by the database, and hence, learning from queries corresponds to learning from entailment. 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. The parameters of such a grammar can be learned starting from sentences belonging to the grammar (learning from entailment / from queries), or alternatively, from parse-trees (learning from proofs), cf. the work on tree-bank grammars (Charniak, 1996; De Raedt et al., 2005). The former setting is typically a lot harder than the latter one because one query may have multiple proofs, which introduces hidden parameters into the learning setting. When learning from parse-trees, however, these parameters are no longer hidden. The third classical setting uses interpretations as examples. While interpretations provide the most informative examples to the learner, they are often impractical to use. Indeed, as an interpretation states the truth-value of all ground atoms in an example, it is hard to apply this to applications such as grammars or biological networks. For a grammar, an example would have to contain essentially all sentences and constituents that could be constructed with the words in the grammars, possibly an infinite number of them. When considering substructures or paths in a network, the sheer number of them makes explicitly listing them virtually impossible. As we aim at applying our learning approach to biological network mining as well as dealing with large examples in datasets such as WebKB and UW-CSE, we focus on learning from entailment and from proofs in this paper.

Another issue that is important for our setting is that we explicitly target probabilistic examples, that is, the examples themselves will have associated probabilities. The reason is that such examples naturally arise in various applications. For instance, text extraction algorithms return the confidence, experimental data is often averaged over several runs, and so forth. As one illustration consider populating a probabilistic database of genes from MEDLINE[3] abstracts

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

using off-the-shelf information extraction tools, where one might extract from a paper that gene a is located in region b and interacting with gene c with a particular probability denoting the degree of belief; cf. (Gupta and Sarawagi, 2006). This requires one to deal with probabilistic examples such as $0.6 : locatedIn(a, b)$ and $0.7 : interacting(a, c)$. Also in the context of the life sciences, Chen et al. (2008) report on the use of such probabilistic examples, where the probabilities indicate the percentage of successes in an experiment that is repeated several times.

Let us now investigate how we can integrate those two ideas, that is, the notion of a probabilistic example and learning from entailment and proofs, within the ProbLog formalism. When learning from entailment, examples are atoms or clauses that are logically entailed by a theory. Transforming this setting to ProbLog leads to examples that are logical queries, and given that we want to work with probabilistic examples, these queries will have associated target probabilities. When learning from proofs in ProbLog, a proof corresponds to a set of facts, or a conjunction of propositional variables, again with associated target probabilities. It is easy to integrate both learning settings in ProbLog because the logical form of the example will be translated to a monotone DNF formula and it is this last form that will be 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. In our graph example, using the query *path(a,c)* as training 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 learning from proofs and learning from entailment are integrated in one setting.

It is important to realize that the setting considered here differs from the usual statistical relational learning approach with respect to the characteristics of the underlying generative model. As probabilistic context free grammars, both stochastic logic programs (SLPs) (Cussens, 2001) and PRISM programs (Sato and Kameya, 2001) define a generative model at the level of proofs or derivations, and as a consequence, at the level of queries as well. Learning procedures for those models, therefore, often rely on the fact 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 (or exactly 1 for loss-free grammars). ProbLog's generative model, however, lies at the level of interpretations, and therefore does not meet these conditions, as several different facts for the same background knowledge predicate can be true in any possible world. Recently, Chen et al. (2008) also proposed working with probabilistic examples. However, in their work, the probabilities associated with examples are viewed as specifying the degree of being sampled from some distribution employing a generative model on the level of examples or queries, which does not hold in our case. Furthermore, Chen *et al.* only learn from entailment and not from proofs as we do.

By now we are able to formally define the learning setting addressed in this paper:

Definition 2 (Parameter Learning in ProbLog). ***Given*** *a set of training examples* $\{q_i, \tilde{p}_i\}_{i=1}^{M}$, $M > 0$, *where each* $q_i \in \mathcal{Q}$ *is a query or proof and* $\tilde{p}_i$ *is the k-probability of* q_i, ***find*** *a function* $h \in \mathcal{H}$ *with low approximation error on the training examples as well as on unseen examples, where* $\mathcal{H} = \{h : \mathcal{Q} \to [0,1] | h(.) = P_k(.|T')\}$ *comprises all parameter assignments* T' *for a given ProbLog program* T.

Note that in this definition we have chosen the k-probability as probabilistic coverage relation as this allows for maximal flexibility. The definition also leaves the question open how to measure a "low approximation error". In this paper, we propose to use the mean squared error as *error function*

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

Our setting is related to the one considered by Kwoh and Gillies (1996) in that we learn from examples where not everything is observable and that we assume a distribution over the result (that is, whether a query fails or succeeds) and the examples are independent of one another. While in our case, all the observations are binary, Kwoh and Gillies use training examples which contain several variables. Kwoh and Gillies show that minimizing the squared error for this type of problem corresponds to finding a maximum likelihood hypothesis, provided that each training example $(q_i, \tilde{p}_i)$ is disturbed by an error term. The actual distribution of this error is such that the observed query probability is still in the interval $[0, 1]$.

Gradient descent is a standard way of minimizing a given error function. The tunable parameters are initialized randomly. Then, as long as the error does 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.

4 Gradient of the Mean Squared Error

Applying the sum and chain rule to Equation (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 $c_j \in L_T \setminus S$. This formula sums over all subprograms $S \subseteq L_T$ where the query q can be proven. If the fact p_j – the one with respect to which the gradient is derived – is contained in the subprogram, the partial sum is added ($\delta_{jS} = 1$). Otherwise, if the fact p_j is not contained, the partial sum is subtracted ($\delta_{jS} = -1$). Equation (7) 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 straightforward, and then summing over all subsets S where q_i can be proven.

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, i.e. $p_j = \sigma(a_j) := 1/(1+\exp(-a_j))$ applied to $a_j \in \mathbb{R}$. This technique has been used for Bayesian networks and in particular for sigmoid belief networks (Saul et al., 1996). We can 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

$$\frac{\partial P_s(q_i|T)}{\partial a_j} = \sigma(a_j) \cdot (1-\sigma(a_j)) \cdot \sum_{\substack{S \subseteq L_T \\ L \models q_i}} \delta_{jS} \prod_{\substack{c_x \in S \\ x \neq j}} \sigma(a_x) \prod_{\substack{c_x \in L_T \setminus S \\ x \neq j}} (1-\sigma(a_x)). \quad (8)$$

We also have to replace every p_j in Equation (1) by $\sigma(p_j)$. 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 this would happen though, one can take standard counter measures like simulated annealing or random restarts. 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 (De Raedt et al., 2007). In the following section we update this towards the gradient and combine it with a general gradient descent search.

5 Using Gradient Descent for Parameter Learning in ProbLog

To compute the success probability P_s for a query q efficiently, De Raedt et al. (2007) collect all proofs and compactly represent them in a Binary Decision Diagram (BDD) (Bryant, 1986). BDDs, one of the best understood data structures today, have been used to solve a wide variety of computer science problems. They can be viewed as a compact encoding of a Boolean decision tree. 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 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

Algorithm 1 Evaluating the gradient of a query efficiently by traversing the corresponding BDD, calculating partial sums, and adding only relevant ones. The algorithm returns the probability of the query and the gradient with respect to the target fact n_j. Since we reparameterized the search space, we apply the sigmoid function $\sigma : \mathbb{R}^n \mapsto]0, 1[$ to obtain probabilities.

```
1: function GRADIENT(node n, target fact n_j)
2:     if n is the 1-terminal then return (1, 0)
3:     if n is the 0-terminal then return (0, 0)
4:     Let h and l be the high and low children of n
5:     (prob(h), grad(h)) := GRADIENT(h, n_j)
6:     (prob(l), grad(h)) := GRADIENT(l, n_j)
7:     prob := σ(a_n) · prob(h) + (1 − σ(a_n)) · prob(l)
8:     grad := σ(a_n) · grad(h) + (1 − σ(a_n)) · grad(l)
9:     if n ⊆_Θ n_j then                    ▷ Current node n is a ground instance of n_j
10:        grad := grad + (prob(h) − prob(l)) · σ(a_j)(1 − σ(a_j))
11:    return (prob, grad)
```

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. (2007) 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 extend this to the computation of the gradient (cf. Equation (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 and assume there are no non-ground facts. 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 with respect to the target fact n_j can be obtained by simply summing the gradients of all branches ending in a leaf labeled 1. In BDDs however, isomorphic subparts 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. Furthermore, when using non-ground facts the same variable n_x might appear several times on a particular path, where the nodes correspond to different ground instances of the fact. If n_1 and n_2 are ground instances of x (written $n_1 \subseteq_\Theta x$ and $n_2 \subseteq_\Theta x$) appearing as true on the branch $n_1 \cdot n_2 \cdot \ldots \cdot n_i$, then the gradient w.r.t. x is $n_2 \cdot \ldots \cdot n_i + n_1 \cdot n_3 \cdot \ldots \cdot n_i$. To account for that, we combine all paths and all ways to apply the derivation rule exactly once. At the root of the BDD we sum over all those paths. The details can be found in the correctness proof for Algorithm 1 in Appendix A.

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 the probability of the sub-BDD. The symbol $\subseteq_\Theta$ denotes the *is-an-instance-of* relation. In terms of first order logic, $a \subseteq_\Theta b$ holds if there exists a substitution Θ such that $a = b\Theta$. For instance $\texttt{heads(1)} \subseteq_\Theta \texttt{heads(1)}$, $\texttt{heads(2)} \subseteq_\Theta \texttt{heads(X)}$ but $\texttt{heads(2)} \not\subseteq_\Theta \texttt{heads(1)}$. Let us reconsider the coin toss example for illustration.

Example 3 (Gradient of a query). Suppose we want to estimate unknown fact probabilities are unknown (indicated by the symbol ??) from the training example $P(\texttt{win}) = 0.3$.

$$?? :: \texttt{heads(X)}. \qquad ?? :: \texttt{cheat\_succesfully}.$$
$$\texttt{win} : -\texttt{cheat\_successfully}.$$
$$\texttt{win} : -\texttt{heads(1)}, \texttt{heads(2)}.$$

As a first step the fact probabilities get initialized with some random probabilities:

$$0.6 :: \texttt{heads(X)}. \qquad 0.2 :: \texttt{cheat\_succesfully}.$$
$$\texttt{win} : -\texttt{cheat\_successfully}.$$
$$\texttt{win} : -\texttt{heads(1)}, \texttt{heads(2)}.$$

In order to calculate the gradient of the MSE (cf. Equation (6)), the algorithm evaluates the partial derivative for every probabilistic fact and every training example. Figure 3 illustrates how the partial derivate $\partial P(\texttt{win})/\partial \texttt{heads(X)}$ is obtained by running Algorithm 1. Note that $\texttt{heads(X)}$ is a non-ground fact and the BDD contains two nodes, which can be unified with the target fact $\texttt{heads(X)}$, written $\texttt{heads(1)} \subseteq_\Theta \texttt{heads(X)}$ and $\texttt{heads(2)} \subseteq_\Theta \texttt{heads(X)}$ respectively.

To obtain the final learning algorithm, the BDD-based gradient calculation is combined 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 this gradient descent search.

6 Imbalanced Data Sets

In some applications the probabilities of the training examples are limited to 1 and 0. We refer to them as positive and negative examples respectively. If in such domains, the training set is imbalanced, in the sense that the number of positive training examples significantly differs from the number of negative training examples, the MSE as defined in Equation (5) performs poorly as we discovered in our experiments. To account for this we use a weighted MSE

$$MSE_{\text{cost}}(T) = \frac{1}{M} \sum_{1 \leq i \leq M} \left(P_s(q_i|T) - \tilde{p}_i\right)^2 \cdot \text{cost}(\tilde{p}_i) \tag{9}$$

Algorithm 2 Gradient descent for ProbLog, 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, n_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

where $cost(1) := 1$ and $cost(0) := \alpha$. Thus, negative training examples have only α times the influence on the MSE than positive training examples. When deriving the gradient (cf. Equation (6)) the *cost* factor can be treated as constant resulting in

$$\frac{\partial MSE_{\text{cost}}(T)}{\partial p_j} = \frac{2}{M} \sum_{1 \leq i \leq M} \underbrace{\left(P_s(q_i|T) - \tilde{p}_i\right)}_{\text{Part 1}} \cdot \underbrace{\frac{\partial\ P_s(q_i|T)}{\partial p_j}}_{\text{Part 2}} \cdot \text{cost}(\tilde{p}_i). \tag{10}$$

Introducing the *cost* factor corresponds to an asymmetric loss matrix

$$\begin{bmatrix} P_s(q_i|T) \\ 1 - P_s(q_i|T) \end{bmatrix}^T \cdot \begin{bmatrix} 0 & \alpha \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} \tilde{p}_i \\ 1 - \tilde{p}_i \end{bmatrix} \tag{11}$$

which is a common technique for classification problems with non-uniform class distributions (Bishop, 2006). While the standard MSE assumes a uniform prior over positive and negative examples, the cost-based MSE assumes a non-uniform prior.

Since the cost term introduces an additional parameter it is reasonable to ask which value to use for α. As we will show later in the experiments, we suggest to set $\alpha = M_p/M_n$ where M_P and M_n are the numbers of positive and negative training examples respectively. This particular value assures that both the positive and the negative examples have in total the same influence on the gradient. If the ratio of M_p to M_n is approximately 1, then $\alpha \approx 1.0$. Whereas if M_p is much smaller than M_n then α is very low, which in turn degrades the influence of a single negative example.

7 Experiments

In this section, we empirically evaluate the proposed approach to parameter estimation. The experiments are designed to get insight into

A the quality of the estimated parameters,
B the influence of using approximations in the algorithm (such as choosing a low k in P_k and not updating BDDs every iteration)
C the interplay between learning from proofs and learning from entailment, and
D the performance of the approach as compared to state-of-the-art statistical relational learning systems such as Alchemy.

To this aim, we implemented the gradient descent algorithm for ProbLog in Prolog (Yap-5.1.3)[4]. Before presenting the results of our experimental investigation of the four issues sketched above, we will now describe the data-sets, the evaluation criteria used as performance measure, and the initialization of the parameters.

7.1 Datasets

We consider three different datasets in our experiments

- The *Biomine* graph (Sevon et al., 2006) is a large biological network extracted from various sources. The nodes correspond to entities such as genes, diseases, or medical papers. The edges indicate dependencies and they are labeled with probabilities. Two subgraphs from the Biomine network have been derived: one around Alzheimer disease and another around Asthma; cf. Appendix C for more details.
- The *UW-CSE* dataset (Richardson and Domingos, 2006) comprises information about the computer science department of the University of Washington. It contains 12 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 contains in total 3880 tuples. It was obtained by crawling the Web site of the computer science department of the University of Washington[5] and the BibServ database[6]. The database is split into five subdatabases, each containing tuples of a particular area of the CS department: AI, graphics, programming languages, systems, and theory.
- The *WebKB* dataset (Craven and Slattery, 2001) contains the link structure of the web pages of 4 universities: Cornell, Texas, Washington and Wisconsin. For every web page the stems of words appearing on it are listed. Each page is labeled with a subset of the possible classes person, student, faculty, professor, department, research project, and course. The goal is to predict the class of an unseen page by using the information on words and links.

[4] For the BDD operations we used SimpleCUDD a wrapper around CUDD, it is available at `http://www.cs.kuleuven.be/~theo/tools/simplecudd.html`
[5] `http://www.cs.washington.edu`
[6] `http://www.bibserv.org`

7.2 Evaluation Metrics

We use the following metrics to assess the results

- The mean squared error (MSE), cf. Equation (5), measures the difference between the distributions defined by two sets of probabilities with respect to a set of datapoints. If the MSE is zero, the distributions agree on the data, if the MSE is greater than 0 they differ. We report $\sqrt{MSE_{\text{Test}}}$, the root of the MSE on the hold-out dataset, averaged over all folds.
- The mean absolute difference of the fact probabilities MAD_{facts} defined as

$$MAD_{\text{facts}} := \frac{1}{n} \sum_{j=1}^{n} \left| p_j - p_j^{\text{true}} \right|$$

  measures how close the estimated fact probabilities p_j are to the ground truth probabilities p_j^{true}. We use this measure on the Biomine dataset since the ground truth probabilities are known there.
- The area under the precision-recall curve *AUC* is used for the UW-CSE and WebKB datasets. We report the average AUC as well as the standard deviation on the hold out dataset which we calculate in the same way as Richardson and Domingos (2006).

7.3 Methodology

The initial fact probabilities were sampled randomly as follows. For the Biomine and UW-CSE dataset we sampled uniformly in the interval $[-0.5, 0.5]$ and applied the sigmoid function which yielded probability values in the interval $[0.43, 0.57]$. For the WebKB dataset we sampled the initial fact probabilities uniformly in $[0.03995, 0.04005]$. The learning rate η was always set to the number of training examples. We performed 10-fold cross validation on the Biomine dataset and leave-one-out cross validation on the UW-CSE and WebKB dataset. On the Biomine dataset we ran the experiments with a time limit of 24 hours per fold - depending on the number of training example used this lead to different numbers of iterations of gradient descent. On the UW-CSE dataset we ran 200 iterations of gradient descent and on the WebKB we ran 60 iterations of gradient descent. These different values are due to practical limitations, such us the maximum job duration on the cluster and available computing nodes.

7.4 Quality of Estimated Probabilities

The first set of experiments is meant to answer the following question:

Q1 Does the method reduce the mean squared error on both the training and test set?
Q2 Can it recover the original parameters?

These questions serve as an initial sanity check for both the algorithm and the implementation. To answer them, we employed the asthma and Alzheimer datasets – subgraphs of the Biomine graph – and sampled 500 random node pairs (a, b) in these graphs, estimating the query probability for path(a,b) using P_5, the probability of the 5 best proofs. We used the same approximation $k = 5$ 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 cross validation. 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 4 shows the change of $\sqrt{MSE_{\text{Test}}}$ 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**.

Also, the MAD_{facts} error is reduced as can be seen in the right column of Figure 5. Again, all differences are significant (two-tailed t-test, $\alpha = 0.05$). Using more training examples results in faster error reduction. These results affirmatively answer **Q2**. 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.

7.5 Influence of Approximations

Our algorithm relies on several computationally expensive operations. First, recomputing the proofs (and especially the associated BDDs) for each query in each iteration is expensive. On the other hand, BDDs can easily be saved and reevaluated with updated parameters, providing an approximation of the results obtained using the true best proofs. Second, using the exact success probability P_∞ may result in large, computationally intractable BDDs. Choosing P_k with a smaller k as an approximation would again result in significant computational savings. To get insight into the influence of such approximations, we set up experiments to answer the following questions:

Q3 Is it necessary to update the set of k best proofs in each iteration?
Q4 Can we obtain good results approximating P_∞ by P_k for finite (small) k?

To answer **Q3**, we used the same series of experiment as before, but now without updating the set of proofs used for constructing the BDDs. The change of $\sqrt{MSE_{\text{Test}}}$ as well as of MAD_{Facts} are plotted in the left column of Figures 4 and 5 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 answer **Q3**.

To answer **Q4**, we sampled 200 random node pairs (a, b) from the asthma graph and estimated the probability $P_\infty(path(a, b))$ using the lower bound of the approximative inference algorithm (De Raedt et al., 2007) with interval width $\delta = 0.01$. During learning, however, we employ P_k to approximate probabilities. We ran parameter learning for ProbLog on this dataset, varying 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. Figure 6 shows the results for this experiment after 50 iterations of gradient descent. As can be seen, the average absolute error per fact (MAD_{facts}) decreases 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**.

7.6 Learning from entailment and from proofs

We introduced the first learning algorithm that simultaneously employs proofs and queries as examples. Therefore, it is interesting to investigate:

Q5 Do proofs carry more information than queries?

To answer this question, we created mixed data sets containing both proofs and queries. This was realized by sampling 300 random node pairs (a, b) and computing P_1 for path(a,b), the probability of the best path between a and b from both the Asthma and Alzheimer graphs. We then constructed several sets where different proportions of the examples where given as proof, the edges of the best path, instead of the path(a,b) query. Learning uses $k = 1$. We used proofs for $0, 50, \ldots, 300$ examples and queries for the remaining ones, and performed stratified 10-fold cross validation, 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 7 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 confirm that proofs carry more information and hence, that **Q5** can be answered affirmatively.

7.7 Comparison to state-of-the-art

The other question of interest is, of course:

Q6 Does ProbLog perform as well as current state-of-the art statistical relational learning methods?

To answer this question, we provide a comparison to Markov Logic on the UW-CSE dataset, which was introduced by Richardson and Domingos (2006). The goal is to predict the `advisedBy` relation given the other predicates. To apply our algorithm to this dataset, we translated the Markov logic network (MLN) used by Richardson and Domingos (2006) into ProbLog clauses using Algorithm 3 described in Appendix B. This translation yielded a ProbLog program with unknown fact probabilities, which were then learned using the gradient descent algorithm and *leave-one-department-out* cross-validation. Given n person constants in a subdatabase, we generated n^2 training examples, one for every possible grounding of the `advisedBy(X,Y)` atom. The training examples got the probability 1.0 if the particular `advisedBy` tuple was contained in the sub database, otherwise 0.0. The resulting training sets are highly imbalanced: Averaged over all departments, the ratio M_P/M_n of positive to negative examples was ≈ 0.00756. First test runs with the standard MSE showed poor performance, namely all the fact probabilities were set to values close to zero. To account for the imbalance in the training set, we therefore minimized the cost-based MSE_{cost} (cf. Equation (9) in Section 6) with $\alpha = 0.00756$.

The clauses generated by converting the MLN contain cycles which irritate Prolog's depth-first inference mechanism. We therefore imposed a depth-limit of four, that is, only proofs with at most 4 probabilistic facts were incorporated in calculating the probability of a query. Using $k = 1000$ to approximate probabilities, we observed that no query had more than 100 proofs obeying the depth limit, meaning that all proofs are contained in the initial sets of proofs. We therefore reused initial BDDs during learning to speed up the algorithm.

We ran 200 iterations of gradient descent with the learning rate η set to the number of training examples. For the prediction we ran the ProbLog inference algorithm to calculate $P_s(\texttt{advisedBy}(\texttt{X},\texttt{Y})|L)$ for every possible grounding of `advisedBy(X,Y)` using person constants from the test set. Then we choose a threshold τ and classified those atoms as true which had a success probability of at least τ. Others were classified as negative. By varying τ from 0 to 1 we got the precision-recall curve.

We repeated this experiment twice. In the *All Info* setting, all predicates were available during learning, whereas for *Partial Info*, we removed the `student(X)` and `professor(X)` predicates which made learning and inference harder.

Table 1 shows the results of this experiment. We used the same setup as Richardson and Domingos (2006) to have an objective comparison. In their experiments, MLN(KB) and MLN(KB+CL) perform best. We extracted the graphs for those two systems from their plots and included them in our plots (Figure 8 - Figure 13). The difference between ProbLog and MLN(KB) is statistically significant in the All Info case (two-tailed t-test, $\alpha = 0.05$), whereas it is not significant

for the Partial Info case. These results affirmatively answers **Q6**, confirming that our approach is competitive.

Table 1. Experimental results for predicting `advisedBy(X, Y)` when all other predicates are known (All Info) and when `student(X)` and `professor(X)` are unknown (Partial Info). AUC is the area under the precision-recall curve. The results for MLN(KB) and MLN(KB+CL) are copied from (Richardson and Domingos, 2006). To compute the AUC and the standard deviation we used the Richardson and Domingos' method. ProbLog was trained using MSE_{cost} with $\alpha = 0.0075566$ (**Q6**).

System	All Info	Partial Info
ProbLog	0.260 ± 0.0223	0.223 ± 0.0182
MLN(KB)	0.215 ± 0.0172	0.224 ± 0.0185
MLN(KB+CL)	0.152 ± 0.0165	0.203 ± 0.0196

To assess the influence of the cost parameter α on the outcome, we repeated the experiment with different values. For $\alpha = 1$ we get the standard MSE which performs worst. For $\alpha = M_P/M_N$ we get the best result in the All Info setting and fairly good results in the Partial Info setting. There is a strong correlation between both graphs, with an optimum value around $\alpha = M_P/M_N$ for the Full Info case and with a rather good value in the Partial Info case. This result indicates that the choice of α is a suitable one. The results also suggest that using a hold-out dataset to tune α could potentially yield significantly better results.

As second test case for question **Q6** we considered the WebKB dataset (Craven and Slattery, 2001). The ProbLog program we used for this problem is related to the MLN used by Lowd and Domingos (2007). But in difference to their model, we ignore words which are absent on a page while they explicitly take them into account. Our model consists out of two parts. The first part captures the dependencies between words appearing on a particular page and the class of the page. The program contains one probabilistic fact `word_class`($Word$,$Class$) for each combination of $Word$ and $Class$, resulting in $774 \cdot 6 = 4644$ probabilistic facts (774 word stems, 6 possible class labels).

```
?? :: word_class(Word,Class).
class(Page, C, Depth) :-
   word_class(W, C),
   has_word(Page, W).
```

The class person always co-occurs with faculty, student, or staff. Therefore we treated it separately by not generating `link_class/4` and `word_class/2` facts for person. Instead, we used the following clauses to express that if a page is classified as student, staff, or faculty page it should also be classified as a person

page.

$$\texttt{class}(\texttt{Page},\texttt{person},\texttt{D}) : -\texttt{class}(\texttt{Page},\texttt{student},\texttt{D}).$$
$$\texttt{class}(\texttt{Page},\texttt{person},\texttt{D}) : -\texttt{class}(\texttt{Page},\texttt{staff},\texttt{D}).$$
$$\texttt{class}(\texttt{Page},\texttt{person},\texttt{D}) : -\texttt{class}(\texttt{Page},\texttt{faculty},\texttt{D}).$$

The second part of our model captures the dependencies between pages. We generated one non-ground probabilistic fact $\texttt{link\_class}(\texttt{P1},\texttt{P2},Class1,Class2)$ for every combination of two classes – except person. The variables P1 and P2 get instantiated by the identifiers of the two pages involved in a link. Using non-ground facts, yields independent facts for every ground instance – namely every link. The counter Depth is decreased every time a link is followed in a proof to prevent endless cycles. During learning and inference we set $\texttt{Depth} = 1$ which restricts the search to the direct neighborhood of each page.

$$?? :: \texttt{link\_class}(\texttt{P1},\texttt{P2},Class1,Class2).$$
$$\texttt{class}(\texttt{Page},\texttt{C},\texttt{Depth}) : -$$
$$\quad \texttt{Depth} > 0, \texttt{Depth2 is Depth} - 1,$$
$$\quad \texttt{links\_to}(\texttt{OtherPage},\texttt{Page}),$$
$$\quad \texttt{class}(\texttt{OtherPage},\texttt{COther},\texttt{Depth2}),$$
$$\quad \texttt{link\_class}(\texttt{OtherPage},\texttt{Page},\texttt{COther},\texttt{C}).$$

We ran 60 iterations of gradient descent and performed *leave-one-out* cross validation. We repeated this experiment twice. In the first run, we used only the first part of our model which considers the words appearing on a page. In the second run, we used the full model which considers both words and links. We refer to the runs as *words* and *words+links* respectively. The results are shown in Figure 15. As expected, the *words+links* model outperforms the version restricted to words. It reaches its high of 0.606 ± 0.003 in iteration 50 and then slightly overfits. This result is in the same range as the one obtained by Lowd and Domingos (2007) using a voted perceptron algorithm (≈ 0.605) and the contrastive divergence algorithm (≈ 0.604). However, Lowd and Domingos also were able to improve the AUC up to ≈ 0.73 using second order gradient techniques such as scaled conjugated gradients. n each page. These results again affirmatively answer **Q6** and show an interesting direction for future work – namely, applying second order gradient techniques for ProbLog.

8 Related Work

Probabilistic relational models (PRMs) (Friedman et al., 1999) and Bayesian logic programs (BLPs) (Kersting and De Raedt, 2008) are relational extensions of Bayesian networks using entity relationship models or logic programming respectively. Similar to ProbLog, both frameworks define generative models on the level of interpretations, and learning methods for PRMs and BLPs thus use interpretations as examples. However, while learning from full interpretations is theoretically possible and straightforward in ProbLog, it suffers from practical limitations, especially in applications where interpretations are huge. Indeed,

consider the probabilistic network, where full interpretations would contain information on edges (the probabilistic facts) as well as paths (as following from the background knowledge), whereas natural examples would typically focus on some specific paths only, and often not include edges at all. It is unclear how different paths could be sampled and, clearly, the sum of the probabilities 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.

Riguzzi's ALLPAD (Riguzzi, 2008) for learning ground LPADs is related to the work presented here in that it also uses training examples with associated probabilities. However, examples are interpretations there, and the focus of his work lies on learning the structure of an LPAD by combining a complete search for clauses with finding an approximate solution to a constraint satisfaction problem, whereas the parameters for a given structure can be obtained directly from the probabilities of the training examples.

Within the probabilistic database community, parameter estimation has received surprisingly few attention. Nottelmann and Fuhr (2001) consider learning probabilistic Datalog rules in a setting with underlying distribution semantics similar to ProbLog's. 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 uses the training probabilities differently: they generate training examples labeled with 0/1 randomly according to the observed probabilities whereas we use the observed probabilities directly. Finally, whereas our learning algorithm follows a principled gradient approach employing all combinations (or a subset) 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 a hold-out validation set. Gupta and Sarawagi (2006) also consider a closely related learning setting but only extract probabilistic facts from data.

Recently, approaches that compile probabilistic inference problems into propositional formulae are becoming increasingly popular. Ishihata et al. (2008) present a general EM-algorithm based on BDDs, which opens an interesting perspective for alternative learning techniques for ProbLog. Darwiche (2002) uses arithmetic circuits (ACs) for probabilistic inference in belief networks. A multi-linear function encoding the belief network is first represented in d-DNNF, a generalization of BDDs, and then translated into an AC. Probabilistic queries are answered by calculating partial derivatives of the multi-linear function on the AC, which can be done in one pass through the AC similar to the gradient calculation on BDDs for ProbLog parameter estimation.

In a sense, keeping the BDD fixed when using the k-probability for learning exploits a similar idea as Friedman's structural EM learning for Bayesian networks (Friedman, 1997), as it also reuses structures computed for similar problems. However, in contrast to structural EM, we do not evaluate the changes, but use the old structure as an approximation of the new one.

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 (De Raedt et al., 2008b). 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 (Wrobel et al., 1996), albeit that only the parameters are changed and not the structure.

9 Conclusions

We have introduced a novel setting to learn parameters of probabilistic databases that integrates the classical settings of both learning from entailment and learning from proofs with the use of probabilistic examples. Probabilistic databases pose new challenges for parameter learning, as they define a distribution on the level of interpretations, but interpretations are typically too large to be used as training examples for parameter learning. We use ProbLog, a logic-programming based generalization of probabilistic databases, in our investigation of this setting, introducing a gradient descent approach that extends ProbLog's efficient BDD-based inference mechanism to parameter learning. Extensive experiments on three well-known real world datasets confirm the practicality of the approach, illustrate the advantages of the approximation options it offers, and demonstrate that parameter learning for ProbLog is competitive with state-of-the-art techniques in statistical relational learning.

A Proof of Correctness of Algorithm 1

Theorem 1. GRADIENT$(root(b_F), n_j)$ *returns the probability* $P(F = true)$ *and the partial derivative* $\partial P(F = true)/\partial a_j$*, if* b_F *is a BDD representing the boolean function* F *and* $\sigma(a_j) = 1/(1 + \exp(-a_j))$ *is the probability of the (non-) ground probabilistic fact* n_j*.*

Proof. We prove the Theorem 1 by induction over the structure of the BDD. There are two *base cases* as shown in Figure 16. If the BDD consists only of the 1-terminal it represents the function $F = true$ and the probability that F is true is 1. If the BDD consist only of the 0-terminal, it represents the function $F = false$. The probability of F being true is 0. The gradient of F with respect to X is 0 in both cases.

For the *inductive case* the function F is – due to the construction of the BDD – represented as $F = (n \wedge F_{True}) \vee (\neg n \wedge F_{False})$ where both F_{True} and F_{False} are BDDs. Figure 17 illustrates this. The probability that F yields true can be calculated by $P(F = true) = P(n) \cdot P(F_{True} = true) + (1 - P(n)) \cdot P(F_{False} = true)$ where we make use of the fact that F_{True} and F_{False} are disjoint with respect to n because of the BDD structure. Furthermore we apply the inductive

hypothesis which says, that the algorithm will return the correct probabilities if it is applied to F_{True} and F_{False}. Since we re-parameterize the search space, the probability $P(n)$ is given by $\sigma(a_n) = 1/(1 + \exp(-a_n))$. This part of the proof covers line 2-7 of Algorithm 1. For the gradient with respect to X there are two cases.

- If $n \not\subseteq_\Theta n_j$, namely the current node is not a ground instance of the target fact n_j, then

$$\begin{aligned}\frac{\partial P(F=1)}{\partial a_j} =& \frac{\partial\Big(P(n)\cdot P(F_{True}=true)\Big)}{\partial a_j} + \frac{\partial\Big((1-P(n))\cdot P(F_{False}=true)\Big)}{\partial a_j} \\ =& \sigma(a_n)\cdot\frac{\partial P(F_{True}=true)}{\partial a_j} + (1-\sigma(a_n))\cdot\frac{\partial P(F_{False}=true)}{\partial a_j}\ .\end{aligned}$$

  The probability of the probabilistic (non-) ground fact corresponding to node n does not depend on a_j and we can treat $P(n) = \sigma(a_n)$ as a constant factor. This part of the proof covers line 8 of Algorithm 1.
- If $n \subseteq_\Theta n_j$, namely the current node n is a ground instance of n_j, we get

$$\frac{\partial P(F=1)}{\partial a_j} = \frac{\partial\Big(P(n)\cdot P(F_{True}=true)\Big)}{\partial a_j} + \frac{\partial\Big((1-P(n))\cdot P(F_{False}=true)\Big)}{\partial a_j}$$

  where $P(n) = \sigma(a_j)$ is a function of a_j. Applying the product rule twice yields

$$\begin{aligned}=& \sigma(a_j)\cdot(1-\sigma(a_j))\cdot P(F_{True}=true) + \sigma(a_j)\cdot\frac{\partial P(F_{True}=true)}{\partial a_j} - \\ & \sigma(a_j)\cdot(1-\sigma(a_j))\cdot P(F_{False}=true) + (1-\sigma(a_j))\cdot\frac{\partial P(F_{False}=true)}{\partial a_j}\ .\end{aligned}$$

  This part of the proof covers line 9-11 of Algorithm 1.

In both cases, we use the inductive hypothesis which says that the algorithm computes the correct values for the partial derivatives of F_{True} and F_{False}. □

B Translating Markov Logic Constraints into ProbLog Clauses

A Markov logic network (MLN) is a set of weighted first-order clauses. Together with a set of constants representing objects in the domain of interest, it defines a Markov network with one node per ground atom and one feature per ground clause. The weight of a feature is the weight of the first-order clause that originated it. The probability of a state $\mathbf{x}$ in such a network is given by $P(\mathbf{x}) = \frac{1}{Z}\exp\left[\sum_i w_i \cdot g_i(\mathbf{x})\right] = \frac{1}{Z}\prod_i f_i(\mathbf{x})$, where w_i is the weight of the ith

Algorithm 3 The function ConvertConstraint translates a constraint c which is a disjunction of positive c^+ and negative c^- literals into ProbLog clauses and facts. Every probabilistic fact introduced during conversion gets a globally unique identifier. If the constraint does not contain positive literals, we use an error predicate and an additional training example to preserve the meaning of the constraint.

```
1: function ConvertConstraint(c)
2:     if c+ = ∅ then
3:         ID := UniqueNumber()
4:         return {error(ID) : −c1−, c2−, ..., ck−}
5:     else
6:         result := ∅
7:         for p ∈ c+ do
8:             ID := UniqueNumber()
9:             result := result ∪{?? :: fact(ID)} ∪ {p : −fact(ID), c1−, c2−, ..., ck−}
10:    return result
```

clause, $g_i = 1$ if the ith clause is true, $g_i = 0$ otherwise. Inference can be carried out by creating the ground network and running any Markov network inference algorithm such as belief propagation.

To convert the Markov logic clauses into ProbLog clauses, we used the function ConvertConstraint shown in Algorithm 3. It takes as input a constraint c and returns a set of clauses and probabilistic facts. The set of positive and negative literals appearing in c are depicted by c^+ and c^- respectively. Constraint weights are not transformed, the corresponding fact probabilities are set to ?? indicating that they have to be estimated using parameter learning. For speed reasons, clauses were manually reordered. Finally, we deleted duplicated clauses.

Table 2 shows some examples of translated constraints. Clauses which contain only negative literals, cf. the third example in the table, can not directly be represented in ProbLog. In order to deal with such constraints, we define an *error predicate*, that does not carry probabilistic facts directly. During training we provide additional training examples of the form `training_example(error(D,N),0.0)`, one for each department `D` and error predicate `N`. This ensures that the probability for the error predicate being true stays low. We restricted the translation to ground probabilistic facts in order to limit the size of the BDDs and speed up learning. Every grounding of a non-ground fact introduces a variable in the BDD which in turn increases the time to build and traverse the BDD. However, it is possible to use non-ground facts. The first constraint in Table 2 – for instance – can be translated into:

```
?? :: fact(1,P,S).
professor(Department,P) :−
   advisedBy(Department,P,S),
   fact(1,P,S).
```

In difference to MLNs – where constraints always get grounded – selectively using non-ground facts allows one to trade off between runtime and memory on the one hand and expressivity on the other hand.

Table 2. Constraints in Markov Logic and their translation using Algorithm 3. Note that in order to deal with constraints consisting only out of negated atoms, like the third example, we have to provide additional training examples. We follow the Prolog notation, depicting variables by capitalized letters, constants and functors by lowercase letters. The atoms are extended by an additional argument `Department` which allows to store all training examples in a single database.

MLN Constraint	Resulting ProbLog Code
?? ¬advisedBy(P, S) ∨ professor(P)	?? :: fact(1). professor(Department, P) : − fact(1), advisedBy(Department, P, S).
?? ¬tempAdvisedBy(X, Y) ∨ yearsInProgram(X, 1) ∨ yearsInProgram(X, 2)	?? :: fact(2). yearsInProgram(Department, X, 1) : − fact(2), tempAdvisedBy(Department, X, Y). ?? :: fact(3). yearsInProgram(Department, X, 1) : − fact(3), tempAdvisedBy(Department, X, Y).
?? ¬student(X) ∨ ¬professor(X)	error(Department, 4) : − student(Department, X), tprofessor(Department, X). training_example(error(Department, 4), 0.0).

C The Biomine Network

As working with the full Biomine network would involve estimating several millions of parameters, we extracted two subgraphs from Biomine (Sevon et al., 2006), one around Alzheimer and another one around Asthma. 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 edges as described in (Sevon et al., 2006). 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.

Acknowledgements

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 high performance computational resources provided by the University of Leuven, `http://ludit.kuleuven.be/hpc`. We thank Theofrastos Mantadelis for the development of SimpleCUDD. We thank Marc Sumner for the help with the fastAUC evaluation script.

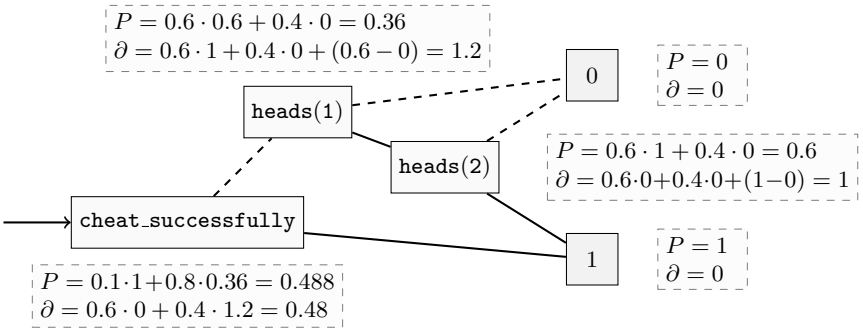

Fig. 3. Intermediate results when calculating the gradient $\partial P(\texttt{win})/\partial \texttt{heads(X)}$ using Algorithm 1. Both the node `head(1)` and `heads(2)` are ground instances of the target fact ($\subseteq_\Theta$). The result is read off at the root node of the BDD.

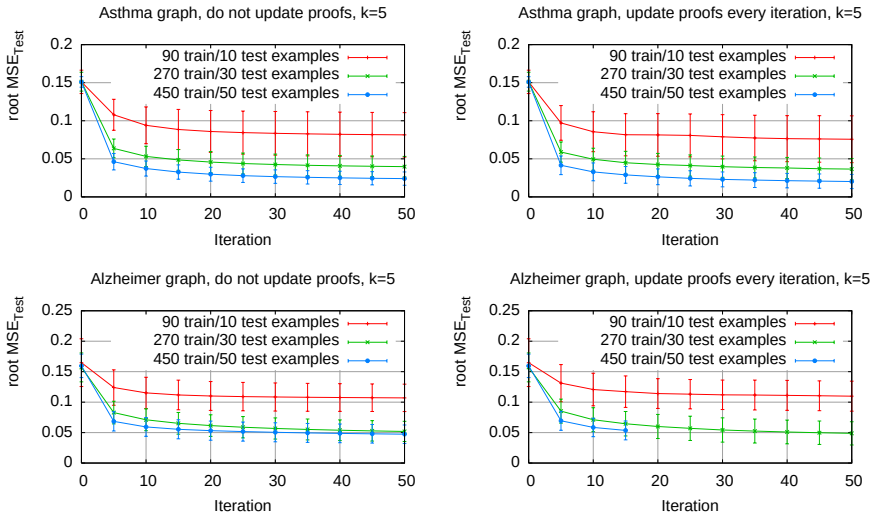

Fig. 4. $\sqrt{MSE_{\text{Test}}}$ for asthma and Alzheimer using the 5 best proofs ($k = 5$); when the BDDs and proofs are not updated (left column); when they are updated every iteration (right column) (**Q2** and **Q3**)

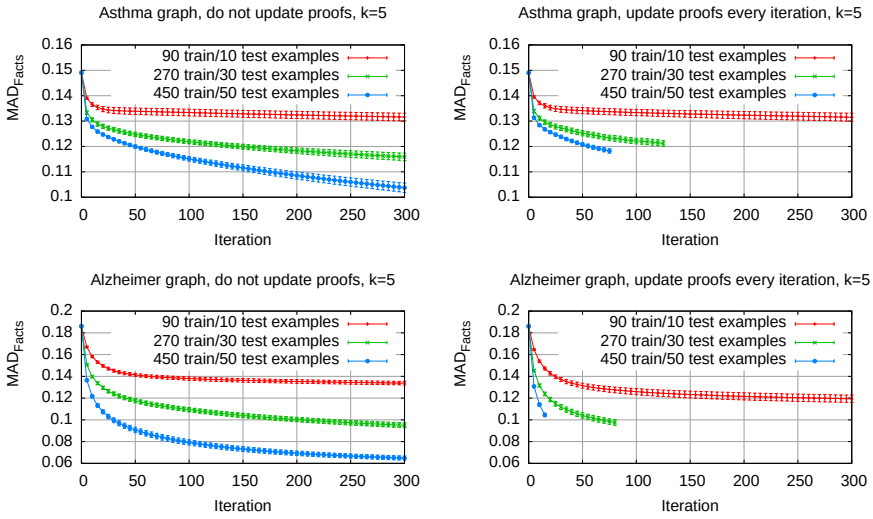

Fig. 5. MAD_{facts} for asthma and Alzheimer using the 5 best proofs ($k = 5$); when the BDDs and proofs are not updated (left column); when they are updated every iteration (right column) (**Q2** and **Q3**)

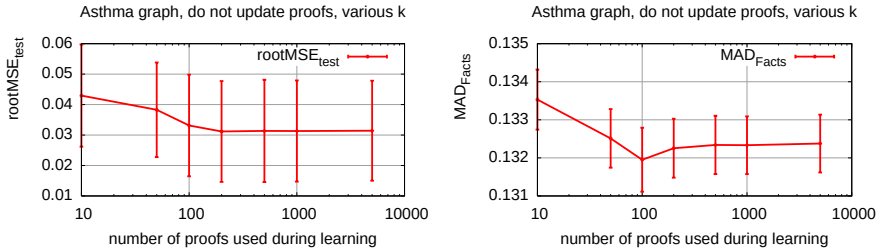

Fig. 6. MAD_{facts} and $\sqrt{MSE_{\text{Test}}}$ after 50 iterations for different k (number of best proofs used) on the asthma graph where training examples carry P_∞ probabilities (**Q4**)

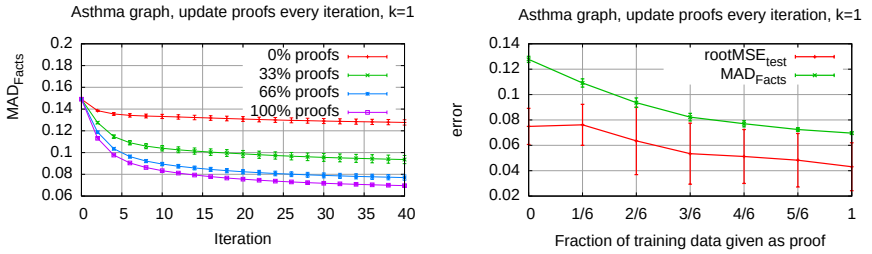

Fig. 7. MAD_{facts} and $\sqrt{MSE_{\text{Test}}}$ after 40 iterations on the asthma graph when different fractions of the data are given as proof (**Q5**)

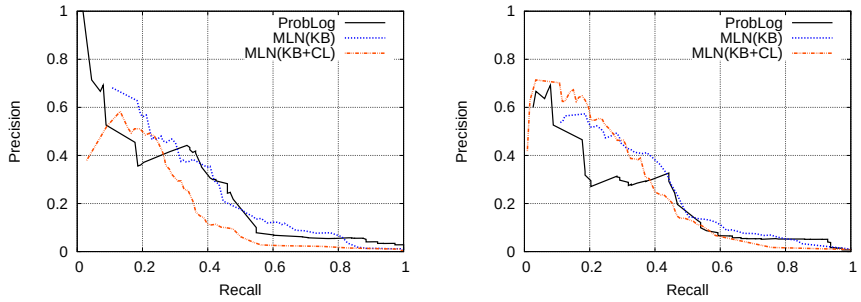

Fig. 8. Precision and recall for all areas: All Info (left) and Partial Info (right) (**Q6**).

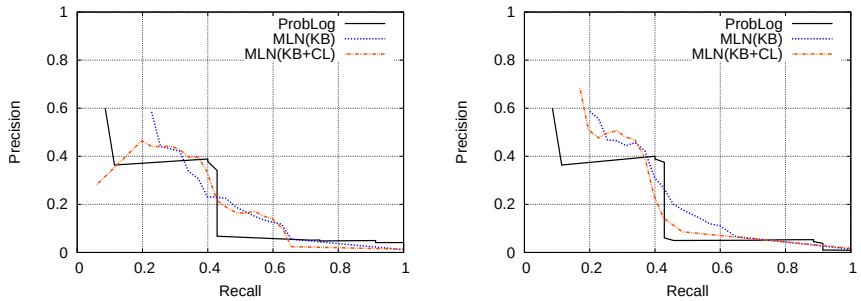

Fig. 9. Precision and recall for the AI area: All Info (left) nnd Partial Info (right) (**Q6**).

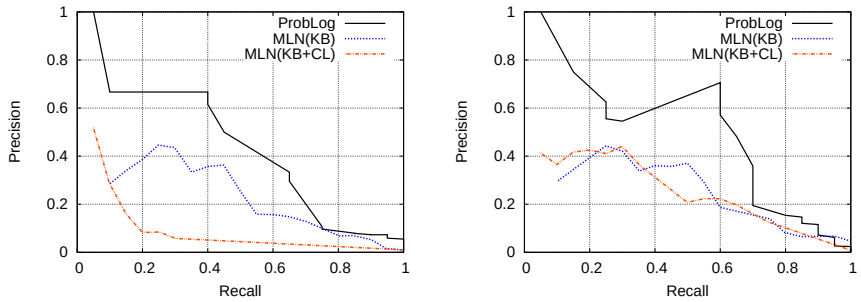

Fig. 10. Precision and recall for the graphics area: All Info (left) and Partial Info (right) (**Q6**).

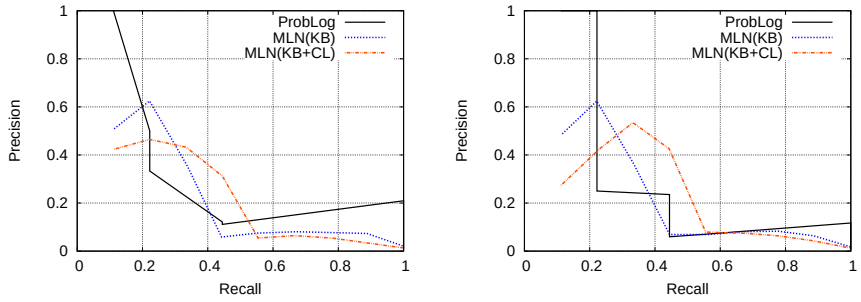

Fig. 11. Precision and recall for the programming languages area: All Info (left) and Partial Info (right) (**Q6**).

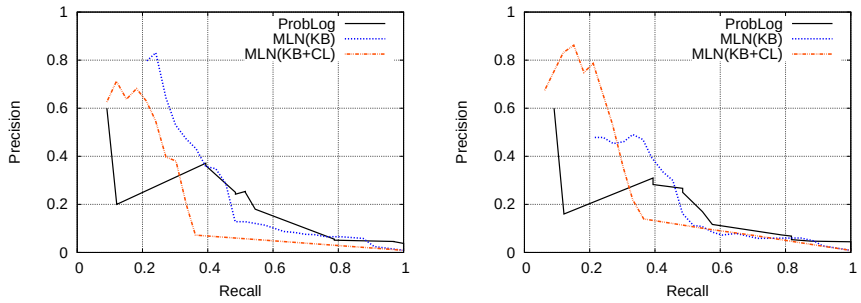

Fig. 12. Precision and recall for the systems area: All Info (left) and Partial Info (right) (**Q6**).

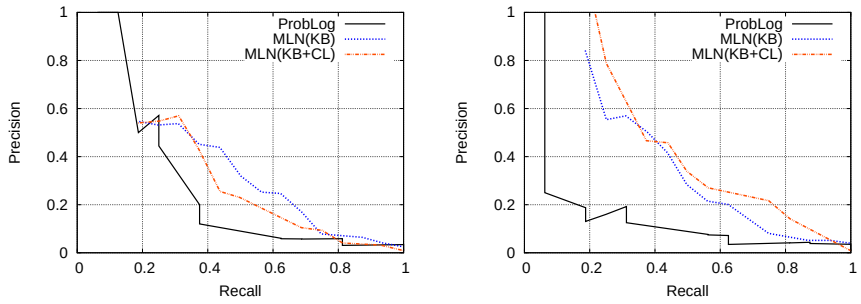

Fig. 13. Precision and recall for the theory area: All Info (left) and Partial Info (right) (**Q6**).

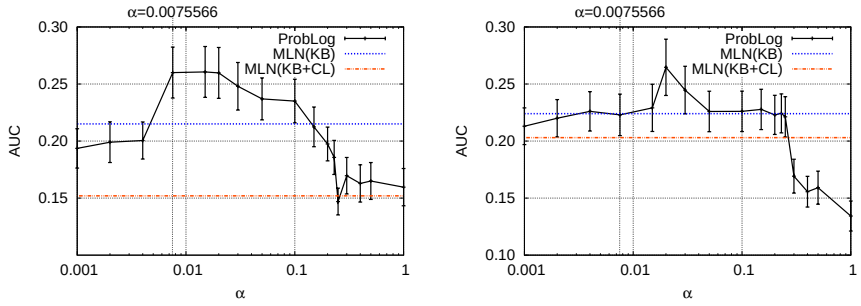

Fig. 14. The area under the Precision-Recall curve (AUC) for different α values when everything is available (All Info, left graph) and when `student(X)` and `professor(X)` are unknown (Partial Info, right graph). The horizontal lines indicate the best results obtained with MLN(KB) and MLN(KB+CL) reported in (Richardson and Domingos, 2006) (**Q6**).

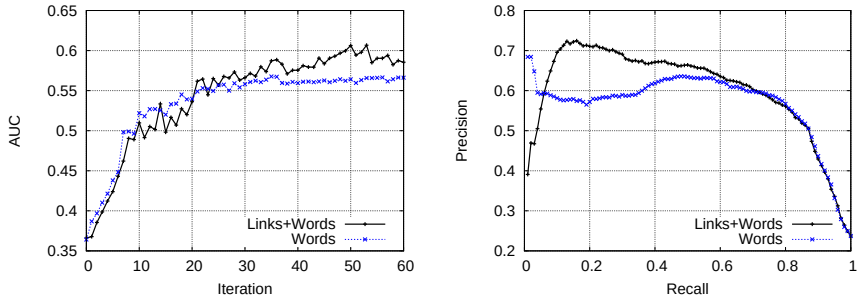

Fig. 15. Area under the Precision-Recall curve (AUC) after each learning step for WebKB (left), Precision-Recall curve after 50 iterations of gradient descent (right) (**Q6**)

Fig. 16. The two most simple BDDs, (left) the 1-terminal represents the constant function `true`, and (right) the 0-terminal represents the constant function `false`.

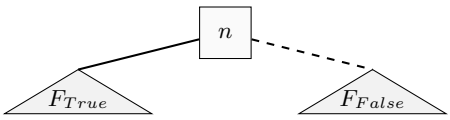

Fig. 17. The inductive case. The function F is represented as node with two sub-BDDs.

Bibliography

Antanas, L.-A., Thon, I., van Otterlo, M., Landwehr, N., and De Raedt, L. (2009). Probabilistic logical sequence models for video. In *19th International Conference on Inductive Logic Programming (ILP 2009)*.

Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer-Verlag New York.

Bryant, R. E. (1986). Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691.

Charniak, E. (1996). Tree-bank grammars. In *AAAI/IAAI, Vol. 2*, pages 1031–1036.

Chen, J., Muggleton, S., and Santos, J. (2008). Learning probabilistic logic models from probabilistic examples. *Machine learning*, 73(1):55–85.

Craven, M. and Slattery, S. (2001). Relational learning with statistical predicate invention: Better models for hypertext. *Machine Learning*, 43(1/2):97–119.

Cussens, J. (2001). Parameter estimation in stochastic logic programs. *MLJ*, 44(3):245–271.

Dalvi, N. N. and Suciu, D. (2004). Efficient query evaluation on probabilistic databases. In *Proceedings of VLDB*, pages 864–875.

Darwiche, A. (2002). A logical approach to factoring belief networks. In *Proceedings of KR*, pages 409–420.

De Raedt, L. (2008). *Logical and Relational Learning*. Cognitive Technologies. Springer Berlin Heidelberg.

De Raedt, L., Frasconi, P., Kersting, K., and Muggleton, S., editors (2008a). *Probabilistic Inductive Logic Programming - Theory and Applications*, volume 4911 of *LNAI*. Springer.

De Raedt, L. and Kersting, K. (2003). Probabilistic Logic Learning. *ACM-SIGKDD Explorations: Special issue on Multi-Relational Data Mining*, 5(1):31–48.

De Raedt, L., Kersting, K., Kimmig, A., Revoredo, K., and Toivonen, H. (2008b). Compressing probabilistic prolog programs. *Machine Learning*, 70(2-3):151–168.

De Raedt, L., Kersting, K., and Torge, S. (2005). Towards learning stochastic logic programs from proof-banks. In *AAAI*, pages 752–757.

De Raedt, L., Kimmig, A., and Toivonen, H. (2007). ProbLog: A probabilistic Prolog and its application in link discovery. In Veloso, M., editor, *IJCAI*, pages 2462–2467.

Friedman, N. (1997). Learning belief networks in the presence of missing values and hidden variables. In *ICML '97: Proceedings of the Fourteenth International Conference on Machine Learning*, pages 125–133, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Friedman, N., Getoor, L., Koller, D., and Pfeffer, A. (1999). Learning probabilistic relational models. In *IJCAI*, pages 1300–1309.

Getoor, L. and Taskar, B., editors (2007). *Statistical Relational Learning*. The MIT press.

Gupta, R. and Sarawagi, S. (2006). Creating probabilistic databases from information extraction models. In *VLDB*, pages 965–976.

Gutmann, B., Kimmig, A., De Raedt, L., and Kersting, K. (2008). Parameter learning in probabilistic databases: A least squares approach. In Daelemans, W., Goethals, B., and Morik, K., editors, *Proceedings of the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD 2008), Part I*, volume 5211 of *LNCS (Lecture Notes In Computer Science)*, pages 473–488, Antwerp, Belgium. Springer Berlin/Heidelberg.

Ishihata, M., Kameya, Y., Sato, T., and ichi Minato, S. (2008). Propositionalizing the EM algorithm by BDDs. In Železný, F. and Lavrač, N., editors, *Proceedings of Inductive Logic Programming (ILP 2008), Late Breaking Papers*, pages 44–49, Prague, Czech Republic.

Kersting, K. and De Raedt, L. (2008). Basic principles of learning bayesian logic programs. In De Raedt, L., Frasconi, P., Kersting, K., and Muggleton, S., editors, *Probabilistic Inductive Logic Programming - Theory and Applications*, volume 4911 of *Lecture Notes in Computer Science*, pages 189–221. Springer.

Kimmig, A., De Raedt, L., and Toivonen, H. (2007). Probabilistic explanation based learning. In *ECML*, pages 176–187.

Kwoh, C.-K. and Gillies, D. (1996). Using hidden nodes in bayesian networks. *Artificial Intelligence*, 88(1-2):1–38.

Lloyd, J. W. (1989). *Foundations of Logic Programming*. Springer, Berlin, 2. edition.

Lowd, D. and Domingos, P. (2007). Efficient weight learning for markov logic networks. In Kok, J. N., Koronacki, J., de Mántaras, R. L., Matwin, S., Mladenic, D., and Skowron, A., editors, *Knowledge Discovery in Databases: PKDD 2007*, volume 4702 of *Lecture Notes in Computer Science*, pages 200–211. Springer.

Nottelmann, H. and Fuhr, N. (2001). Learning probabilistic datalog rules for information classification and transformation. In *CIKM*, pages 387–394. ACM.

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

Poole, D. (2008). The independent choice logic and beyond. In De Raedt, L., Frasconi, P., Kersting, K., and Muggleton, S., editors, *Probabilistic Inductive Logic Programming - Theory and Applications*, volume 4911 of *Lecture Notes in Computer Science*, pages 222–243. Springer.

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

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

Sato, T. (1995). A statistical learning method for logic programs with distribution semantics. In Sterling, L., editor, *International Conference on Logic Programming*, pages 715–729. MIT Press.

Sato, T. and Kameya, Y. (2001). Parameter learning of logic programs for symbolic-statistical modeling. *J. Artif. Intell. Res. (JAIR)*, 15:391–454.

Saul, L., Jaakkola, T., and Jordan, M. (1996). Mean field theory for sigmoid belief networks. *JAIR*, 4:61–76.

Sevon, P., Eronen, L., Hintsanen, P., Kulovesi, K., and Toivonen, H. (2006). Link discovery in graphs derived from biological databases. In Leser, U., Naumann, F., and Eckman, B. A., editors, *Proceedings 3rd International Workshop on Data Integration in the Life Sciences (DILS 2006)*, volume 4075 of *Lecture Notes in Computer Science*, pages 35–49. Springer.

Wrobel, S., Wettschereck, D., Sommer, E., and Emde, W. (1996). Extensibility in data mining systems. In *KDD*, pages 214–219.