KATHOLIEKE UNIVERSITEIT
LEUVEN

Arenberg Doctoral School of Science, Engineering & Technology
Faculty of Engineering
Department of Computer Science

On Continuous Distributions and Parameter Estimation in Probabilistic Logic Programs

Bernd GUTMANN

Dissertation presented in partial
fulfillment of the requirements for
the degree of Doctor
in Engineering

October 2011

On Continuous Distributions and Parameter Estimation in Probabilistic Logic Programs

Bernd GUTMANN

Jury:
Prof. Dr. ir. Willy Sansen, president
Prof. Dr. Luc De Raedt, promotor
Prof. Dr. ir. Maurice Bruynooghe
Prof. Dr. ir. Tom Holvoet
Prof. Dr. Vítor Santos Costa
(Universidade do Porto, Portugal)
Prof. Dr. rer. nat. Manfred Jaeger
(Aalborg Universitet, Denmark)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor in Engineering

October 2011

D/2011/7515/123
ISBN 978-94-6018-422-2

Abstract

In the last decade remarkable progress has been made on combining statistical machine learning techniques, reasoning under uncertainty, and relational representations. The branch of Artificial Intelligence working on the synthesis of these three areas is known as statistical relational learning or probabilistic logic learning.

ProbLog, one of the probabilistic frameworks developed, is an extension of the logic programming language Prolog with independent random variables that are defined by annotating logical facts with probabilities. The separation of the logical and probabilistic part of the model is based on the distribution semantics. Driven by the demand for models that are able to handle continuous values and can be automatically optimized on training data, this thesis introduces several algorithms and extensions to the ProbLog language. Continuous-valued data arise naturally in robotics, human activity recognition and bio-medical applications. Moreover, the models used are complex and the available data is often noisy and incomplete. Hence tuning a model towards the specifics of the environment can hardly be done manually. This poses two crucial challenges for probabilistic programming languages such as ProbLog: processing continuous values and being able to learn from training data.

This thesis makes four main contributions to the field of probabilistic logic learning. *Hybrid ProbLog* is an extension for ProbLog with continuous facts that allows for exact inference. *Distributional Programs* combine elements of ProbLog, Hybrid ProbLog and CP-Logic into a very expressive language for dealing with continuous distributions. A sampling-based inference algorithm is used to answer conditional queries, while the deterministic information in the program guides the sampling process. *LFE-ProbLog* is able to learn the parameters of a ProbLog program from queries and proofs, while *LFI-ProbLog* is optimized to learn the parameters from partial interpretations. Together they cover the standard learning settings considered in PLL. All learning approaches have been evaluated in several relational real-world domains.

Acknowledgements

First and foremost I would like to express my sincere gratitude to my supervisor Luc De Raedt for his advice and encouragement. He has guided me through my Ph.D. studies, was continuously suggesting interesting research questions and giving me the freedom to find my own answers. Despite his amazingly tight schedule he has always found the time to help me with urgent questions and tricky problems. Before major deadlines, he would even take the time for Skype meetings after 22:00 on Saturday nights! Luc taught me that in science, seeking the right questions is at least as important as finding good answers. Bedankt! Back in my student days in Freiburg, it was Kristian Kersting who got me interested in Statistical Relational Learning. He was the daily supervisor of my diploma thesis and later guided me during the beginning of my Ph.D. work. He is a walking library of machine learning papers. Talking to him for ten minutes is comparable to reading ten papers :-)
I enjoyed many interesting discussions with him, where I obtained new ideas and insights. Sometimes I still wonder how exactly Luc and Kristian got me interested in logic programming... But they most certainly did!

I would like to thank the members of my jury for their time and helpful suggestions on the text. I also thank Willy Sansen for chairing the jury.

Both Laura Antanas and Ingo Thon have read parts of the manuscript and helped me improving the text. I especially would like to thank Laura for her efforts on running experiments using the TildeCRF implementation.

I thank Niels Landwehr for his help and especially for pointing me to logistic regression. Working on the "Rock Paper Scissors" project together with Niels, Wannes Meert and Ingo was a fun experience and welcome distraction from "real work". I greatly enjoyed working on LFI-ProbLog together with Ingo and our discussions about life, the universe and the rest.

I thank Manfred Jaeger for co-authoring the Hybrid ProbLog paper. His comments on the semantics and his dedication to the detail (finding errors at the 4th level of the subindices!) helped making this paper more readable. I also thank Maurice Bruynooghe for co-authoring our ICLP paper on distributional programs, where

his vast knowledge on logic programming significantly helped enriching the text.

I thank my colleagues Angelika Kimmig, Daan Fierens and Theofrastos Mantadelis for many interesting discussions and stimulating and pleasant atmosphere in our office. I am glad that Davide Nitti and Thanh Le Van have taken challenge of using distributional programs and that Davide volunteered to implement a stable version of our inference algorithm.

Of course there are many more people I am indebted to thank. Unfortunately, there is not enough space left (and in particular time) to mention all of them. Basically I want to thank the whole Machine learning group in Leuven who made working there very enjoyable and the constant feedback and assistance I received helped me finishing and succeeding at my Ph.D. studies. Also, I would like to express my gratitude to the people in the background, such as the secretaries and the system group, who kept everything running swiftly.

Prolog is not just another programming language :-) Lucky for me, there is Vítor Santos Costa who is very quick when it comes to providing any kind of assistance around the YAP Prolog system. Feature requests, bug reports, general Prolog hardships or incompatibilities due to major changes in Mac OS – you name it – he normally sends the solution within an hour. Paulo Moura helped us improving the runtime performance of LFI-ProbLog. Like Vítor he seems to be gifted with the ability to make any Prolog program at least 20% faster just by "looking at it".

I spent a great summer at Intel research Seattle where I learned a lot while working with Matthai Philipose on human activity recognition. Thank you for this unique opportunity.

I gratefully acknowledge the financial support received for the work performed during my thesis from the European Union FP6-508861 project on Applications of Probabilistic Inductive Logic Programming II, the GOA/08/008 Probabilistic Logic Learning and the Research Foundation Flanders (FWO-Vlaanderen).

Des Weiteren möchte ich mich bei meinen Eltern für ihre Unterstützung bedanken.

I would like to take this opportunity to thank my wife Maria for her love, encouragement and unfailing optimism. Her support during the last five years and particularly the help with proofreading this thesis is invaluable. I also want to apologize for the collateral damage this caused, namely her getting fluent in "SRL lingo" and using words like "atoms" and "axioms" in casual conversations.
Маша, огромное тебе спасибо!

Bernd Gutmann
Heverlee, September 2011

Contents

List of Symbols

BK	Background knowledge, page 22
$\mathrm{dep}_T(I)$	Dependency set of the interpretation I in the theory T, page 119
$\mathrm{dep}_T(x)$	Dependency set of the atom x in the theory T, page 119
F	Set of probabilistic facts, page 22
F^c	Set of continuous facts, see equation (4.1)
F^T	Maximal set of ground probabilistic facts that can be generated from F in T, see equation (3.3)
L^T	Maximal set of ground facts that can be generated from F in T, see equation (3.2)
$\mathcal{A}, \mathcal{B}, \mathcal{C}$	Different partitions of $\mathbb{R}^n$, page 43
$T = F \cup BK$	ProbLog theory, page 22
$T^r(I)$	Interpretation-restricted ProbLog theory of T given the interpretation I, page 119
$I = (I^+, I^-)$	A partial interpretation, page 117
$p_n :: f_n$	Probabilistic fact, page 22
$\sigma_s(a)$	Sigmoid function, see equation (6.2)
$\varphi_{\mu;\sigma}(x)$	Probability density function of a Gaussian with mean μ and standard deviation σ, see equation (4.2)
$\delta^+_{j,L}$	Number of true ground instances of the probabilistic fact f_j in the possible world L, see equation (6.3)
$\delta^-_{j,L}$	Number of false ground instances of the probabilistic fact f_j in the possible world L, see equation (6.4)

$\mathrm{MSE}(T)$	Mean squared error, see equation (6.1)
$\mathrm{MSE}_{\mathrm{cost}}(T)$	Mean squared error with an associated cost function, see equation (6.9)
$P^T(L)$	Probability of the subprogram $L \subseteq L^T$, see equation (3.4)
$P^T(I)$	Probability of partial interpretation, see equation (3.5)
$P_w^T(LH(L))$	Probability of the possible world associated to L, page 23
$P_s^T(q)$	Success probability of query, see equation (3.6)
$P_k^T(q)$	Probability of k best proofs of q, see equation (3.9)
$P^T(X \in I)$	Probability of the continuous subprogram, see equation (4.3)

List of Figures

List of Tables

List of Algorithms

Part I

Foundations

Chapter 1

Introduction

Artificial Intelligence, or simply AI, is concerned with building systems that are able to act rationally [Russell and Norvig, 2003]. While the early AI researchers' dream – to create systems with human-level intelligence – seems to be out of reach, remarkable progress has been made on solving individual subtasks such as image recognition, voice recognition, and planning to name only few. The Stanley project [Thrun et al., 2006], which resulted in an autonomously driving car that was able to navigate through unknown off-road terrain and won the DARPA Grand Challenge, illustrates this. Another example is Watson [Ferrucci et al., 2011], a computer system that has beaten the reigning human champion in a TV quiz show. To facilitate research and make results usable across different application domains, AI uses the concept of so-called *rational agents* to characterize systems that act rationally. This allows for studying individual subtasks, such as speech recognition, face recognition or planning, and combining them later. In this thesis, we focus on two key requirements of such a rational agent: knowledge representation and the ability to learn.

In order to make rational decisions, an agent needs to possess knowledge, which in turn needs to be stored such that it is accessible and can be used for *reasoning*. Knowledge is stored in form of a so-called *knowledge representation language*, e.g., a variant of first order logic. In order to account for the uncertainty in the environment, such a language needs to provide constructs for representing probabilistic information. A system like Watson, for example, needs to know that there is a city "Toronto" in Canada and a city "Toronto" in the USA. However, when somebody says "Toronto" it is more likely that they refer to the city in Canada than to the one in the USA. Probabilistic logic languages offer a combination of logic with probabilities, which in turn allow one to quantify information with probabilities that can be used in the reasoning process.

Another requirement for rational agents is the ability to learn, that is, increasing their performance on a given task with experience. Complex tasks that exceed the capabilities of traditional handwritten algorithms, such as face recognition, and domains that can change over time, such as Spam filtering, are best solved using machine learning methods [Mitchell, 1997; Bishop, 2006]. Often it is easier to generate a set of *training examples* than to develop a full model for a particular task. In case of the Spam filtering task, a training set consists of emails labeled as "wanted" or "unwanted". And in the case of face recognition the training set might contain images of persons annotated with the position of the face on the picture as well as the name of the person shown. In this thesis we focus on *parameter learning*, that is, the model structure is given while the model parameters are unknown and have to be estimated using the training examples.

Statistical relational learning [Getoor and Taskar, 2007; De Raedt et al., 2008a] is a branch of AI, that combines the three above mentioned fields: logic is used to model dependencies between elements, probabilities represent uncertainty in the domain and learning adds the power to tune the resulting models towards the specifics of the environment. The advent of statistical relational learning and probabilistic programming has resulted in a vast number of different representation languages and systems. While all of them use a relational or logical language for expressing dependencies, they deviate in how they make use of this information.

To begin with, systems based on the principle of *knowledge-based model construction* use the logical part of the model to create a so-called *grounded model*, which in turn is used by the inference mechanism to compute probabilities of queries. For instance, BLPs [Kersting and De Raedt, 2008], LBNs [Fierens, 2010], RBNs [Jaeger, 1997] and CLP($\mathcal{BN}$) [Santos Costa et al., 2008] generate Bayesian networks [Pearl, 1988] upon grounding the model for answering a particular query. Similar principles are employed by Markov logic networks [Richardson and Domingos, 2006], relational dependency networks [Neville and Jensen, 2007] and PRMs [Friedman et al., 1999a].

Probabilistic programming languages emphasize the programming aspect of the model, that is, they interleave the program execution with sampling random events. Sato's [1995] distribution semantics is used by many logic-programming-based languages such as PRISM [Sato, 1995], ProbLog [De Raedt et al., 2007; Kimmig, 2010] and CP-Logic [Vennekens et al., 2006]. The core of this semantics lies in splitting the model into a probabilistic and a logical part and using the reasoning mechanism of the underlying host language, typically Prolog, to combine them. Furthermore, there are also probabilistic languages such as BLOG [Milch et al., 2005a], Church [Goodman et al., 2008], IBAL [Pfeffer, 2001] and Figaro [Pfeffer, 2009] that are based on a functional paradigm.

Thesis Contributions and Roadmap

In this thesis we study the following two questions:

> (Q1) How can continuous distributions be integrated in probabilistic logic programming languages?
>
> (Q2) How can the parameters of probabilistic programming languages be estimated?

We use ProbLog [De Raedt et al., 2007; Kimmig, 2010] as representation language, even though the techniques developed here are also applicable to other probabilistic programming languages. This thesis makes four main contributions with respect to the above mentioned questions: (1) an extension of ProbLog with continuous distributions along with an exact inference algorithm, (2) a more expressive language based on concepts from ProbLog and CP-logic together with an optimized sampling algorithm, (3) a parameter estimation algorithm for ProbLog that is able to learn from queries and proofs and (4) a parameter estimation algorithm for ProbLog that learns from partial interpretations.

Continuous-valued information is critical in many real-world applications. However, probabilistic programming languages based on Sato's [1995] distribution semantics have not yet been extended towards continuous distributions. Specifically, they employ exact inference techniques that are limited to finite distributions. On the contrary, probabilistic languages based on a functional paradigm, such as Church [Goodman et al., 2008], IBAL [Pfeffer, 2001], BLOG [Milch et al., 2005a], do support continuous distributions but not exact inference. One direction we follow in this thesis, is the extension of ProbLog's exact inference algorithm with continuous distributions. Being able to perform exact inference requires the underlying representation language to be restrictive, which in turn does limit its applicability to real-world tasks. This has motivated the work on distributional programs, where we aim at increasing the expressivity of the language without imposing restrictions on the use of continuous values. This allows one, for instance, to specify a model where the parameters of distributions are initialized with sampled values from another distribution, hence making inference in a *Bayesian* setting possible. In turn, this requires an inference algorithm based on sampling.

Another question is how to estimate the parameters of a ProbLog program and how the standard learning settings known in statistical relational learning be integrated into ProbLog. Without parameter learning the designer of a model has to set the parameters manually. Apart from the extra effort, this process can lead to sub-optimal models and in some cases it is impossible to decide on good parameters. Parameter learning supports the designer and lets them focus on the structural part of the model, for instance, the dependency between random variables.

By integrating learning techniques in ProbLog we enable many applications, as providing training examples is typically easier than finding good model parameters.

The thesis is divided into four parts: discussing preliminaries of ProbLog, extensions of ProbLog with continuous distributions, learning techniques to estimate the parameters of ProbLog programs, and the conclusions as well as several appendices with proofs. In the following, we give an outline of each part.

Part I introduces basic concepts and existing work that are required in the course of this thesis. **Chapter 2** discusses the terminology and concepts from logic programming that form the basis of the probabilistic programming language ProbLog. Moreover, it reviews Binary Decision Diagrams (BDDs) that are used by ProbLog's inference algorithms. **Chapter 3** reviews ProbLog, a probabilistic extension of the logic programming language Prolog.

Part II is about extending ProbLog with continuous distributions. In **Chapter 4** we introduce continuous facts and define a semantics that is consistent with Sato's [1995] distribution semantics. Our approach, called *Hybrid ProbLog*, uses an interval calculus to extract relevant intervals and performs *dynamic discretization*, that is, the discretization depends on the query to be evaluated. The resulting discretization is optimal, that is, any finer discretization would yield the same result but would require more effort to be evaluated. In **Chapter 5** we introduce distributional programs, which are more expressive and better suited for real-world applications. We propose a modified rejection sampling algorithm that uses the well-known magic sets transformation to restrict the size of the samples. Moreover, the deterministic dependencies in the program are exploited by a lookahead step to increase the number of samples consistent with the evidence. The work presented in Part II has been previously published in:

> B. Gutmann, M. Jaeger, and L. De Raedt. *Extending ProbLog with continuous distributions.* In P. Frasconi and F. Lisi, editors, *Proceedings of the 20th International Conference on Inductive Logic Programming (ILP-10)*, volume 6489 of LNCS (Lecture Notes in Computer Science), pages 76–91. Springer Berlin / Heidelberg, 2011. DOI: 10.1007/978-3-642-21295-6_12

> B. Gutmann, I. Thon, A. Kimmig, M. Bruynooghe, and L. De Raedt. *The magic of logical inference in probabilistic programming.* Theory and Practice of Logic Programming, 11:663–680, 2011. DOI: 10.1017/S1471068411000238

Part III proposes two parameter learning approaches for ProbLog. They differ both in the format of training examples and in the underlying principle that guides optimization during learning. The first approach, presented in **Chapter 6**,

learns from training examples given as *annotated queries*, that is atoms or conjunctions of atoms together with a *target probability*. It treats the learning problem as a regression task, similar to logistic regression. The second approach, presented in **Chapter 7**, learns from training examples given in the form of *partial interpretations*, that is, sets of atoms that state the truth value for some (or all) atoms in a logic program. The work presented in Part III has been previously published in:

> B. Gutmann, A. Kimmig, K. Kersting, and L. De Raedt. *Parameter learning in probabilistic databases: A least squares approach.* In W. Daelemans, B. Goethals, and K. Morik, *Proceedings of the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD 2008)*, volume 5211 of LNCS (Lecture Notes In Computer Science), pages 473–488, September 2008. Springer Berlin/Heidelberg. DOI: 10.1007/978-3-540-87479-9_49
>
> B. Gutmann, A. Kimmig, K. Kersting, and L. De Raedt. *Parameter estimation in ProbLog from annotated queries.* Technical Report CW 583, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, April 2010.
>
> B. Gutmann, I. Thon, and L. De Raedt. *Learning the Parameters of Probabilistic Logic Programs from Interpretations.* In D. Gunopulos, T. Hofmann, D. Malerba, and M. Vazirgiannis, *European Conference on Machine Learning and Principles and Practices of Knowledge Discovery in Databases (ECML PKDD 2011)*, volume 6911 of LNCS (Lecture Notes in Computer Science), pages 581–596. Springer Berlin/Heidelberg, 2011. **Winner of the Best Paper Runner up Award in Machine Learning** (599 submissions). DOI: 10.1007/978-3-642-23780-5_47

Part IV concludes the thesis and discusses possible directions for future work. Additionally, it contains several appendices with proofs, which were omitted in the individual chapters for the ease of reading.

Finally, some of work that was performed during my Ph.D. research has not been included in this thesis. It is briefly summarized below.

Conditional Random Fields (CRFs) are a class of undirected graphical models that are used for classification tasks in natural language domains such as part-of-speech tagging. Based on gradient tree boosting [Friedman, 2001; Dietterich et al., 2004] and the results of my diploma thesis [Gutmann, 2005], we developed a relational extension of CRFs called TildeCRF that is able to tag sequences of logical atoms [Gutmann and Kersting, 2006]. We studied several modifications

of the learning algorithm including conjugated gradient search [Kersting and Gutmann, 2006] and a stratification of the training set that is used for the internal regression models [Gutmann and Kersting, 2007]. We successfully applied TildeCRF on different biological tasks such as protein secondary structure prediction and protein classification [Gutmann and Kersting, 2006], as well as on human activity recognition tasks based on RFID tags [Landwehr et al., 2008] and on video [Antanas et al., 2009]. Lastly, we also adopted the technique of relational gradient tree boosting for learning relational dependency networks [Natarajan et al., 2011], where we obtained a significant improvement of the predictive accuracy compared to the state-of-the-art frameworks.

Implementation

The algorithms for exact inference in Hybrid ProbLog, as well as the two parameter learning algorithms LFI-ProbLog and LFE-ProbLog are implemented and integrated in the ProbLog system, which is open source and available at `http://dtai.cs.kuleuven.be/problog/`. The BDD algorithms are implemented using SimpleCUDD [Mantadelis et al., 2008] and CUDD [Somenzi, 2009]. For our experiments we used the YAP Prolog system [Santos Costa et al., 2011]. The sampling-based inference for distributional programs has been implemented as proof-of-concept.

Chapter 2

Preliminaries

This chapter reviews existing work and provides the relevant background on the core concepts used in the course of this thesis. Section 2.1 reviews logic programming that serves as representation language, while Section 2.2 discusses Binary Decision Diagrams that are used by the inference and learning algorithms. Section 2.3 introduces the relevant concepts from probability theory. We conclude by discussing the key ideas of the distribution semantics in Section 2.4.

2.1 Logic Programming

Logic programming is a declarative programming paradigm based on logic. In this section we review the terminology and introduce basic concepts needed later on in this thesis. Prolog is one example of a logic-programming-based language. For a detailed introduction to logic programming please refer to [Lloyd, 1984; Flach, 1994; Nilsson and Małuszyński, 1995], and for a detailed discussion of Prolog see [Sterling and Shapiro, 1994].

Example 2.1. *The following program consists of four parts. The first part enumerates all days starting from day 0, to s(0), s(s(0)) and so on. The second part specifies that there are two people, John, who wants to eat steak or spaghetti, and Mary, who wants to eat fish. The third part specifies that there is enough steak in the fridge for three days, whereas the fish is only available on the first day. Also, there is an infinite supply of spaghetti available. The last part of the program states that if a person wants to eat something that is available they will get it.*

```
day(0).
day(s(X)) :- day(X).

wants(john, steak).
wants(john, spaghetti).
wants(mary, fish).

available(steak, 0).
available(steak, s(0)).
available(steak, s(s(0))).
available(fish, 0).
available(spagetthi, Day) :- day(Day).

gets(Person, Product, Day) :- wants(Person, Product),
                              available(Product, Day).
```

2.1.1 Syntax

The symbols used in the program in Example 2.1 can be grouped into the following categories. The symbols `0`, `john` and `steak` are *constants*. Uppercase symbols, such as `X`, `Day`, `Person`, and `Product` denote *variables*. The symbol `s` is a functor of arity 1. The symbol `day` is a predicate symbol of arity 1, while `wants` is a predicate symbol of arity 2. Symbols can be recursively combined to *terms* as follows:

- each constant c is a term
- each variable X is a term
- If $t_1, \ldots, t_n$ are terms and f is an n-ary functor, then $f(t_1, \ldots, t_n)$ is a term.

If a is an n-ary predicate and $t_1, \ldots, t_n$ are terms, then $a(t_1, \ldots, t_n)$ is an *atom*. An atom is also called a *positive literal*, while the negation of an atom, for instance `not(day(0))`, is called a *negative literal*.

A *clause* is a disjunction of literals, for instance $i_1 \vee i_2 \vee \neg i_3$. A clause with exactly one positive literal h is called a *definite clause*. In Prolog notation, such clauses are denoted by

$$h :- b_1, \ldots, b_n.$$

The atom h is called the *head* of the clause and $b_1, \ldots, b_n$ is the *body*. The intuitive meaning of the symbol :− is that of the implication, that is, when the body of

the clause is true, the head of the clause is also true. If the body is empty, that is, the clause consists of one positive literal, then it is called a *fact*. The intuitive meaning of a fact is, that it is always *true*. The Prolog notation for facts is the atom followed by a period:

$h.$

A set P of definite clauses is called a *logic program*. Terms, atoms and clauses are called *ground* if they do not contain variables. For instance, the atom $day(0)$ is ground and $s(X)$ is non-ground. A clause is *range-restricted* if all variables from the head appear in the body of the clause, i.e., $\text{var}(h) \subseteq \text{var}((b_1, \ldots, b_n))$, where $\text{var}(\cdot)$ is the function that maps a term onto the set of variables occurring in the term.

We will adopt Prolog's notation when describing logic programs. We use a `fixedfont` to mark parts that should be read as Prolog code. Atoms, functors and symbols start with lowercase letters, while variables start with uppercase letters. The period marks the end of the clause, while :− separates head and body of the clause.

A *substitution* $\theta = \{V_1/t_1, \ldots, V_m/t_m\}$ is a mapping of variables to terms. When applying a substitution onto t, denoted by $t\theta$, each variable in t is replaced simultaneously by the corresponding term.

Example 2.2. *The substitution* {`Person`/`john`, `Product`/`steak`, `Day`/`s(s(0))`} *applied on* `gets(Person, Product, Day)` *yields* `gets(john, steak, s(s(0)))`.

One can apply substitutions on atoms, terms and clauses. Substitutions do not necessarily have to map all variables as well as mapped terms do not have to be ground. Two terms t_1 and t_2 are called *unifiable* if there exist substitutions θ_1 and θ_2 such that $t_1\theta_1 = t_2\theta_2$.

Definition 2.1 (Most General Unifier)**.** *A substitution θ is a* most general unifier *of a and b, denoted by $\theta = mgu(a, b)$, if and only if $a\theta = b\theta$ and for each substitution θ' such that $a\theta' = b\theta'$, there exists a substitution γ such that $\theta' = \theta\gamma$.*

Example 2.3. *A most general unifier of the atom* `gets(Person, steak, Day)` *and* `gets(P, Product, s(s(0)))` *is* θ = {`Day`/`s(s(0))`, `Product`/`steak`, `P`/`Person`}.

Please note that there can exist more than one most general unifier, and each MGU yields different variable names in the term instances obtained by applying it. For instance, the substitution θ = {`Day`/`s(s(0))`, `Product`/`steak`, `Person`/`P`} is another MGU for the atoms in Example 2.3.

2.1.2 Semantics And Inference

The *Herbrand base* of a logic program P contains all ground atoms that can be constructed by the terms in P. Each subset of the Herbrand base is called a *Herbrand interpretation.*

Definition 2.2 (Herbrand Model)**.** *A Herbrand interpretation I is called a* model *of a clause h :– $b_1, \ldots b_n$ if for every grounding substitution θ, where all $b_i\theta \in I$, the head element $h\theta$ is contained in I. A Herbrand interpretation I is called a* Herbrand model *of the logic program P, if it is a model for all clauses in P. The smallest Herbrand model of P is called the* least Herbrand model *and denoted as LH(P).*

A Herbrand model captures the intuitive meaning of a logic program, that is, facts are always *true* and whenever the body of a clause is *true*, the head atom is *true* as well. The least Herbrand model captures the intuition that clauses are not implications but inductive definitions.

Example 2.4 (Least Herbrand Model)**.** *The least Herbrand model of the theory in Example 2.1 is*

{ day(0), day(s(0)), day(s(s(0))), ..., wants(john, steak),
wants(john, spagetthi), wants(mary, fish), available(steak, 0),
available(steak, s(0)), available(steak, s(s(0))), available(fish, 0),
available(spaghetti, 0), available(spaghetti, s(0)),
available(spaghetti, s(s(s0))), ..., gets(john, steak, 0),
gets(john, steak, s(0)), gets(john, steak, s(s(0))), gets(mary, fish, 0),
gets(john, spaghetti, 0), gets(john, spaghetti, s(0)),
gets(john, spaghetti, s(s(0))), ...}

A crucial inference task is to decide whether an atom a, called *query atom*, is true in the least Herbrand model of a logic program. If the atom is true in LH(P), one says *the query holds*, if not one says *the query does not hold* or, equivalently, *its negation holds.* If the query atom a is non-ground, then one is interested in some, or all, substitutions θ such that $a\theta$ is true in the least Herbrand model. In Example 2.1, for instance, the query gets(john, steak, s(0)) succeeds and gets(mary, fish, s(0)) fails. There exist different methods to determine LH(P). One approach, for instance, is to compute the least fixpoint of the *immediate consequence operator* T_P.

Definition 2.3 (Immediate Consequence Operator)**.** *Let P be a logic program and I a Herbrand interpretation of P, then the immediate consequence operator is defined as*

$$T_P(I) := \{A\theta \mid \textit{there exists } \theta \textit{ and a clause } h :- b_1, \ldots b_n \in P \textit{ such that}$$
$$h\theta :- b_1\theta, \ldots b_n\theta \textit{ is ground and } \{b_1\theta, \ldots b_n\theta\} \subseteq I\} \ .$$

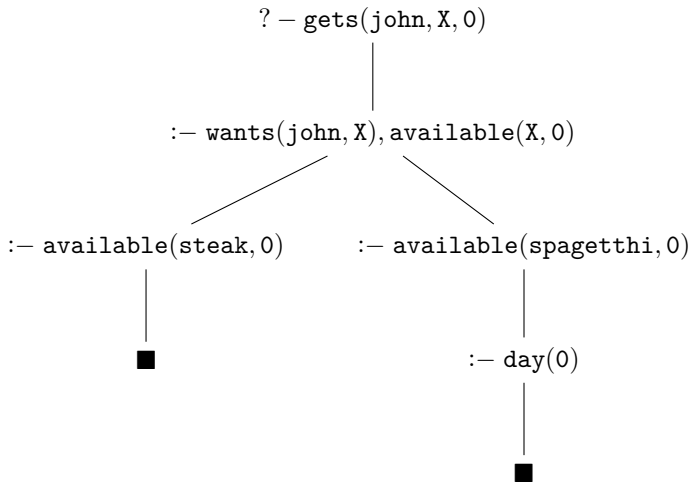


Figure 2.1: SLD tree for query gets(john, X, 0) in Example 2.1

It can be shown that the least Herbrand model of a definite logic program is the least fixpoint of T_P (cf. Nilsson and Małuszyński [1995]). For programs that have a finite least Herbrand model, i.e., definite programs with range-restricted functor-free clauses, one can compute the least Herbrand model by repeatedly applying the operator on its output until there is no change.

Computing the least Herbrand model is often impractical. In order to answer a single query one has to compute *everything* that can be derived from the program. And for programs where LH(P) is infinite an explicit computation is not feasible. Logic programming language like Prolog use a *resolution mechanism* to decide whether a query succeeds or not. In difference to computing the least Herbrand model, resolution algorithms employ *backward reasoning* starting from the goal, which in turn avoids deriving irrelevant atoms.

Definition 2.4 (SLD resolution step)**.** *Given a logic program P, a goal $:- G_1, \ldots, G_m$ and a clause $h :- b_1, \ldots b_n \in P$ such that $G_i\theta = h\theta$ where $\theta = mgu(h, G_1)$ then SLD resolution yields the new goal $:- G_1\theta, \ldots, G_{i-1}\theta, b_1\theta, \ldots, b_n\theta, G_{i+1}\theta, \ldots, G_m\theta$.*

Prolog uses a variant of SLD resolution called SLDNF. The inference algorithm uses the clauses in the order they are specified in the program, while selecting always the leftmost atom of the current goal. Moreover, it assumes *negation-as-failure* (NF), that is, the negation of a query is true if there does not exist a refutation proof for it.

A *proof* of a query a is a successful *refutation*, that is, a sequence of resolution steps starting from $:- a$ and yielding the empty goal $:-$. One says a proof *fails* if there is no sequence of resolution steps resulting in the empty goal.

Example 2.5. *One can prove the query* $\texttt{gets}(\texttt{john}, \texttt{spagetthi}, \texttt{s}(\texttt{0}))$ *in Example 2.1 as follows:*

$:-\texttt{gets}(\texttt{john}, \texttt{spagetthi}, \texttt{s}(\texttt{0}))$

$:-\texttt{wants}(\texttt{john}, \texttt{spagetthi}),\ \texttt{available}(\texttt{spagetthi}, \texttt{s}(\texttt{0}))$

$:-\texttt{available}(\texttt{spagetthi}, \texttt{s}(\texttt{0}))$

$:-\texttt{day}(\texttt{s}(\texttt{0}))$

$:-\texttt{day}(\texttt{0})$

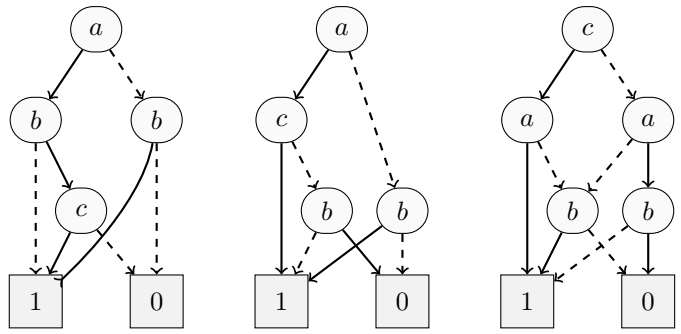
$:-$

It can be shown that SLD resolution is *sound* and *refutation-complete*. For definite programs, if SLD resolution finds a refutation of a query it is contained in the least Herbrand model. And if an atom is contained in the least Herbrand model, then there exists a refutation proof [Nilsson and Małuszyński, 1995]. SLD resolution traverses the so-called SLD tree by backtracking. Figure 2.1 shows the SLD tree for the query $\texttt{gets}(\texttt{john}, \texttt{X}, \texttt{0})$ in Example 2.1. There are two refutations for the query, indicated by ■ in the leaves, and each corresponds to a proof in the program.

2.2 Binary Decision Diagrams

A Binary Decision Diagram (BDD) [Bryant, 1986] is a data structure that represents Boolean functions. A BDD corresponds to a rooted directed acyclic graph. The internal nodes represent decisions and they correspond to the variables in the Boolean function. Each internal node has two outgoing edges, the "high edge" represents the variable being *true* and the "low edge" represents the variable being *false*. Each BDD has two terminal nodes. The 1-terminal indicates the Boolean function being *true* and the 0-terminal indicates it being *false*. Each path from the root node to the 1-terminal corresponds to a (partial) assignment of truth values to variables that results in the function being true. Figure 2.2(a) shows a BDD that represents the function $(a \wedge b \wedge c) \vee (a \wedge \neg b) \vee (\neg a \wedge b)$.

We assume the BDDs to be *ordered* and *reduced*. This means, that on all paths the decision nodes occur in the same order. Furthermore, irrelevant sub-BDDs are removed as explained below. Such a reduced ordered BDD is a *canonical* representation of a Boolean formula, which is a key property, for instance, in



(a) Using the variable order a, b, c

(b) Using the variable order a, c, b

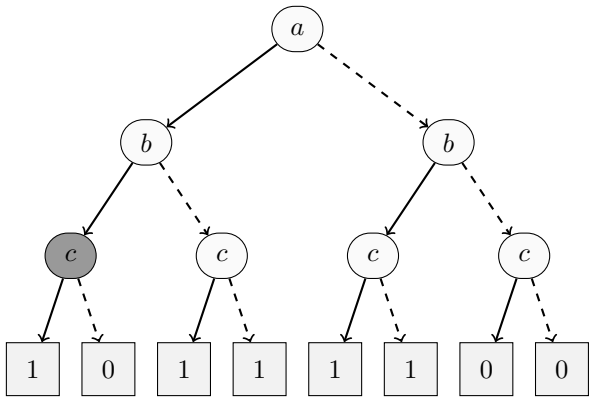
(c) Using the variable order c, a, b

Figure 2.2: Different binary decision diagrams representing the Boolean function $(a \wedge b \wedge c) \vee (a \wedge \neg b) \vee (\neg a \wedge b)$. The number of nodes in the BDD depends on the variable order and the represented function.

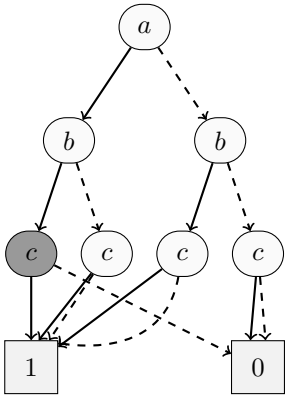
function verification. Please note, that depending on the variable order, one obtains different BDDs for a Boolean function (cf. Figure 2.2). The variable order also influences the size of the BDD, that is, the number of decision nodes. While the differences in size are small in this particular case (5 nodes compared to 4 nodes) it can be exponential in more complex formulae. Finding a good variable order is crucial for being able to construct the BDD and represent it in memory. While this is an NP-hard problem, there exist various heuristics that work well in practice. BDD packages like CUDD [Somenzi, 2009] have built-in reordering heuristics that are being applied while constructing the BDD. For some functions such as the multiplication of integers in binary notation, however, even the smallest BDD is exponentially large [Bryant, 1991]. One can construct a BDD based on a full binary decision tree as follows. After choosing an order of the variables in the Boolean formula one constructs a full binary decision tree, which has this order on each path and where the leaves are labelled with the function value given the corresponding variable assignment. After that one repeats the following two operations until no simplification step is possible:

- **Subgraph merging**: Find two isomorphic sub-BDDs G_1 and G_2. Delete G_2 and direct the ingoing edges of G_2 to the corresponding nodes in G_1.
- **Node removal:** Find a node v whose outgoing edges $low(v)$ and $high(v)$ point to the same node v'. Delete v and direct the ingoing edges of v to v'.

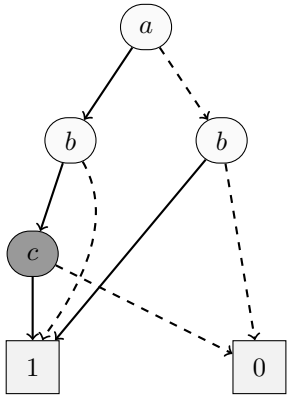
The resulting BDD solely depends on the function and the chosen variable order, while the order in which simplification steps are performed does not matter [Bryant, 1986]. Figure 2.3 shows the BDD construction for $(a \wedge b \wedge c) \vee (a \wedge \neg b) \vee (\neg a \wedge b)$.



(a) A full binary decision tree with the same variable order (a, b, c) on all paths.



(b) After merging all the 1-terminals into a single node and all 0-terminals into a single node.



(c) After deleting the redundant c nodes, that is, all but the dark gray shaded.

Figure 2.3: Constructing a BDD for the Boolean formula $(a \wedge b \wedge c) \vee (a \wedge \neg b) \vee (\neg a \wedge b)$. Solid lines represent the value of the node being true, dashed lines represent the value being false. The leaves indicate the formula's value given the variable assignment corresponding to the path from the root node. The single c node remaining in the final BDD has been shaded gray in every step for better visibility. Please note, the final BDD is isomorphic to the BDD shown in Figure 2.2(a).

2.3 Probabilistic Inference

A discrete probabilistic model defines a probability distribution $p(\cdot)$ over a set Ω of basic outcomes, that is, value assignments to the model's random variables. This distribution can then be used to evaluate a conditional probability distribution $p(q|e) = \frac{p(q \wedge e)}{p(e)}$, also called *target distribution*. Here, q is a query involving random variables, and e is the *evidence*, that is, a partial value assignment of the random variables. If e contains assignments to continuous variables, $p(e)$ is zero. Hence, evidence on continuous values has to be defined via a probability density function, also called a sensor model. Evaluating the target distribution is called *probabilistic inference*. In probabilistic logic programming, random variables often correspond to ground atoms such that $p(\cdot)$ defines a distribution over truth value assignments.

Probabilistic inference computes the probability of a logical query being true given truth value assignments for a number of such ground atoms. The probability $p(\cdot)$ of a query q is in the discrete case the sum over those outcomes $\omega \in \Omega$ where the query succeeds. In the continuous case, the sum is replaced by an integral and the distribution $p(\cdot)$ by a product of density function $\mathbf{F}(\cdot)$ That is,

$$p(q) = \sum_{\omega \in \Omega} p(\omega) \mathbb{1}_q(\omega) \ , \qquad \text{and} \qquad p(q) = \int_{\Omega} \cdots \int \mathbb{1}_q(\omega) \, d\mathbf{F}(\omega) \ , \tag{2.1}$$

where $\mathbb{1}_q(\omega) = 1$ if $\omega \models q$ and 0 otherwise. When describing the semantics of distributional programs in Chapter 5 we will use the notation $\int x \, dF(x)$ as unifying notation for both discrete and continuous distributions (cf. [Wasserman, 2003]).

The set of outcomes Ω is often very large or even infinite. Hence exact inference based on the summation in (2.1) quickly becomes infeasible and inference has to resort to approximation techniques based on *samples*, that is, randomly drawn outcomes $\omega \in \Omega$. Given a large set of such samples $\{s_1, \ldots, s_N\}$ drawn from $p(\cdot)$, the probability $p(q)$ can be estimated as the fraction of samples where q is true. If the task is not to compute $p(\cdot)$ but the conditional probability distribution $p(\cdot|e)$ given some *evidence* e, one can estimate the probability as

$$\hat{p}(q|e) := \frac{1}{N} \sum_{i=1}^{N} \mathbb{1}_q(s_i) \ .$$

Sampling from $p(\cdot|e)$ can be inefficient or infeasible in practice as the evidence needs to be taken into account. For instance, if one would use the definition of *conditional probability* to draw samples from $p(\cdot)$, all samples that are inconsistent with the evidence would not contribute to the estimate and would thus have to be *rejected*. More advanced sampling methods therefore often resort to a so-called *proposal distribution* that allows for easier sampling. The error introduced by

this simplification then needs to be accounted for when generating the estimate from the set of samples. An example for such a method is *importance sampling*, where each sample s_i has an associated weight w_i. Samples are drawn from an *importance distribution* $\pi(\cdot|e)$, and weights are defined as $w_i = \frac{p(s_i|e)}{\pi(s_i|e)}$. The true target distribution can then be estimated as

$$\hat{p}(q|e) = \frac{1}{W} \sum_{i=1}^{N} w_i \cdot \mathbb{1}_q(s_i) \ ,$$

where $W = \sum_i w_i$ is a normalization constant. The simplest instance of this algorithm is *rejection sampling* as already outlined above, where the samples are drawn from the prior distribution $p(\cdot)$ and weights are 1 for those samples consistent with the evidence, and 0 for the others. Especially for unlikely evidence rejection sampling suffers from a very high rejection rate. This means many samples that do not contribute to the final estimate are generated. This is known as the *rejection problem*. One way to address this is *likelihood weighted sampling*, which generates samples consistent with the evidence but drawn from a different distribution that is easier to evaluate. This requires corresponding modifications of the associated weights in order to produce correct estimates.

2.4 Distribution Semantics

Sato's distribution semantics [Sato, 1995] extends logic programming to the probabilistic setting by randomly choosing truth values of basic facts. The core of this language lies in splitting the logic program into a set F of *facts* and a set R of *rules*. Given a probability distribution P_F over the facts, the rules then allow one to extend P_F into a distribution over least Herbrand models of the logic program. Such a Herbrand model is called a *possible world*.

More precisely, it is assumed that $DB = F \cup R$ is ground and denumerable, and that no atom in F unifies with the head of a rule in R. Each truth value assignment to F gives rise to a unique least Herbrand model of DB. Thus, a probability distribution P_F over F can directly be extended into a distribution P_{DB} over these models. Furthermore, Sato shows that, given an enumeration $f_1, f_2, \ldots$ of facts in F, P_F can be constructed from a series of finite distributions $P_F^{(n)}(f_1 = x_1, \ldots, f_n = x_n)$ provided that the series fulfills the so-called *compatibility condition*, that is,

$$P_F^{(n)}(f_1 = x_1, \ldots, f_n = x_n) = \sum_{x_{n+1}} P_F^{(n+1)}(f_1 = x_1, \ldots, f_{n+1} = x_{n+1}) \ . \quad (2.2)$$

Chapter 3

ProbLog*

This chapter reviews ProbLog, the probabilistic logic programming language used in this thesis. We begin by outlining the basic concepts of the ProbLog language and its semantics in Section 3.1. We review the exact inference algorithm in Section 3.2 and discuss the mapping of annotated disjunctions to ProbLog in Section 3.3.

3.1 Syntax And Semantics

ProbLog is a simple probabilistic extension of the logic programming language Prolog (cf. [De Raedt et al., 2007; Kimmig, 2010]). One of its key properties is the separation of the model into a probabilistic part F and a logical part BK, which is akin to the distribution semantics (cf. Section 2.4). In this chapter we use a simplified version of the ALARM domain (adopted from [Russell and Norvig, 2003]) to explain the syntax and the basic concepts of the ProbLog language.

Example 3.1. *Consider the* ALARM-2 *program, which states that there is a burglary with probability* 0.1*, an earthquake with probability* 0.2 *and if either of them occurs the alarm will go off. If the alarm goes off, a person* `X` *will be notified*

*This chapter mainly reviews existing work on ProbLog (cf. De Raedt et al. [2007]; Kimmig [2010]). The results on mapping annotated disjunctions into ProbLog are joint work of the author with Luc De Raedt, Ingo Thon and Angelika Kimmig.

and will therefore call with the probability of `hears_alarm(X)`*, that is,* 0.8.

$$F = \{0.1{::}\ \texttt{burglary},\ 0.2{::}\ \texttt{earthquake},\ 0.8{::}\ \texttt{hears\_alarm(X)}\}$$

$$\begin{aligned} BK = \{&\texttt{person(mary)},\\ &\texttt{person(john)},\\ &\texttt{alarm} :- \texttt{burglary}\\ &\texttt{alarm} :- \texttt{earthquake}\\ &\texttt{calls(X)} :- \texttt{person(X)}, \texttt{alarm}, \texttt{hears\_alarm(X)}\} \end{aligned}$$

This example illustrates the separation between probabilistic and logical elements in a ProbLog program $T = F \cup BK$. The probabilistic facts

$$F = \{p_1 :: f_1, \cdots, p_n :: f_N\} \tag{3.1}$$

are a finite set of *probabilistic facts* and BK is a finite set of definite clauses called *background knowledge*. The probabilistic facts $p_n :: f_n$ in F are annotated with the probability p_n stating that $f_n\theta$ is true with probability p_n for all substitutions θ grounding f_n. We assume that probabilistic facts do not subsume one another, that is, the following case is not allowed:

$0.8 :: \texttt{hears\_alarm(X)}.$

$0.9 :: \texttt{hears\_alarm(bob)}.$

Furthermore, we assume that no head of a clause in BK can be unified with a probabilistic fact and, lastly, that the program T has the *finite support condition*. This condition holds if each atom in the Herbrand base of T has a finite SLD tree. Intuitively, this implies that the probability of an atom can only be influenced by finitely many ground probabilistic facts. The finite support condition is crucial for inference as it guarantees finiteness of the part of the ground program that is relevant for answering a query. Furthermore, it allows us to restrict the probability distributions and algorithms in this chapter to a finite set of *ground probabilistic facts*, which are assumed to be given in terms of a finite set of substitutions.

A ProbLog program T and a finite set of substitutions $\Theta = \{\theta_{nk}, \ldots \theta_{nK_n} \,|\, k = 1, \ldots K_n\}$ for the variables in the probabilistic facts $F = \{p_1 :: f_1, \cdots, p_N :: f_N\}$ define the maximal set of ground facts that can be generated from F

$$L^T := \{f_1\theta_{11},\ f_1\theta_{12},\ \ldots,\ f_N\theta_{NK_N}\}\ , \tag{3.2}$$

where K_n is the number of substitutions for fact f_n, Similarly, the maximal set of ground probabilistic facts is defined as

$$F^T := \{p_1 :: f_1\theta_{11},\ p_1 :: f_1\theta_{12},\ \ldots,\ p_N :: f_N\theta_{NK_N}\}\ . \tag{3.3}$$

In the ALARM-2 program, for example, these sets are

$$L^T = \{\texttt{burglary}, \texttt{earthquake},$$
$$\texttt{hears\_alarm}(\texttt{john}), \texttt{hears\_alarm}(\texttt{mary})\}$$
$$F^T = \{0.1 :: \texttt{burglary}, 0.2 :: \texttt{earthquake},$$
$$0.8 :: \texttt{hears\_alarm}(\texttt{john}), 0.8 :: \texttt{hears\_alarm}(\texttt{mary})\}$$

A ProbLog program defines a probability distribution over subsets of the ground facts $L \subseteq L^T$ as follows

$$P^T(L) := \prod_{f_n\theta_{nk} \in L} p_n \prod_{f_n\theta_{nk} \in L^T \setminus L} (1 - p_n) \ . \tag{3.4}$$

In terms of logic, L is a *complete interpretation* stating that all atoms contained in L are *true* and the rest is *false*. The interpretation $L = \{\texttt{burglary}\}$, for instance, expresses that a burglary and not an earthquake has occurred, while neither John nor Mary heard the alarm. The probability of this is $P^T(L) = 0.1 \times (1 - 0.2) \times (1 - 0.8) \times (1 - 0.8) = 0.0032$.

A *partial interpretation* I specifies for some but not necessarily for all atoms the truth value. We represent partial interpretations as $I = (I^+, I^-)$, where I^+ contains all *true* atoms and I^- all *false* atoms and $I^+ \cap I^- = \varnothing$. For instance, the partial interpretation

$$I^+ = \{\texttt{burglary}\}$$
$$I^- = \{\texttt{hears\_alarm}(\texttt{mary})\}$$

states that there was a burglary and Mary did not hear an alarm. However, it does not specify the truth value of `earthquake` and `hears_alarm(john)`. Hence when we calculate the probability of the partial interpretation $I = (I^+, I^-)$ of the probabilistic facts being sampled by a ProbLog program, we need to marginalize out all ground probabilistic facts $L^T \setminus (I^+ \cup I^-)$ whose truth value is unknown, which results in

$$P^T(I) = \prod_{f_n\theta_{nk} \in I^+} p_n \prod_{f_n\theta_{nk} \in I^-} (1 - p_n) \ . \tag{3.5}$$

ProbLog also defines a probability distribution P_w^T over possible worlds, that is Herbrand interpretations. Each subset of ground facts $L \subseteq L^T$ can be extended to a possible world by computing the least Herbrand model of L. This possible world is assigned the probability $P_w^T(L) = P^T(L)$.

The distribution P^T can be extended towards the *success probability* of a query q

$$P_s^T(q) := \sum_{\substack{L \subseteq L^T: \\ L \cup BK \models q}} P^T(L) \ . \tag{3.6}$$

Intuitively, the success probability is the probability that q is provable in a randomly sampled logic program. As there are exponentially many subprograms $L \subseteq L^T$, it is computationally infeasible to enumerate all of them. As shown by De Raedt et al. [2007], this problem can be solved by reducing it to that of computing the probability of the monotone DNF formula

$$P_s^T(q) = P\left(\bigvee_{(I^+,I^-) \in Expl^T(q)} \Big(\bigwedge_{b_i \in I^+} b_i \wedge \bigwedge_{b_i \in I^-} \neg b_i \Big) \right) \ , \tag{3.7}$$

where $Expl^T(q)$ denotes the set of explanations of the query q. This set contains all minimal partial interpretations $(I^+, I-) \subseteq L^T$ supporting q:

$$Expl^T(q) := \big\{ I \subseteq L^T \,\big|\, \big((I^+, I-) \models_{BK} q \,\big) \wedge \neg\big(\exists I' : I' \neq I \wedge I' \subseteq I' \wedge I' \models_{BK} q\big) \big\} \tag{3.8}$$

For instance, the set of explanations for `calls(john)` in the ALARM-2 program is

$$I_1^+ = \{\texttt{hears\_alarm(john)}, \texttt{burglary}\}$$

$$I_1^- = \varnothing$$

$$I_2^+ = \{\texttt{hears\_alarm(john)}, \texttt{earthquake}\}$$

$$I_2^- = \varnothing \ .$$

3.2 Inference

In the following we discuss the core elements of ProbLog's exact inference mechanism [Kimmig et al., 2011] and illustrate them using the query `calls(john)` in the ALARM-2 program. Given a query q the goal is to compute the success probability of the query in the program $T = F \cup BK$ (cf. Eq. 3.6). It is clear that computing this probability by enumerating all possible subprograms $L \subseteq L^T$ is infeasible since there are exponentially many. On the contrary, the number

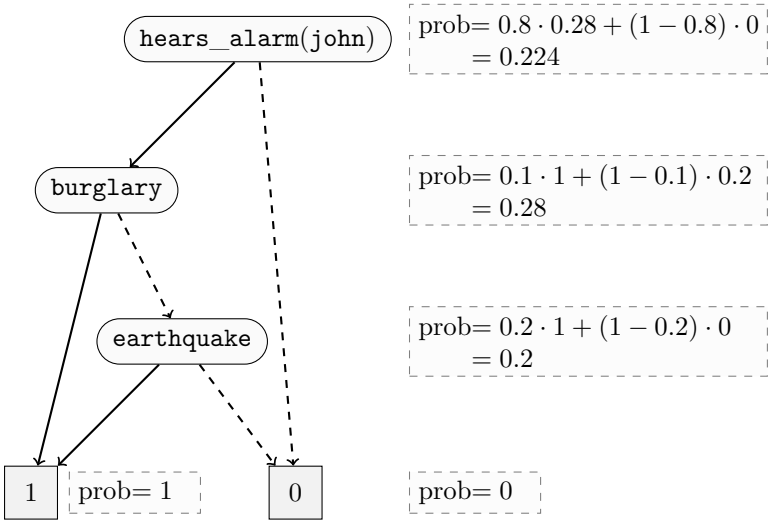


Figure 3.1: The BDD used by ProbLog to compute the probability of the query `calls(john)` in the ALARM-2 program. The annotations in the dashed boxes represent the intermediate values computed by Algorithm 2 when traversing the BDD. The probability of the query 0.224 is returned at the root node.

of proofs for a query is typically much smaller. For instance, the ALARM-2 program defines $2^4 = 16$ subprograms $L \subseteq L^T$, while there are only 2 proofs for the query `calls(john)`. Based on this observation the inference algorithm computes the success probability of the query based on the set of proofs instead. The main inference steps are: (1) finding all proofs for the query, (2) constructing a Boolean formula, (3) translating this into a BDD and (4) evaluating the BDD (cf. Figure 3.1).

As ProbLog is tightly integrated with the underlying Prolog system YAP [Santos Costa et al., 2011], it is possible to interact with the SLD resolution step (cf. Section 2.1.2) that searches for the proofs of a query. From an imperative programmer's point of view finding all ProbLog proofs corresponds to iterating over all possible proofs in BK and extracting the set of ground facts that is used by each proof as shown in Algorithm 1. The query `calls(john)` has two proofs (or explanations) in the ALARM-2 program using the set of probabilistic facts

$$e_1 = \{\texttt{hears\_alarm(john)}, \texttt{burglary}\}$$

$$e_2 = \{\texttt{hears\_alarm(john)}, \texttt{earthquake}\}$$

While each of these explanations can be assigned a probability, it is incorrect to sum them up in order to compute the success probability. The probability of e_1,

Algorithm 1 Exact Inference in ProbLog by collecting all proofs for a query, building a Boolean formula and evaluating the BDD corresponding to that formula.

```
1: function SuccessProb(query q, theory T)
2:     E ← ∅                                   ▷ Set of explanations for q
3:     initialize SLD resolution for q in BK ∪ L^T
4:     while SLD resolution finds another proof do
5:         e ←ProbabilisticFacts(proof)
6:         E ← E ∪ {e}
7:     end while
8:     BDD ←GenerateBDD(E)
9:     return BDDProb(root(BDD))
10: end function
```

for instance, is 0.8×0.1 and the probability of e_2 is 0.8×0.2 . The sum of these two is 0.24, while 0.224 is the success probability of `calls(john)`. The reason for this phenomenon is simple: e_1 and e_2 are not statistically independent. Hence one has to compute the probability of the joint event as

$$P(e_1 \vee e_2) = P(e_1) + P(e_2) - P(e_1 \wedge e_2) \ .$$

In other words, the explanations overlap and one has to *disjoin* them when computing the probability. Furthermore, if there are several explanations one has to consider all possible combinations of them. This so-called disjoint-sum of product (DSOP) problem is known to be $\#P$-complete [Valiant, 1979].

ProbLog inference solves this task by mapping it onto building a Binary Decision Diagram. To this end, ProbLog translates the set of explanations into a Boolean formula in disjunctive normal form, which is in turn represented compactly as a BDD. In our example, the formula

$$\big(\texttt{hears\_alarm(john)} \wedge \texttt{burglary}\big) \ \vee \ \big(\texttt{hears\_alarm(john)} \wedge \texttt{earthquake}\big)$$

is represented by the BDD shown in Figure 3.1. Lastly, the BDD is traversed by the function BDDProb shown in Algorithm 2. This step computes for each node the probability that the sub-BDD starting at that node is *true*. The probability of the root note 0.224 then corresponds to the success probability of the query. Algorithm 2 is linear in the number of BDD nodes when intermediate results are cached. For the ease of reading this was omitted in the algorithm's pseudocode.

The Shannon expansion [Shannon, 1948] of the Boolean formula, which is employed during the BDD construction, results in the high and low child of each node being disjoint. Namely, following the edge to the high child corresponds to assigning the value *true* to the variable, while following the edge to the low child corresponds to the value *false*. Therefore, the probabilities computed by Algorithm 2 on the two sub-BDD can be summed up as the corresponding events are statistically

Algorithm 2 Evaluating the BDD by traversing it top-down starting at the root. When caching intermediate results, the algorithm is linear in the size of the BDD.

```
1: function BDDProb(node n)
2:     if n is the 1-terminal of the BDD then return 1        ▷ Base Case
3:     if n is the 0-terminal of the BDD then return 0        ▷ Base Case
4:     let h and l be the high and low children of n          ▷ Inductive Case
5:     p_h ← BDDProb(h)
6:     p_l ← BDDProb(l)
7:     return p · p_h + (1 − p) · p_l
8: end function
```

independent. Constructing a BDD might still result in an exponential blow-up of the number of nodes. This requires one to employ so-called *reordering heuristics*, which try to find an order that minimizes the size of the BDD such that it becomes representable in memory.

Constructing the BDD is typically the limiting factor in large programs. In particular for the parameter estimation algorithms we develop in Part III of the thesis, where we need to repeatedly evaluate several queries, we hence need to be able to limit the size of the BDDs. Reducing the size of the BDD in turn allows us to apply parameter learning on large-scale real world datasets such as the *Biomine* graph [Sevon et al., 2006] ; cf. Section 6.5.1 for a description.

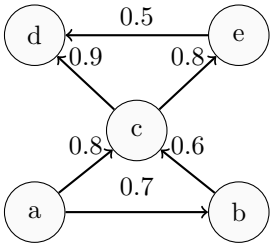
We therefore introduce the k-probability $P_k^T(q)$, which approximates the success probability by using the k best explanations, that is the k most likely ones, instead of all proofs when building the DNF formula used in Equation (3.7)

$$P_k^T(q) := P\left(\bigvee_{(I^+,I^-)\in Expl_k^T(q)} \left(\bigwedge_{b_i\in I^+} b_i \wedge \bigwedge_{b_i\in I^-} \neg b_i\right)\right) , \tag{3.9}$$

where $Expl_k^T(q)$ is limited to the k most likely explanations (cf. Eq. 3.8). There are two *special cases*: (1) if k is set to ∞, all proofs will be taken into account, which is similar to the success probability; (2) if k is set to 1 the most likely proof – which can be seen as the best explanation – is used.

Algorithm 1 can be used for computing the k-probability as well. Instead of collecting all proofs for the query, one has to modify step (1) of the algorithm towards returning the k best. This can be realized using a simple branch-and-bound approach (cf. also [Poole, 1993]).

We illustrate the k-probability using the probabilistic graph shown in Figure 3.2 and compute the k-probability for the query $\texttt{path}(\texttt{a},\texttt{d})$. 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. For $k = 1$, one obtains the



(a) Probabilistic graph

$$F = \{0.8{::}\ \texttt{edge(a,c)},\ 0.7{::}\ \texttt{edge(a,b)},$$
$$0.8{::}\ \texttt{edge(c,e)},\ 0.6{::}\ \texttt{edge(b,c)},$$
$$0.9{::}\ \texttt{edge(c,d)},\ 0.5{::}\ \texttt{edge(e,d)}\}$$
$$BK = \{\texttt{path(X,Y)} :\!-\ \texttt{edge(X,Y)}$$
$$\texttt{path(X,Y)} :\!-\ \texttt{edge(X,Z)},\ \texttt{path(Z,Y)}\}$$

(b) Encoding in ProbLog

Figure 3.2: This probabilistic graph represents unsure dependencies between nodes. Each edge is part of the graph with the probability attached to it. It can be modeled in ProbLog by the program shown on the right side.

probability of the most likely proof $P_1(path(a,d)) = 0.72$, the so-called *explanation probability*. For $k = 2$, overlap between the two best proofs has to be taken into account: the second proof only adds information if the first is disconnected. As they share the 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 for all $k \geq 4$ the probability $P_k(path(a,d)) = 0.83096$ for since there are only four proofs.

Some of the algorithms developed in this thesis are variants of the BDD traversal algorithm for computing the probability. In Chapter 4 we extend the computation on the BDD towards exact inference with continuous distributions. In Chapter 6 we compute the gradient of the success probability using the BDD representation, which is in turn used to optimize the parameters of the program with respect to a training set. LFI-ProbLog(cf. Chapter 7) introduces a variant of the BDD representation that allows for so-called *deterministic nodes*. The function Alpha (cf. Algorithm 14) used there is a variant of BDDProb that accounts for such type of node. Its pseudocode description (cf. Algorithm 14) shows how caching the intermediate results can be integrated.

Next to exact inference, ProbLog also provides sampling algorithms for approximating the success probability of a query. The algorithm of Kimmig et al. [2011] interleaves SLD resolution with sampling from the probabilistic facts, which corresponds to sampling proofs for a query. Another possibility is to sample at the level of the DNF representation [Shterionov et al., 2010]. ProbLog inference can also be combined with tabling [Mantadelis and Janssens, 2010]. This technique is used in logic programming to avoid revisiting similar parts of the SLD tree. Next to the performance improvement due to avoiding redundant proof steps, tabling can also be used to ensure *termination* of the proof procedure in cyclic programs.

3.3 Annotated Disjunctions

Annotated disjunctions (ADs) are a generalization of probabilistic facts with two elements: a condition in the form of a clause body and the exclusive choice between alternatives. They are defined as

$$p_1 :: h_1;\ \ldots\ ; p_N :: h_N :\!-\ b_1,\ \ldots\ , b_M$$

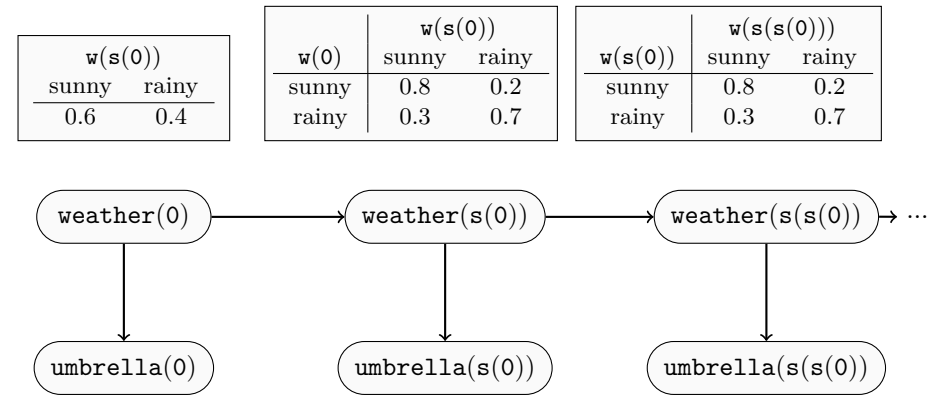
where $h_1, \ldots, h_N$ are atoms, the body $b_1,\ \ldots, b_M$ is a possibly empty conjunction of atoms and p_i are probabilities such that $\sum_{i=1}^{N} p_i \leq 1$. Furthermore, we assume the head to be range-restricted, that is, all variables in $h_1, \cdots, h_N$ do appear in the body as well. Lastly, we assume that no h_i can be unified with another head atom h_j, $i \neq j$.

An annotated disjunction states that if the body $b_1, \cdots, b_M$ is true then at most one of the h_i is true as well, where the choice is governed by the probabilities. The annotated disjunction `0.6 :: sunny(0); 0.4 :: rainy(0)`, for instance, states that the weather is either sunny or rainy with the corresponding probability. As for probabilistic facts, a non-ground AD denotes the set of all its groundings, and for each such grounding, choosing one of its head atoms to be true is seen as an independent random event. If the probabilities p_i in the head of the AD do not sum to 1, there is also a chance that nothing is chosen. The probability of this event is $1 - \sum_{i=1}^{N} p_i$. The following example illustrates how ADs can be used for representing Hidden Markov Models [Rabiner, 1989].

Example 3.2 (Hidden Markov Model)**.** *A Hidden Markov Model (HMM) represents the dependency of random variables over time. The HMM shown in Figure 3.3, for instance, models the weather depending on the weather the day before. The weather can be either rainy or sunny and a person will or will not take an umbrella when leaving the house in the morning. Using annotated disjunctions, this HMM can be written as follows:*

```
0.6 :: sunny(0); 0.4 :: rainy(0)                          // initial state

0.8 :: sunny(s(X)); 0.2 :: rainy(s(X)) :- sunny(X)        // state transition

0.3 :: sunny(s(X)); 0.7 :: rainy(s(X)) :- rainy(X)        // state transition

0.2 :: umbrella(X) :- sunny(X)                            // observation

0.9 :: umbrella(X) :- rainy(X)                            // observation
```

Annotated disjunctions can be translated into a set of equivalent probabilistic facts and background knowledge in ProbLog. However, probabilistic facts represent independent random variables, while two different atoms in the head of an AD



w(s(0))	
sunny	rainy
0.6	0.4

w(0)	w(s(0))	
	sunny	rainy
sunny	0.8	0.2
rainy	0.3	0.7

w(s(0))	w(s(s(0)))	
	sunny	rainy
sunny	0.8	0.2
rainy	0.3	0.7

w(0)	u(0)	
	t	f
sunny	0.2	0.8
rainy	0.9	0.1

w(s(0))	u(s(0))	
	t	f
sunny	0.2	0.8
rainy	0.9	0.1

w(s(s(0)))	u(s(s(0)))	
	t	f
sunny	0.2	0.8
rainy	0.9	0.1

Figure 3.3: A hidden Markov model; the rounded boxes represent the random variables, the edges represent the dependencies among them and the squared boxes contain the conditional probability table (CPT) for each node.

are dependent in the sense that at most one of them can be true in each possible world. Hence we need to modify the AD's body when translating it to ProbLog such that an AD

$$\mathtt{p_1} :: \mathtt{h_1};\ \ldots\ ; \mathtt{p_N} :: \mathtt{h_N} :\!\!-\ \mathtt{b_1},\ \ldots,\ \mathtt{b_M}$$

is translated into the set of probabilistic facts

$$F = \left\{ \tilde{p}_1 :: \mathtt{msw}(\mathtt{1}, \mathtt{V_1}, \ldots, \mathtt{V_K}),\ \ldots,\ \tilde{p}_N :: \mathtt{msw}(\mathtt{N}, \mathtt{V_1}, \ldots, \mathtt{V_K}) \right\}$$

where $\mathtt{V}_1, \ldots, \mathtt{V}_\mathtt{K}$ are *all* variables appearing in the AD. Furthermore, for each head atom $\mathtt{h}_\mathtt{i}$ one clause is added to the background knowledge BK as follows:

$$\mathtt{h}_1 \text{:-} \mathtt{b}_1, \cdots, \mathtt{b}_\mathtt{M}, \mathtt{msw}(1, \mathtt{V}_1, \ldots, \mathtt{V}_\mathtt{K}).$$

$$\mathtt{h}_2 \text{:-} \mathtt{b}_1, \cdots, \mathtt{b}_\mathtt{M}, \mathtt{msw}(2, \mathtt{V}_1, \ldots, \mathtt{V}_\mathtt{K}), \mathtt{not}(\mathtt{msw}(1, \mathtt{V}_1, \ldots, \mathtt{V}_\mathtt{K})).$$

$$\mathtt{h}_3 \text{:-} \mathtt{b}_1, \cdots, \mathtt{b}_\mathtt{M}, \mathtt{msw}(3, \mathtt{V}_1, \ldots, \mathtt{V}_\mathtt{K}), \mathtt{not}(\mathtt{msw}(2, \mathtt{V}_1, \ldots, \mathtt{V}_\mathtt{K})), \mathtt{not}(\mathtt{msw}(1, \mathtt{V}_1, \ldots, \mathtt{V}_\mathtt{K})).$$

$$\vdots$$

$$\mathtt{h}_\mathtt{N} \text{:-} \mathtt{b}_1, \cdots, \mathtt{b}_\mathtt{M}, \mathtt{msw}(\mathtt{i}, \mathtt{V}_1, \ldots, \mathtt{V}_\mathtt{K}), \mathtt{not}(\mathtt{msw}(\mathtt{i}-1, \mathtt{V}_1, \ldots, \mathtt{V}_\mathtt{K})), \vdots \mathtt{not}(\mathtt{msw}(1, \mathtt{V}_1, \ldots, \mathtt{V}_\mathtt{K})).$$

The probability $\tilde{p}_1$ is defined as p_1 and for $i > 1$ it is

$$\tilde{p}_i := \begin{cases} p_i \cdot \left(1 - \sum_{j=1}^{i-1} p_j\right)^{-1} & \text{if } p_i > 0 \\ 0 & \text{if } p_i = 0 \end{cases} . \tag{3.10}$$

Please note that the resulting probabilistic facts `msw` require an extra argument that identifies the AD, for instance, a unique number. We omitted this argument for the ease of reading. If the p_i sum to 1, it is possible to omit the last probabilistic fact $\mathtt{msw}(\mathtt{i}, \mathtt{V}_1, \ldots, \mathtt{V}_\mathtt{K})$ from the translation since its probability $\tilde{p}_n$ is 1. The correctness of this mapping is shown in Appendix A.

Although we assume definite programs in this thesis, the introduced mapping of annotated disjunctions to ProbLog uses negation, i.e., $\mathtt{not}(\mathtt{msw}(\mathtt{i}-1, \mathtt{V}_1, \ldots, \mathtt{V}_\mathtt{K}))$. Please note that this is a special kind of negation, provided by the ProbLog inference algorithm, that is restricted to ground probabilistic facts only. This type of negation can be represented in the distribution semantics by two distinct probabilistic facts $\mathtt{f}$ and $\mathtt{f}_\mathtt{not}$ that are governed by a joint distribution: $P(\mathtt{f} \wedge \mathtt{f}_\mathtt{not}) = P(\neg\mathtt{f} \wedge \neg\mathtt{f}_\mathtt{not}) = 0$, $P(\mathtt{f} \wedge \neg\mathtt{f}_\mathtt{not}) = p$ and $P(\neg\mathtt{f} \wedge \mathtt{f}_\mathtt{not}) = 1 - p$.

It is worth mentioning that one can recover the original probabilities from $\tilde{p}$ by setting $p_1 := \tilde{p}_1$ and iteratively applying the following transformation for $i = 2, 3, \ldots, N$

$$p_i := \tilde{p}_i \cdot \left(1 - \sum_{j=1}^{i-1} p_j\right) . \tag{3.11}$$

Hence Equation 3.10 and 3.11 define a bijection between p and $\tilde{p}$. This allows one to apply the parameter learning techniques we develop in Part III of this thesis on the ProbLog representation of the ADs such that the estimated parameters can be mapped back onto the corresponding ADs.

Example 3.3. *Using the above defined mapping of annotated disjunctions, the HMM defined in Example 3.2 is translated into a ProbLog theory as follows.*

$$F = \{0.6 :: \texttt{msw}(1,1),\ 0.8 :: \texttt{msw}(2,1,\texttt{X}),\ 0.3 :: \texttt{msw}(3,1,\texttt{X}),$$
$$0.2 :: \texttt{msw}(4,1,\texttt{X}),\ 0.9 :: \texttt{msw}(5,1,\texttt{X})\}$$

The first argument of each `msw` *atom identifies the corresponding annotated disjunction. This is necessary as there exist more than one AD.*

$$BK = \{\texttt{sunny}(0) :- \texttt{msw}(1,1),$$
$$\texttt{rainy}(0) :- \texttt{not}(\texttt{msw}(1,1)),$$
$$\texttt{sunny}(\texttt{s}(\texttt{X})) :- \texttt{sunny}(\texttt{X}),\ \texttt{msw}(2,1,\texttt{X}),$$
$$\texttt{rainy}(\texttt{s}(\texttt{X})) :- \texttt{sunny}(\texttt{X}),\ \texttt{not}(\texttt{msw}(2,1,\texttt{X})),$$
$$\texttt{sunny}(\texttt{s}(\texttt{X})) :- \texttt{rainy}(\texttt{X}),\ \texttt{msw}(3,1,\texttt{X}),$$
$$\texttt{rainy}(\texttt{s}(\texttt{X})) :- \texttt{rainy}(\texttt{X}),\ \texttt{not}(\texttt{msw}(3,1,\texttt{X})),$$
$$\texttt{umbrella}(\texttt{X}) :- \texttt{sunny}(\texttt{X}),\ \texttt{msw}(4,1,\texttt{X}),$$
$$\texttt{umbrella}(\texttt{X}) :- \texttt{rainy}(\texttt{X}),\ \texttt{msw}(5,1,\texttt{X})\}$$

Since no AD has a head with more than two elements in this example, the resulting translation is rather simple. Moreover, the probabilities in the ADs that govern the weather do sum up to one. Hence we can omit the last auxiliary fact in these cases, i.e., `msw(1,2)`*, since the attached probability is one. Please note, that the bodies of the resulting clauses are disjoint. For instance, in each possible world there is exactly one proof for each ground* `sunny/1`*,* `rainy/1` *and* `umbrella/1` *atom.*

Annotated disjunctions are the building blocks of other probabilistic programming languages closely related to ProbLog. CP-logic uses ADs to express causal dependencies between probabilistic events [Vennekens et al., 2009]. CPT-L is a variant of this formalism that uses ADs to represent sequential models over interpretations [Thon et al., 2008, 2011]. The multi-valued switches in PRISM [Sato, 1995] and the distributions in Poole's [2008] independent choice logic (ICL) enable one to define finite distributions over more than two elements. In Chapter 4 we introduce Hybrid ProbLog that allows for the use of continuous distributions in the form of so-called *continuous facts*. It is worth mentioning that the exact inference algorithm we develop for that language employs a mapping similar to that of ADs.

Part II

Continuous Distributions

Outline of Part II

In this part of the thesis we study how ProbLog can be extended with continuous distributions.

Chapter 4 extends ProbLog with *continuous facts* that contain a logical variable representing a continuous random variable. The resulting language, Hybrid ProbLog, is designed such that exact inference is tractable and can be performed by a dynamic query-dependent discretization approach.

Chapter 5 introduces a more expressive extension of ProbLog in the form of *distributional programs*. The building blocks of such a program are distributional clauses, which are a generalization of annotated disjunctions (cf. Section 3.3). It is possible to use finite, discrete as well as continuous distributions in the head of such clauses. We propose a sampling-based inference mechanism that combines elements of rejection sampling and likelihood weighted sampling. It uses lookahead to identify inconsistencies with the evidence early in the sample generation. Moreover, we adapt the well-known magic sets transformation towards distributional clauses in order to restrict the sampling to the relevant parts of the program.

Chapter 4

An Exact Inference Approach to Continuous Distributions*

Continuous distributions are essential for building a natural model in many applications. Probabilistic logic programming languages, such as ProbLog and CP-Logic [Vennekens et al., 2006], have, so far, largely focused on modeling discrete distributions and typically perform exact inference. The PRISM [Sato, 1995] system provides primitives for Gaussian distributions but requires the exclusive explanation property, which complicates modeling. Furthermore, many of the functional probabilistic programming languages, such as BLOG [Milch et al., 2005a] and Church [Goodman et al., 2008], are able to cope with continuous distributions but perform only approximate inference based on Markov Chain Monte Carlo sampling. Statistical relational learning systems such as Markov Logic Networks [Richardson and Domingos, 2006] and Bayesian Logic Programs [Kersting and De Raedt, 2007] have also been extended with continuous distributions. The key contribution of this chapter is a simple probabilistic extension of Prolog based on Sato's [1995] distribution semantics with both discrete and continuous distributions. This is realized by introducing a novel type of probabilistic fact where arguments of the fact can be distributed according to a continuous distribution. Queries can then be posed about the probability that the resulting arguments fall into specific intervals. We introduce the semantics of using continuous variables in combination with comparison operations and show how ProbLog's inference mechanism, based on Binary Decision Diagrams (BDDs) [Bryant, 1986], can be extended to cope with these distributions. The resulting language is called Hybrid ProbLog as it is essentially an extension of ProbLog.

*This chapter presents joint work with Manfred Jaeger and Luc De Raedt previously published in [Gutmann et al., 2010a].

Similarly to Hybrid ProbLog, Hybrid Markov Logic Networks (HMLNs) [Wang and Domingos, 2008] aim at integrating Boolean and numerical random variables in a probabilistic-logic modeling framework. While one can specify the distribution of continuous values *directly* in Hybrid ProbLog programs, one defines it *indirectly* in HMLNs by formulating equations that function as soft constraints for relationships among numerical and logical variables. For example, one could express that the temperature on day d is typically around 20° Celsius using the weighted equality w `temperature`$(d) = 20$, where larger weights w lead to a larger penalty for deviations of *temperature*(d) from 20. All weighted formulae containing *temperature*(d), together, implicitly define a probability distribution for the random variable *temperature*(d) due to HMLN semantics. However, one cannot directly specify this distribution to be Gaussian with, for example, mean 20 and standard deviation 5. No exact inference methods have yet been developed for HMLNs.

Hybrid ProbLog can be related to Hybrid Bayesian networks [Murphy, 1998] as they are both upgrading an existing framework with continuous elements. To begin with, Hybrid ProbLog uses logic as representation language, while Hybrid Bayesian networks are based on directed graphs with annotations on each node. Another major difference lies in the interaction between continuous and discrete random variables permitted in the models. In Hybrid Bayesian Networks, the distributions of continuous variables (usually Gaussian) are typically conditioned on discrete variables, though continuous variables cannot be parents of discrete ones. In Hybrid ProbLog this order is reversed: continuous variables are at the roots of the directed model, and the discrete (Boolean) variables are conditioned on the continuous ones. Thus, Hybrid ProbLog provides modeling capabilities and exact inference procedures that, for the propositional case, are in some sense complementary to Hybrid Bayesian networks.

This chapter has three core contributions: (1) an extension of ProbLog with continuous distributions, (2) a formal study of its semantics and (3) an efficient inference algorithm based on dynamic discretization.

The remainder of this chapter is organized as follows. Section 4.1 introduces the syntax and semantics of Hybrid ProbLog. Section 4.2 describes our exact inference algorithm. Before concluding, we evaluate the algorithm in Section 4.3.

4.1 Hybrid ProbLog

A Hybrid ProbLog theory $T = F \cup F^c \cup BK$ is described by a set of probabilistic facts F, a set of continuous facts F^c and a set of definite clauses BK. The continuous facts are of the form

$$F^c = \{(X_1, \phi_1) :: f_1^c, \cdots, (X_m, \phi_m) :: f_m^c\} \ , \tag{4.1}$$

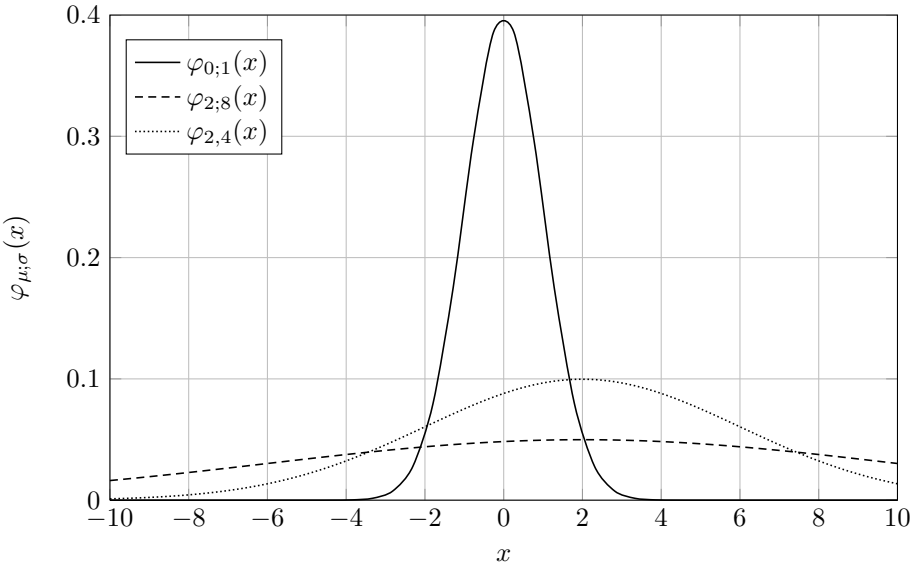


Figure 4.1: The probability density function of a Gaussian distribution for different mean μ and standard deviation σ.

where X_i is a Prolog variable, appearing in the atom f_i^c and ϕ_i is a density function. The continuous fact $(\texttt{X}, \texttt{gaussian}(2, 8)) :: \texttt{temp}(\texttt{D}, \texttt{X})$, for example, states that the temperature for day `D` is Gaussian-distributed with mean 2 and standard deviation 8. The probability density function for a Gaussian distribution with mean μ and standard deviation σ is defined as

$$\varphi_{\mu;\sigma}(x) := \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right) \quad . \tag{4.2}$$

The graph for $\varphi_{2;8}$ in Figure 4.1 shows the probability density function for the variable `X` in the continuous fact $(\texttt{X}, \texttt{gaussian}(2, 8)) :: \texttt{temp}(\texttt{D}, \texttt{X})$.

The probabilistic facts F and the background knowledge BK are akin to ProbLog (cf. Chapter 3), hence each ProbLog theory is a valid Hybrid ProbLog theory with $F^c = \varnothing$ but not vice versa. In this chapter, we denote facts, values, and substitutions related to the continuous part of T by the superscript ^{c}. We refer to variables that are unified with values stemming from continuous fact as *numerical variables*. Other variables will be called *logical variables*. In the atom $\texttt{temp}(\texttt{D}, \texttt{X})$, for instance, `D` is a numerical variable while `X` is logical.

From the user's perspective, continuous facts are queried like normal Prolog facts, and the value of the numerical variable is instantiated with a value drawn from

the underlying distribution. Hybrid ProbLog adds the following predicates to the background knowledge BK of the theory to process numerical variables:

- `below(X,`c`)` succeeds if `X` is a numerical variable, c is a number constant, and $X < c$,
- `above(X,`c`)` succeeds if `X` is a numerical variable, c is a number constant, and $\texttt{X} > c$,
- `ininterval(X,`c_1`,`c_2`)` succeeds if `X` is a numerical variable, c_1 and c_2 are number constants, and $\texttt{X} \in [c_1, c_2]$.

Unification with number constants is not supported for numerical variables, i.e., the call $\texttt{temp(d, 0)}$ fails. But one can express the same using

$$\texttt{temp(d, T)},\ \texttt{ininterval(T, 0, 0)}\ .$$

Similarly, standard Prolog comparison operators are not supported and one has to use the corresponding comparison predicate from the background knowledge:

$$\texttt{temp(d1, T)},\ \texttt{T} > 5$$

has to be written as

$$\texttt{temp(d1, T)},\ \texttt{above(T, 5)}\ .$$

Arithmetic expressions are not supported, i.e, the query

$$\texttt{temp(d1, T)},\ \texttt{Fahrenheit is } 9/5 * \texttt{X} + 32,\ \texttt{above(Fahrenheit, 41)}$$

is incorrect. In this particular example it is possible to rewrite the expression such that it does not require an arithmetic expression:

$$\texttt{temp(d1, T)},\ \texttt{above(T, }(41 - 32) * 5/9)\ .$$

Handling arithmetic expressions in Hybrid ProbLog requires the functions to be *invertible*. The expressions together with the comparison operators in BK define a system of inequality equations that has be solved. In the example above, the function is linear, which resulted in a straight-forward translation of the program since linear functions are *bijective*. In the general case, the translation can result in a more complex program the requires disjunctions. For instance

$$\texttt{temp(d1, T)},\ \texttt{T2 is T} * \texttt{T},\ \texttt{above(T2, 2)}$$

has to be translated into

$$\texttt{temp(d1, T)},\ (\texttt{below(T, } -\texttt{sqrt(2)};\ \texttt{above(T, sqrt(2))})\ ,$$

since the inequality $T^2 > 2$ is true for $T > \sqrt{2}$ and $T < -\sqrt{2}$. It is clear that solving a set of inequalities involving arbitrary functions quickly becomes intractable and one has to use *approximation algorithms* for finding solutions such as Newton's method. As this counters our goal of developing an *exact* inference algorithm we hence exclude arithmetic expressions.

Furthermore, if several numerical variables are used in one arithmetic expression they get "coupled". As a result there is a dependency that requires one to always consider the values of both variables simultaneously, which in turn complicates the inference algorithm. For the same reason, the comparison of two numerical variables is not supported, i.e., the query

$$\texttt{temp(d1,T1)},\ \texttt{temp(d2,T2)},\ \texttt{above(T1,T2)}$$

is incorrect. Despite these restriction, Hybrid ProbLog can handle non-trivial programs such as finite mixtures of Gaussian distributions.

Example 4.1 (Gaussian Mixture Model)**.** *The following theory encodes a Gaussian mixture model. The atom* `mix(X)` *can be used later on as if it were a simple continuous fact, which means the variable* `X` *can be processed using* `above/2`*,* `below/2` *and* `ininterval/3`*.*

```
0.6::heads.                    tails :- not(heads).

(X, gaussian(0,1))::g(X).      mix(X) :- heads, g(X).

(X, gaussian(5,2))::h(X).      mix(X) :- tails, h(X).
```

Please note, the clauses for `heads` and `tails` represent the annotated disjunction `0.5 :: heads; 0.5 :: tails` after translation to ProbLog (cf. Section 3.3).

The following theory shall be used as running example throughout the chapter to define the semantics of Hybrid ProbLog and to explain the inference algorithm.

Example 4.2 (Weather)**.** *This theory models weather during the winter time. The background knowledge states that a person catches a cold when the temperature is below 0° Celsius or when it is lower than 5° Celsius while it also rains.*

```
0.8::rain.          catchcold :- rain, temp(T), below(T,5).

(T, gaussian(2,8))::temp(T).    catchcold :- temp(T), below(T,0).
```

The semantics of Hybrid ProbLog theory $T = F \cup F^c \cup BK$ is given by probability distributions over subsets of the facts f_i (called *subprograms*), and over sample values for the numeric variables in the continuous facts f_i^c (called *continuous subprograms*). The subprograms $L \subseteq L_F$ are distributed as in ProbLog (cf. Equation (3.4)), and

the continuous subprograms are distributed as described in Section 4.1.1. Combining both gives one the success probability of queries in a Hybrid ProbLog theory as described in Section 4.1.2.

4.1.1 Distribution Over Continuous Subprograms

Let $\Theta^c = \{\theta^c_{j1}, \ldots \theta^c_{ji'_j} | j = 1, \ldots, m\}$ be a finite set of possible substitutions for the logical variables in the continuous facts $(X_j, \phi_j) :: f^c_j$ where i'_j is the number of substitutions for fact j. Each substitution instance $f^c_j\theta^c_{jk}$ is associated with a random variable X_{jk} with probability distribution ϕ_j. The X_{jk} are assumed to be independent. Let X denote the $|\Theta^c|$-dimensional vector of the random variables, and $f(X)$ their joint density function. A sample value x for X defines the continuous subprogram $L_x := \{f^c_j\theta^c_{jk}\{X_{jk} \leftarrow x_{jk}\} \mid j = 1, \ldots, m; k = 1, \ldots i'_j\}$ where $\{X_{jk} \leftarrow x_{jk}\}$ is the substitution of X_{jk} by x_{jk}.

Example 4.3 (Continuous Subprogram)**.** *Consider the following set of continuous facts where the second fact is* non-ground*. That is, one can obtain several ground instances where each instance has a value drawn independently from the same distribution.*

$$(\mathtt{X}, \mathtt{gaussian}(1,2)) :: \mathtt{h}(\mathtt{X}). \qquad (\mathtt{X}, \mathtt{gaussian}(4,3)) :: \mathtt{f}(\mathtt{X}, \mathtt{Y}).$$

When one applies the substitutions $\theta^c_{1,1} = \varnothing$, $\theta^c_{2,1} = \{\mathtt{Y} \leftarrow a\}$, $\theta^c_{2,2} = \{\mathtt{Y} \leftarrow b\}$ *together with the point* $x_{1,1} = 0.9$, $x_{2,1} = 2.3$, $x_{2,2} = 4.2$*, one obtains the* continuous subprogram $L_x = \{\mathtt{h}(0.9), \mathtt{f}(2.3, \mathtt{a}), \mathtt{f}(4.2, \mathtt{b})\}$.

The joint distribution of X thus defines a distribution over continuous subprograms. Let $X = (x_1, \ldots, X_M)$ be the value of the continuous facts ($M = |\Theta^c|$), let $I = [l_1, h_1] \times \ldots \times [l_M, h_M]$ be a $|\Theta^c|$-dimensional, and let f_i be the probability density function attached to the continuous fact identified by x_i. Then the probability of the set of continuous subprograms with values in I is defined as

$$P^T(X \in I) := \int_{l_1}^{h_1} \cdots \int_{l_M}^{h_M} f(x_1) \cdot f(x_2) \cdot \ldots \cdot f(x_M)\; dx_1\, dx_2 \ldots dx_M \;. \tag{4.3}$$

Example 4.4 (Joint Density)**.** *In Example 4.3, the joint density function is* $f(x) = f(x_{1,1}, x_{2,1}, x_{2,2}) = \varphi_{1;2}(x_{1,1}) \cdot \varphi_{4;3}(x_{1,2}) \cdot \varphi_{4;3}(x_{2,2})$ *where* $\varphi_{\mu;\sigma}$ *is the density of a Gaussian distribution (cf. Eq. 4.2).*

It is worth mentioning that the multi-dimensional interval I in (4.3) may be open or half-open in every dimension, that is, due to the integral the probability $P^T(X \in I)$ is independent of the openness of the intervals. One can exploit this property to

simplify the inference algorithm for computing the success probability, for instance, by ignoring endpoints and always generate right-open intervals. However, if one is not interested in the success probability of a query but in testing whether a particular query succeeds in the program, one has to mind the interval endpoints as the following program illustrates.

$(\mathtt{X}, \mathtt{gaussian}(1,2)) :: \mathtt{h}(\mathtt{X}).$

$\mathtt{a1} :- \mathtt{h}(\mathtt{X}),\ \mathtt{ininterval}(\mathtt{X},2,3).$

$\mathtt{a2} :- \mathtt{h}(\mathtt{X}),\ \mathtt{above}(\mathtt{X},2),\ \mathtt{below}(\mathtt{X},3).$

$\mathtt{b1} :- \mathtt{h}(\mathtt{X}),\ \mathtt{ininterval}(\mathtt{X},2,2).$

$\mathtt{b2} :- \mathtt{h}(\mathtt{X}),\ \mathtt{above}(\mathtt{X},2),\ \mathtt{below}(\mathtt{X},2).$

The success probability of `a1` is equal to the success probability of `a2` because $P^T(X \in [2,3]) = P^T(X \in (2,3)) = \int_2^3 \varphi_{1;2}(x)\,dx$. Similarly, the success probability of `b1` is equal to the one of `b2` because $P^T(X \in [2,2]) = P^T(X \in (2,2)) = 0$. However, while the query `b1` succeeds, there does not exist a proof for `b2`.

4.1.2 Success Probabilities of Queries

The success probability $P_s^T(q)$ of a query q is the probability that q is provable in $L \cup L_x \cup BK$, where L is distributed according to $P^T(L)$ (cf. (3.4)), and x according to $f(x)$ respectively. The key to computing success probabilities is the consideration of admissible intervals, as introduced in the following definition.

Definition 4.1. *An interval $I \subseteq \mathbb{R}^{|\Theta^c|}$ is called* admissible *for a query q and a theory $T = F \cup F^c \cup BK$ iff*

$$\forall x, y \in I, \forall L \subseteq L^T : \Big(L \cup L_x \cup BK\Big) \models q \Leftrightarrow \Big(L \cup L_y \cup BK\Big) \models q \tag{4.4}$$

If (4.4) holds, we can also write $L \cup L_I \cup BK \models q$.

A partition $\mathcal{A} = I_1, I_2, \ldots, I_k$ of $\mathbb{R}^{|\Theta^c|}$ is called admissible *for a query q and a theory T iff all I_i are admissible intervals for q and T.*

In other words, an admissible interval I is "small enough" such that the values of the numerical variables, as long as they are in I, do not influence the provability of q. Within an admissible interval, the query either always fails or always succeeds for any sampled subset $L \subseteq L^T$ of probabilistic facts.

Example 4.5 (Admissible Intervals)**.** *Consider the Hybrid ProbLog theory consisting out of a single continuous fact:*

$(\mathtt{X}, \mathtt{gaussian}(1,2)) :: \mathtt{h}(\mathtt{X})$

For the query $\texttt{h(X)}, \texttt{ininterval(X, 5, 10)}$*, the interval* $[0, 10]$ *is not admissible in this theory. The reason is that for* $x = 4 \in [0, 10]$ *the query fails but for* $x = 6 \in [0, 10]$ *it succeeds. The intervals* $[6, 9)$*,* $[5, 10]$*, or* $(-\infty, 5)$*, for example, are all admissible. Note, that the inference engine allows one to evaluate conjunctive queries and that the predicate* `ininterval/3` *is automatically added to the background knowledge.*

We now introduce the *discretized theory*. Instead of evaluating the distributions and instantiating the numerical variables with numbers, the discretized theory unifies such variables with the atom defined in the continuous fact. Hence instead of resulting in infinitely many proofs for each continuous fact, one for each possible instantiation of the numerical variable, the discretized theory contains only *one* proof. Please note that the discretized theory is constructed such that it simplifies the subsequent correctness proofs and the algorithms for exact inference.

Definition 4.2 (Discretized Theory)**.** *Let* $T = F \cup F^c \cup BK$ *be a Hybrid ProbLog theory, then the* discretized theory T_D *is defined as*

$$F \cup \{f^c\{X \leftarrow f^c\} \mid (X, \phi) :: f^c \in F^c\}$$
$$\cup\, BK$$
$$\cup\, \{\texttt{below(X, C)}, \texttt{above(X, C)}, \texttt{ininterval(X, C1, C2)}\}$$

where $f^c\{X \leftarrow f^c\}$ *is the atom resulting from substituting the variable* X *by the term* f^c*.*

The substitutions simplify the inference process. Whenever a numerical variable is used in a comparison predicate, the variable will be bound to the original continuous fact. Therefore, one can use a standard proof algorithm without keeping track of numerical variables. The discretized theory still contains probabilistic facts if F is not empty, thus it is a ProbLog theory. Each continuous fact results in an infinite number of proofs, one for each possible instantiation of the numerical variable. The discretized theory allows one to merge all these proofs into a single proof. This property is needed for computing admissible intervals efficiently and for defining the success probability.

Example 4.6 (Proofs in the discretized theory)**.** *The discretized theory* T_D *for Example 4.2 is*

```
0.8::rain.                catchcold :- rain, temp(T), below(T, 5).

temp(temp(T)).            catchcold :- temp(T), below(T, 0).

below(X, C).              above(X, C).        ininterval(X, C1, C2).
```

The discretized theory contains two proofs for the query `catchcold`*. For each proof, one can extract*

- f_i *the probabilistic facts used in the proof*
- c_i *the continuous facts used in the proof*
- d_i *the comparison operators used in the proof*

The proofs of `catchcold` *can be characterized by:*

$$f_1 = \{\texttt{rain}\} \qquad c_1 = \{\texttt{temp(temp(T))}\} \qquad d_1 = \{\texttt{below(temp(T),5)}\}$$

$$f_2 = \varnothing \qquad c_2 = \{\texttt{temp(temp(T))}\} \qquad d_2 = \{\texttt{below(temp(T),0)}\}$$

It is possible, though not in Example 4.6, that the same continuous fact is used by several comparison operators within one proof, i.e., $f_i = \{\texttt{below(f(X),10)}, \texttt{above(f(X),0)}\}$. In such cases, one has to build the intersection of all intervals to determine the interval, in which all comparison operators succeed, i.e., $f_i = \{\texttt{ininterval(f(X),0,10)}\}$. If that intersection is empty, the proof will fail in the original non-discretized theory. Building the intersections can also be interleaved with proving the goal.

The following theorem guarantees that an admissible partition exists for each query that has finitely many proofs in the discretized theory.

Theorem 4.1. *Let T be a Hybrid ProbLog theory; let q be a query in T that has only finitely many proofs in T_D and let Θ and Θ^c be finite sets of substitutions for the probabilistic and continuous facts respectively. Then there exists a finite partition of $\mathbb{R}^{|\Theta^c|}$ that is admissible for q in the theory obtained by grounding the probabilistic facts and continuous facts using Θ and Θ^c respectively.*

Proof. This follows from the fact that conditions defined by `below/2`, `above/2`, and `ininterval/3` are satisfied by intervals of sample values, and finite combinations of such conditions, which may appear in a proof, still define intervals. □

Algorithm 4 can be used to find *admissible partitions*. However, it has to be modified as described in Section 4.2 to respect the points at the interval boundaries. Given an admissible partition $\mathcal{A}$ one obtains the success probability of a query q as follows

$$P^T_{s,\mathcal{A}}(q) := \sum_{L \subseteq L^T} \sum_{\substack{I \in \mathcal{A}: \\ L \cup L_I \cup BK \models q}} P^T(L) \cdot P^T(X \in I) \ . \tag{4.5}$$

The following theorem shows that the values of P^T_s are independent of the partition $\mathcal{A}$ and therefore we can write $P^T_s(q)$ instead of $P^T_{s,\mathcal{A}}(q)$.

Theorem 4.2. *Let $\mathcal{A}$ and $\mathcal{B}$ be admissible partitions for the query q and the theory T then $P^T_{s,\mathcal{A}}(q) = P^T_{s,\mathcal{B}}(q)$.*

Proof. Proven directly, by showing that for two admissible partitions one can construct a third partition that returns the same success probability. Using the definition for the success probability (4.5) we get:

$$P^T_{s,\mathcal{A}}(q) := \sum_{L \subseteq L^T} \sum_{\substack{I \in \mathcal{A}: \\ L \cup L_I \cup BK \models q}} P^T(L) \cdot P^T(X \in I) \tag{4.6}$$

$$P^T_{s,\mathcal{B}}(q) := \sum_{L \subseteq L^T} \sum_{\substack{I \in \mathcal{B}: \\ L \cup L_I \cup BK \models q}} P^T(L) \cdot P^T(X \in I) \tag{4.7}$$

Since $\mathcal{A}$ and $\mathcal{B}$ are both finite, one can construct a partition $\mathcal{C}$ such that it subsumes both $\mathcal{A}$ and $\mathcal{B}$, that is

$$\forall I \in \mathcal{A} : \exists I_1, \ldots, I_n \in \mathcal{C} : I = I_1 \cup \ldots \cup I_n \text{ and}$$

$$\forall I \in \mathcal{B} : \exists I'_1, \ldots, I'_{n'} \in \mathcal{C} : I = I'_1 \cup \ldots \cup I'_{n'}$$

Because $\mathcal{A}$ is admissible and by construction of $\mathcal{C}$, we can represent any summand in (4.6) as a sum over intervals in $\mathcal{C}$. That is, for each $L \subseteq L^T$ and each $I \in \mathcal{A}$ there exist $I_1, \ldots, I_n \in \mathcal{C}$ such that

$$P^T(L) \cdot P^T(X \in I) = \sum_{i=1}^{n} P^T(L) \cdot P^T(X \in I_i) \ . \tag{4.8}$$

Because $\mathcal{A}$ is a partition and by construction of $\mathcal{C}$, the intervals needed to cover $I \in \mathcal{A}$ are disjoint from the intervals needed to cover $I' \in \mathcal{A}$ if $I \neq I'$. Therefore

$$\sum_{\substack{I \in \mathcal{A}: \\ L \cup L_I \cup BK \models q}} P^T(L) \cdot P^T(X \in I) = \sum_{\substack{I \in \mathcal{C}: \\ L \cup L_I \cup BK \models q}} P^T(L) \cdot P^T(X \in I) \tag{4.9}$$

for any subprogram $L \subseteq L^T$. From (4.9) and the definition of the *success probability* (4.5) follows

$$P^T_{s,\mathcal{A}}(q) = P^T_{s,\mathcal{C}}(q) \ .$$

Similarly, one can show that

$$P^T_{s,\mathcal{B}}(q) = P^T_{s,\mathcal{C}}(q) \ .$$

□

Theorem 4.1 and 4.2 guarantee that the semantics of Hybrid ProbLog, i.e., the fragment that restricts the usage of numerical variables, is well-defined. The imposed restrictions provide a good balance between expressivity and tractability. They allow one to discretize the space $\mathbb{R}^n$ of possible assignments to the continuous facts in multidimensional intervals such that the actual values within one interval do not matter. In turn, this makes efficient inference algorithms possible. Conversely, comparing two numerical variables against each other couples them. This requires a more complicated discretization of $\mathbb{R}^n$ in the form of polyhedra, which are harder to represent and to integrate over. Finally, allowing arbitrary functions to be applied on numerical variables eventually leads to a fragmentation of the space in arbitrary sets. This makes exact inference virtually intractable.

4.2 Exact Inference

In this section we present an exact inference algorithm for Hybrid ProbLog. Our approach generalizes De Raedt *et al.*'s BDD algorithm [De Raedt et al., 2007] and generates a BDD [Bryant, 1986] that is evaluated by a slight modification of the original algorithm. The pseudocode is shown in Algorithm 3, 4 and 5. In the remainder of this section, we explain the inference steps on Example 4.2 and calculate the success probability of the query `catchcold` using Algorithm 3.

1. All proofs for `catchcold` are collected by SLD resolution (Line 2 in Algorithm 3).

   $$f_1 = \{\texttt{rain}\} \qquad c_1 = \{\texttt{temp(T)}\} \qquad d_1 = \{\texttt{T} \in (-\infty, 5)\}$$
   $$f_2 = \varnothing \qquad c_2 = \{\texttt{temp(T)}\} \qquad d_2 = \{\texttt{T} \in (-\infty, 0)\}$$

   Each proof is described by a set of probabilistic facts f_i, a set of continuous facts c_i, and an interval for each numerical variable in c_i. When a continuous fact is used within a proof, it is added to c_i and the corresponding variable X is added to d_i with $X \in (-\infty, \infty)$.

   When later on `above(X,`c_1`)` is used in the same proof, the interval I stored in d_i is replaced by $I \cap (c_1, \infty)$, similarly for `below(X,`c_2`)` it is replaced by $I \cap (-\infty, c_2)$, and for `ininterval(X,`$\texttt{c}_1, \texttt{c}_2$`)` it is replaced by $I \cap [c_1, c_2]$,

2. The set of proofs we obtained uses one continuous fact ($|\Theta^c| = 1$), hence we partition $\mathbb{R}^1$. The loop in Line 3 of Algorithm 3 iterates over this set of continuous facts $(\cup_{1 \leq i \leq m} c_i) = \{\texttt{temp(T)}\}$. When calling the function CreatePartition($\{d_1, d_2\}$) (cf. Algorithm 4) we obtain the admissible partition $\{(-\infty, 0),\ [0, 5),\ [5, \infty)\}$ that is used to disjoin the proofs with

Algorithm 3 The inference algorithm collects all possible proofs and partitions the $\mathbb{R}^n$ space according to the constraints imposed by each proof. The intermediate variables f'_u and c'_u are superfluous and have been added to simplify the explanations in this section.

1: **function** SUCCESSPROB(query q, theory T)
2: $\{(f_i, c_i, d_i)\}_{1\leq i\leq m} \leftarrow$ FINDALLPROOFS(T, q) ▷ SLD resolution
3: **for** $c\theta \in \cup_{1\leq i\leq m} c_i$ **do** ▷ Iterate over used ground continuous facts
4: **let** $d_1^{c\theta}, \ldots, d_m^{c\theta}$ be the intervals attached to $c\theta$ in each proof
5: $\mathbf{A}_{c\theta} \leftarrow$CREATEPARTITION$(\{d_1^{c\theta}, \ldots, d_m^{c\theta}\})$
6: $\{b_{c\theta,I}\}_{I\in A_{c\theta}} \leftarrow$CREATEAUXBODIES$(\mathbf{A}_{c\theta})$
7: **end for**
8: $u \leftarrow 0$ ▷ # disjoint proofs
9: **for** $i = 1, 2, \ldots, m$ **do** ▷ Go over all proofs
10: **let** $(\hat{c}_1\hat{\theta}_1, \ldots, \hat{c}_t\hat{\theta}_t)$ be the continuous facts c_i used in proof i
11: **let** $(\hat{d}_1, \ldots, \hat{d}_t)$ be the domains d_i for all continuous facts used in proof i
12: **for** $(I_1, \ldots, I_t) \in \mathbf{A}_{\hat{c}_1\hat{\theta}_1} \times \cdots \times \mathbf{A}_{\hat{c}_t\hat{\theta}_t}$ **do** ▷ Go over all possible intervals
13: **if** $(d_1 \cap I_1 \neq \varnothing) \wedge \ \ldots \ \wedge (d_t \cap I_t \neq \varnothing)$ **then**
14: $u \leftarrow u + 1$ ▷ Increase counter for # disjoint proofs
15: $f'_u \leftarrow f_i$ ▷ Probabilistic facts stay the same
16: $c'_u \leftarrow c_i$ ▷ Continuous facts stay the same
17: $d'_u \leftarrow \{\mathrm{dom}(\hat{c}_1\hat{\theta}_1) = I_1, \ldots, \mathrm{dom}(\hat{c}_t\hat{\theta}_t) = I_t\}$ ▷ Adapt domains
18: $f''_u \leftarrow f_i \cup \{b_{c\theta,I} | c\theta \in c'_u, I \in d'_u\}$ ▷ Add aux. bodies to the facts
19: **end if**
20: **end for**
21: **end for**
22: $BDD \leftarrow$GENERATEBDD$(\bigvee_{1\leq i\leq u} \bigwedge_{f\in f''_i} f)$ ▷ cf. [Kimmig et al., 2008]
23: **return** BDDPROB(root(BDD)) ▷ cf. [De Raedt et al., 2007]
24: **end function**

respect to the continuous facts (Line 9-21 in Algorithm 3):

$$f'_1 = \{\texttt{rain}\} \qquad c'_1 = \{\texttt{temp(T)}\} \qquad d'_1 = \{\texttt{T} \in (-\infty, 0)\}$$

$$f'_2 = \{\texttt{rain}\} \qquad c'_2 = \{\texttt{temp(T)}\} \qquad d'_2 = \{\texttt{T} \in [0, 5)\}$$

$$f'_3 = \varnothing \qquad c'_3 = \{\texttt{temp(T)}\} \qquad d'_3 = \{\texttt{T} \in (-\infty, 0)\}$$

3. We create one auxiliary fact per continuous fact and interval. They are dependent, i.e., $\texttt{temp(T)}_{[-\infty,0)}$ and $\texttt{temp(T)}_{[0,5)}$ cannot be true at the same time. We have to make the dependencies explicit. Conceptually, this corresponds to adding an annotated disjunction to the program and replacing calls to continuous facts and background predicates by calls to the atom that corresponds to the interval defined by the background predicate. On our

example, the compiled AD (cf. Section 3.3) contributes the following clauses to BK:

$$\texttt{call\_temp(T)}_{(-\infty,0)} \texttt{ :- temp(T)}_{(-\infty,0)}$$

$$\texttt{call\_temp(T)}_{[0,5)} \texttt{ :- } \neg\texttt{temp(T)}_{(-\infty,0)}\texttt{, temp(T)}_{[0,5)}$$

$$\texttt{call\_temp(T)}_{[5,\infty)} \texttt{ :- } \neg\texttt{temp(T)}_{(-\infty,0)}\texttt{, } \neg\texttt{temp(T)}_{[0,5)}\texttt{, temp(T)}_{[5,\infty)}$$

The function CreateAUXBodies used in Line 6 of Algorithm 3 constructs the body these *auxiliary clauses*. The output is identical to the translation used for annotated disjunctions (cf. Section 3.3). Only one of the clauses can be true at the same time. The bodies encode a linear chain of decisions. The probability attached to an auxiliary fact $\texttt{temp(T)}_{[l,h)}$ is the conditional probability that the sampled value of T is in the interval $[l, h)$ given it is not in $(-\infty, l)$

$$P\left(\texttt{temp(T)}_{[l,h]}\right) := \left[\int_{l}^{h} \varphi_{2;8}(x)\,dx\right] \cdot \left[1 - \int_{-\infty}^{l} \varphi_{2;8}(x)\,dx\right]^{-1}, \quad (4.10)$$

where $\varphi_{2;8}(x)$ is the density function of the Gaussian attached to $\texttt{temp(T)}$ in the program. Akin to the encoding used for representing annotated disjunctions in ProbLog (cf. Section 3.3), this encodes that only one of the intervals can be *true* at a time and the success probability of $\texttt{call\_temp(T)}_{[l,h]}$ is exactly $\int_l^h \varphi_{2;8}(x)\,dx$. To evaluate the cumulative density function, we use the function Phi as described in [Marsaglia, 2004]. If we want to use any other distribution, we have to only replace the evaluation function of the density, as the rest of the algorithm does not depend on the particular distribution. Adding the bodies of the auxiliary clauses to f_i' yields the final set of proofs (Line 18 in Algorithm 3):

$$f_1'' = \{\texttt{rain}, \texttt{temp(T)}_{(-\infty,0)}\}$$

$$f_2'' = \{\texttt{rain}, \neg\texttt{temp(T)}_{(-\infty,0)}, \texttt{temp(T)}_{[0,5)}\}$$

$$f_3'' = \{\texttt{temp(T)}_{(-\infty,0)}\}$$

The proofs are now disjoint with respect to the continuous facts. That is, either the intervals for numerical variables are disjoint or identical. With respect to the probabilistic facts, they are not disjoint and summing up the probabilities of all proofs would yield an incorrect result. One would count the probability mass of the overlapping parts multiple times [De Raedt et al., 2007].

Algorithm 4 This function returns a partition of $\mathbb{R}$ by creating intervals touching the given intervals. Partitions of $\mathbb{R}^n$ can be obtained by building the cartesian product over the partitions for each dimension. This is feasible due to the restrictions imposed in Section 4.1. The resulting partition is not necessarily admissible as it ignores the interval endpoints, that is, intervals are right-open. However, for computing the success probability this is irrelevant (cf. Section 4.1.1).

function CREATEPARTITION(Set of intervals $D = \{d_1, \dots d_m\}$)
  $C \leftarrow \bigcup_{i=1}^{m} \{low_i, high_i\}$ ▷ low_i and $high_i$ are interval endpoints of d_i
  $C \leftarrow C \cup \{-\infty, \infty\}$ ▷ add upper and lower limit of $\mathbb{R}$
  $(c'_1, \dots, c'_k) \leftarrow$SORTANDIGNOREDUPLICATES($C$)
  Result $\leftarrow \{\ (-\infty, c'_2], (c'_{k-1}, \infty)\ \}$ ▷ $c'_1 = -\infty$ and $c'_k = \infty$
  for $i = 3, \dots, k-1$ **do**
    Result $\leftarrow$ Result $\cup \{\ [c'_{i-1}, c'_i)\ \}$
  end for
  return Result
end function

Algorithm 5 To evaluate the BDD we run a modified version of De Raedt *et al.*'s algorithm that takes the conditional probabilities for each continuous node into account. For Gaussian-distributed continuous facts we use the function PHI [Marsaglia, 2004] to evaluate $\int_{l_n}^{h_n} \varphi_{\mu_n;\sigma_n}(x)\,dx$ which performs a Taylor approximation of the cumulative density function (CDF). If the program requires distributions other than Gaussians, the user has to provide the corresponding code to compute the value the the CDF.

function BDDPROB(node n)
  if n is the 1-terminal **then return** 1
  if n is the 0-terminal **then return** 0
  let h and l be the high and low children of n
  $p_h \leftarrow$ BDDPROB(h)
  $p_l \leftarrow$ BDDPROB(l)
  if n is a continuous node with interval $[l_n, h_n]$ and density ϕ_n **then**
    $p \leftarrow \left[\int_{l_n}^{h_n} \phi_n(x)\,dx\right] \cdot \left[1 - \int_{-\infty}^{l_n} \phi_n(x)\,dx\right]^{-1}$
  else
    $p \leftarrow p_n$ ▷ the fact probability defined in the ProbLog program
  end if
  return $p \cdot p_h + (1-p) \cdot p_l$
end function

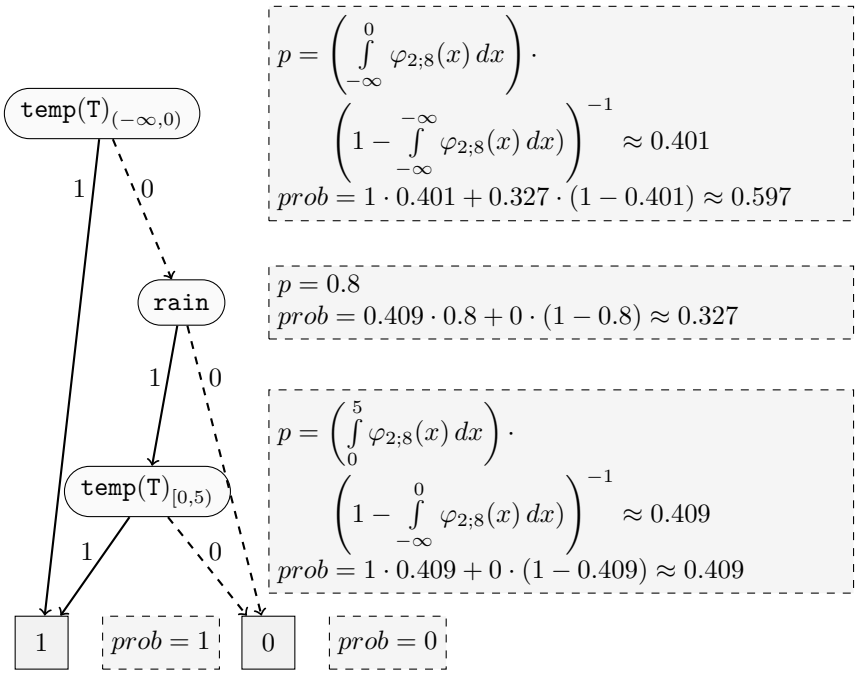


Figure 4.2: This BDD encodes all proofs of `catchcold` in the theory from Example 4.2. The dashed boxes show the intermediate results while traversing the BDD with Algorithm 5. The probabilities p correspond to the fact probability and are used to compute *prob*, the probability of the sub-BDD starting at the particular node. The success probability of the query is returned at the root and is 0.597.

4. To account for that, we translate the proofs into a Boolean expression in disjunctive normal form (cf. Line 22 in Algorithm 3) and represent it as BDD (cf. Figure 4.2). This step is similar to ProbLog's inference mechanism

$$\Big(\texttt{rain} \wedge \texttt{temp(T)}_{(-\infty,0)}\Big)$$
$$\vee\Big(\texttt{rain} \wedge \neg\texttt{temp(T)}_{(-\infty,0)} \wedge \texttt{temp(T)}_{[0,5)}\Big)$$
$$\vee\Big(\texttt{temp(T)}_{(-\infty,0)}\Big)$$

5. We evaluate the BDD with Algorithm 5 and get the success probability of `catchcold`. This algorithm is a modification of that of De Raedt et al. [2007] that takes into account the continuous nodes (cf. Alg. 2 in Section 3.2).

The function CreatePartition (cf. Algorithm 4) does not necessarily return an *admissible partition* as it ignores the interval endpoints by creating right-open intervals. For example, if one obtains two proofs that restrict a numerical variable to the interval $[1, 2]$ and $[2, 3)$ respectively, the minimal admissible partition is

$$\{(-\infty, 1), [1, 2), [2, 2], (2, 3), [3, \infty)\} \ .$$

The function CreatePartition, however, returns the inadmissible partition

$$\{(-\infty, 1), [1, 2), [2, 3), [3, \infty)\} \ .$$

With respect to the interval $[2, 3)$, the first proof succeeds for $x = 2$ but fails for any other value in $[2, 3)$, which is not allowed for admissible intervals (cf. Definition 4.1). When calculating the *success probability*, one can ignore the interval boundaries. Since the calculation involves integrals of the form $\int_{l_i}^{h_i} \phi(x)\, dx$, it is irrelevant whether intervals are open or closed (cf. Equation (4.10)). Also, an integral over a single point interval has the value 0.

However, when one wants to know whether there is a proof with specific values for some or all continuous facts, one has to be precise about the interval boundaries and use a modified algorithm. Admissibility can be ensured, for instance, by creating for each pair of constants the open interval (l_i, h_i) and the single point interval $[l_i, h_i]$. For the former example this is

$$\{(-\infty, 1), [1, 1], (1, 2), [2, 2], (2, 3), [3, 3], (3, \infty)\} \ .$$

4.3 Experiments

We set up experiments to answer the following question:

> How does the inference algorithm (cf. Algorithm 3) scale in the size of the partitions and in the number of ground continuous facts?

In domains where exact inference is feasible, the number of continuous facts and comparison operations is typically small compared to the rest of the theory. Our algorithm is an intermediate step between SLD resolution and BDD generation. Therefore, it is useful to know how much the disjoining operations cost compared

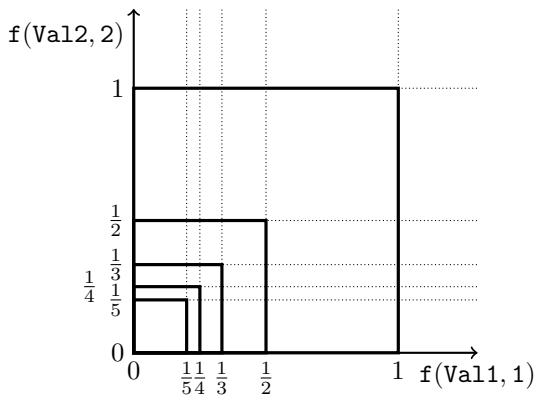


Figure 4.3: The query $\texttt{s}(5,2)$ succeeds if both values $\texttt{f}(\texttt{Val1},1)$ and $\texttt{f}(\texttt{Val2},2)$ lie in one of the intervals $[0,\frac{1}{1}]$, $[0,\frac{1}{2}]$, ..., $[0,\frac{1}{5}]$. These areas correspond to the thick-lined squares starting at $(0,0)$. Since they overlap one has to partition the space $\mathbb{R}^2$ in order to disjoin the proofs. Due to the restrictions of Hybrid ProbLog program, i.e., numerical variables cannot be compared against each other, one can obtain an admissible partition for each dimension independently. Algorithm 4 returns the partitions shown by the dotted lines. The horizontal lines partition the space of $\texttt{f}(\texttt{Val2},2)$ and the vertical the space of $\texttt{f}(\texttt{Val1},1)$.

to the other inference steps. We tested our algorithm on the following theory:

$$(\texttt{Val}, \texttt{gaussian}(0,1)) :: \texttt{f}(\texttt{Val}, \texttt{ID}).$$

$$\begin{aligned}\texttt{s}(\texttt{Consts},\texttt{Facts}) :- \ & \texttt{between}(1,\texttt{Facts},\texttt{ID}),\ \texttt{between}(1,\texttt{Consts},\texttt{Top}),\\ & \texttt{High is Top/Consts},\\ & \texttt{f}(\texttt{Val},\texttt{ID}),\\ & \texttt{ininterval}(\texttt{Val},0,\texttt{High}).\end{aligned}$$

The query $\texttt{s}(\texttt{Consts},\texttt{Facts})$ has $\texttt{Consts}\times\texttt{Facts}$ many proofs where both arguments have to be positive integers. The first argument determines the number of partitions needed to disjoin the proofs with respect to the continuous facts and the second argument determines how many continuous facts are used. The query $\texttt{s}(5,2)$, for instance, uses two continuous facts, $\texttt{f}(\texttt{Val1},1)$ and $\texttt{f}(\texttt{Val2},1)$, and compares them to the intervals $[0,1],[0,\frac{1}{2}],\cdots[0,\frac{1}{5}]$. Figure 4.3 shows the resulting partitioning when proving the query $\texttt{s}(5,2)$. In general, one obtains $(\texttt{Consts}+1)^{\texttt{Facts}}$ many partitions of the space $\mathbb{R}^{\texttt{Facts}}$.

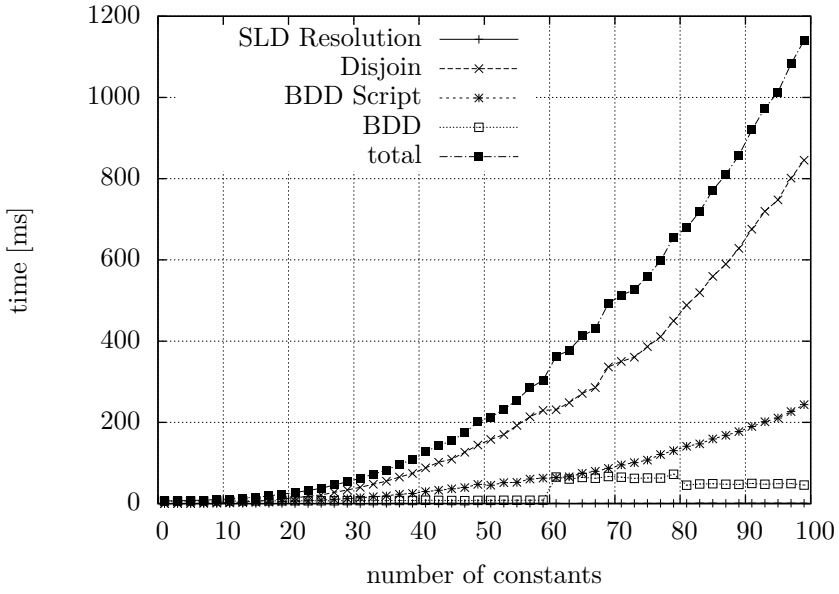


Figure 4.4: Runtimes for calculating the success probability of $\mathtt{s}(\mathtt{Consts},\mathtt{Facts})$ for varying the number of constants $s(1,1),\cdots s(100,1)$. As the graphs show, most of the time is spent on disjoining the proofs, that is partitioning the domains. The time spent on building an traversing the BDD stays more or less constant. This is due to the simplicity of the resulting Boolean expression, i.e., $\neg\mathtt{s}(\mathtt{Val},\mathtt{1})_{(-\infty,0)} \wedge \left(\mathtt{s}(\mathtt{Val},\mathtt{1})_{[0,\frac{1}{\mathtt{n}})} \vee \mathtt{s}(\mathtt{Val},\mathtt{1})_{[\frac{1}{\mathtt{n}},\frac{1}{\mathtt{n}-1})} \vee \ldots \vee \mathtt{s}(\mathtt{Val},\mathtt{1})_{[\frac{1}{2},1)}\right)$, which can be detected and exploited by the BDD package.

The success probability of $\mathtt{s}(\mathtt{Consts},\mathtt{Facts})$ is independent of $\mathtt{Consts}$. That is, for fixed $\mathtt{Facts}$ and any $c_1, c_2 \in \mathbb{N}$

$$P_s^T(\mathtt{s}(c_1,\mathtt{Facts})) = P_s^T(\mathtt{s}(c_2,\mathtt{Facts}))$$

We ran two series of queries[1]. In the first run we used one continuous fact and varied the number of constants from 1 to 100. As the graph in Figure 4.4 shows, the disjoin operation – which is finding all partitions, generating the auxiliary bodies and rewriting the proofs – runs in $O(\mathtt{Consts}^2)$ when everything else stays constant. In this case, due to compression and pruning operations during the BDD script generation [Kimmig et al., 2008] (the input for the BDD package),

[1]The experiments were performed on an Intel Core 2 Quad machine with 2.83GHz and 8GB of memory. We used the CUDD package for BDD operations and set the reordering heuristics to CUDD_REORDER_GROUP_SIFT. Each query has been evaluated 20 times and the runtimes have been averaged.

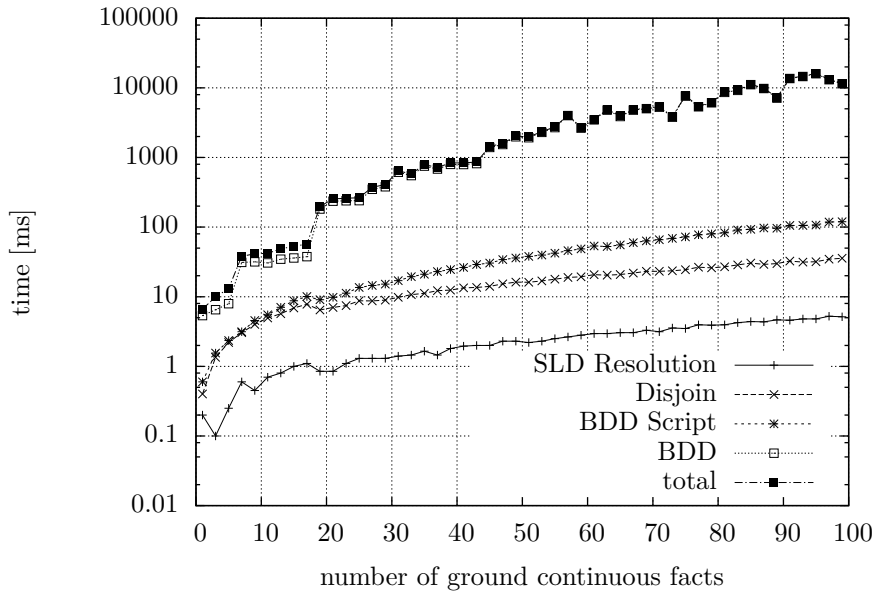


Figure 4.5: Runtimes for calculating the success probability of `s(Consts,Facts)` for varying the number of dimensions $s(5,1),\ldots,s(5,100)$. In this setting, most of the time is spent on BDD operations, that is, building the BDD based on the script and traversing it. The runtime for our disjoin operation grows only linearly. This is due to the fact that the partitions of $\mathbb{R}^n$ can be factorized into independent partitions for each dimension since comparison of two numerical variables is not allowed.

building and evaluating the BDD runs in linear time. In the second run we varied the number of continuous facts by evaluating the queries $\texttt{s}(5,1),\cdots,\texttt{s}(5,100)$. As Figure 4.5 shows, our algorithm (depicted by the Disjoin graph) runs in linear time. The runtime for the BDD operations grows exponentially due to the reordering heuristics used by the BDD package.

4.4 Conclusions And Future Work

We extended ProbLog with continuous distributions and introduced an exact inference algorithm. The resulting Hybrid ProbLog language imposes several restrictions that preclude some use cases and complicate the modeling process. The by far most severe restriction is the inability of comparing two random values against one another. In order to allow such comparisons, the disjoining algorithm

has to be extended towards n-dimensional polytopes as the resulting areas are not intervals but sets bounded by arbitrary planes. In turn, the computation of the cumulative density function will require some modification. Namely, it has to be extended towards computing the n-dimensional volume integral. Introducing linear constraints in the clauses, that is, expressions of the form

$$c_1 \cdot x_1 + \ldots + c_n \cdot x_n > c \ ,$$

increases the expressivity even further. Another useful extension are bivariate or multivariate distributions. Already now, the syntax allows for the representation of a bivariate Gaussian as

$$((\texttt{X1}, \texttt{X2}), \texttt{gaussian}([\texttt{m1}, \texttt{m2}], [[\texttt{c1}, \texttt{c2}], [\texttt{c3}, \texttt{c4}]]) :: \texttt{someatom}(\texttt{X1}, \texttt{X2})$$

based on the mean vector and the covariance matrix of the distribution. Multivariate distributions will require a joint treatment of all related variables. Allowing arbitrary functions to be applied on numerical variables is rather involved. The resulting areas of interest (akin to admissible intervals, cf. Definition 4.1) are not plain geometric objects, they can be arbitrary sets in $\mathbb{R}^n$. Consequently handling such sets and computing the volume integral, will, most likely, require sampling and other approximation techniques, which diverges from the original idea of using exact inference.

Chapter 5

An Approximate Inference Approach to Continuous Distributions*

Sampling-based inference approaches that can handle evidence have received little attention in logic programming based systems such as ProbLog and PRISM [Sato, 1995]. In this chapter, we investigate the integration of such approaches into probabilistic logic programming frameworks to broaden their applicability to real-world models. Particularly relevant in this regard are the ability of Church and BLOG to sample from continuous distributions and to answer conditional queries of the form $p(q|e)$, where e is the evidence. To accommodate (continuous and discrete) distributions, we introduce *distributional clauses*, which define random variables together with their associated distributions, conditional upon logical predicates. Random variables can be passed around in the logic program and the outcome of a random variable can be compared with other values by means of special built-ins. To formally establish the semantics of this new construct, we show that these random variables define a basic distribution over facts (using the comparison built-ins) as required in Sato's distribution semantics [Sato, 1995], and thus induces a distribution over least Herbrand models of the program. This contrasts with previous instances of the distribution semantics in that we no longer enumerate the probabilities of alternatives. Instead, we use arbitrary densities and distributions.

From a logic programming perspective, BLOG [Milch et al., 2005b] and related

*This chapter presents joint work with Ingo Thon, Angelika Kimmig, Maurice Bruynooghe, and Luc De Raedt previously published in [Gutmann et al., 2011b].

languages based on a functional paradigm perform *forward reasoning*. Hence the samples needed for probability estimation are generated starting from known facts and deriving additional facts, thus generating a *possible world*. PRISM and related languages follow the opposite approach of *backward reasoning*, where inference starts from a query and follows a chain of rules backwards to the basic facts, thus generating *proofs*. This difference is one of the reasons for using sampling in the first approach – exact forward inference would require all possible worlds to be generated, which is infeasible in most cases. Based on this observation, we contribute a new inference method for probabilistic logic programming that combines sampling-based inference techniques with forward reasoning. On the probabilistic side, the approach uses rejection sampling [Koller and Friedman, 2009], a well-known sampling technique that rejects samples that are inconsistent with the evidence. On the logic programming side, we adapt the *magic set* technique [Bancilhon et al., 1986] towards the probabilistic setting, thereby combining the advantages of both forward and backward reasoning. Furthermore, the inference algorithm is improved along the lines of the *SampleSearch* algorithm [Gogate and Dechter, 2011], which avoids choices leading to a sample that cannot be used in the probability estimation due to inconsistency with the evidence. We realize this by using a heuristic based on backward reasoning with limited proof length. This approach to inference creates a number of new possibilities for applications of probabilistic logic programming systems, including continuous distributions and Bayesian inference, which are essential in typical robotics tasks such as localization and people tracking [De Laet, 2010].

This chapter has three core contributions: (1) a probabilistic language that allows for finite, discrete and continuous distributions and that is expressive enough for real-world problems, (2) a formal study of its semantics, and (3) an efficient inference algorithm based on rejection sampling enhanced by a lookahead step.

The remainder of this chapter is organized as follows. Section 5.1 introduces the new language and its semantics, Section 5.2 presents a novel forward sampling algorithm for probabilistic logic programs. Before discussing the related work, we evaluate our approach in Section 5.3.

5.1 Distributional Programs

Sato's [1995] distribution semantics provides the basis for most probabilistic logic programming languages including PRISM [Sato and Kameya, 2001], ICL [Poole, 2008], CP-logic [Vennekens et al., 2009] and ProbLog [De Raedt et al., 2007]. The precise way of defining the basic distribution P_F differs among languages, though the theoretical foundations are essentially the same. The most basic instance of the distribution semantics, employed by ProbLog, uses so-called *probabilistic facts*. Each ground instance of a *probabilistic fact* directly corresponds to an

independent random variable, which takes the value of either "true" or "false". These probabilistic facts can also be seen as binary switches, cf. [Sato, 1995], which again can be extended to multi-ary switches or choices as used by PRISM and ICL. For switches, at most one of the probabilistic facts belonging to the switch is "true" according to the specified distribution. Finally, in CP-logic, such choices are used in the head of rules leading to the so-called *annotated disjunction* (cf. Section 3.3).

In Chapter 4 we introduced Hybrid ProbLog that also extends the distribution semantics with continuous distributions. To allow for exact inference, we had to impose severe restrictions on the distributions and their further use in the program. Two sampled values, for instance, cannot be compared against each other. Only comparisons that involve one sampled value and one number constant are allowed. Moreover, using sampled values in arithmetic expressions or as parameters for other distributions is not supported. Hence Hybrid ProbLog cannot be applied on typical robotics tasks such as localization and tracking, where Bayesian inference is essential [De Laet, 2010]. It is also not possible to reason over an unknown number of objects as BLOG [Milch et al., 2005a] does, though this is the case mainly for algorithmic reasons.

Here, we alleviate these restrictions by defining the basic distribution P_F over probabilistic facts based on both discrete and continuous random variables. We use a three-step approach to define this distribution. First, we introduce explicit random variables and corresponding distributions over their domains, both denoted by terms. Second, we use a mapping from these terms to the terms denoting (sampled) outcomes, which are then used to define the basic distribution P_F on the level of probabilistic facts. For instance, assume that an urn contains an unknown number of balls, where the number is drawn from a Poisson distribution and we say that this urn contains many balls if it contains at least 10 balls. We introduce a random variable `number` and define `many` :− `dist_gt`(≃(`number`), 9). Here, ≃(`number`) is the Herbrand term denoting the sampled value of `number`, and `dist_gt`(≃(`number`), 9) is a probabilistic fact whose probability of being true is the expectation that this value is actually greater than 9. This probability then carries over to the derived atom `many` as well. We will elaborate on the details in the following.

5.1.1 Syntax

In a logic program, following Sato, we distinguish between probabilistic facts that are used to define the basic distribution and rules that are used to derive additional atoms.[1] Probabilistic facts are not allowed to unify with any rule head. The distribution over facts is based on random variables whose distributions we define through so-called distributional clauses.

[1] A rule can have an empty body, in that case it represents a deterministic fact.

Definition 5.1 (Distributional clause)**.** *A* distributional clause *is a definite clause with an atom* $\mathtt{h} \sim \mathcal{D}$ *in the head, where* $\sim$ *is a binary predicate used in infix notation.*

For each ground instance $(\mathtt{h} \sim \mathcal{D} :\!-\ \mathtt{b}_1, \ldots, \mathtt{b}_n)\theta$, with θ being a substitution over the Herbrand universe of the logic program, the distributional clause defines a random variable $\mathtt{h}\theta$ and an associated distribution $\mathcal{D}\theta$. In fact, the distribution is only defined when $(\mathtt{b}_1, \ldots, \mathtt{b}_n)\theta$ is true in the semantics of the logic program. These random variables are terms of the Herbrand universe and can be used as any other term in the logic program. Furthermore, a term $\simeq(d)$ constructed from the reserved functor $\simeq/1$ represents the outcome of the random variable d. These functors can be used inside calls to special predicates in $dist\_rel = \{dist\_eq/2, dist\_lt/2, dist\_leq/2, dist\_gt/2, dist\_geq/2\}$. We assume that there is a fact for each of the ground instances of these predicate calls. These facts are the *probabilistic facts* of Sato's distribution semantics. Note that the set of probabilistic facts is enumerable since the Herbrand universe of the program is enumerable. A term $\simeq(d)$ links the random variable d with its outcome. The probabilistic facts compare the outcome of a random variable with a constant or the outcome of another random variable and succeed or fail according to the probability distribution(s) of the random variable(s).

Example 5.1 (Distributional clauses)**.**

$$\mathtt{nballs} \sim \mathtt{poisson}(6). \tag{5.1}$$

$$\mathtt{color}(\mathtt{B}) \sim\ [0.7 : \mathtt{b}, 0.3 : \mathtt{g}] :\!-\ \mathtt{between}(1, \simeq(\mathtt{nballs}), \mathtt{B}). \tag{5.2}$$

$$\mathtt{diameter}(\mathtt{B}, \mathtt{MD}) \sim \mathtt{gamma}(\mathtt{MD}/20, 20) :\!-\ \mathtt{between}(1, \simeq(\mathtt{nballs}), \mathtt{B}), \mathtt{mean\_diameter}(\simeq(\mathtt{color}(\mathtt{B})), \mathtt{MD}). \tag{5.3}$$

The defined distributions depend on the following logical clauses:

$$\mathtt{mean\_diameter}(\mathtt{C}, 5) :\!-\ \mathtt{dist\_eq}(\mathtt{C}, \mathtt{b}).$$

$$\mathtt{mean\_diameter}(\mathtt{C}, 10) :\!-\ \mathtt{dist\_eq}(\mathtt{C}, \mathtt{g}).$$

$$\mathtt{between}(\mathtt{I}, \mathtt{J}, \mathtt{I}) :\!-\ \mathtt{dist\_leq}(\mathtt{I}, \mathtt{J}).$$

$$\mathtt{between}(\mathtt{I}, \mathtt{J}, \mathtt{K}) :\!-\ \mathtt{dist\_lt}(\mathtt{I}, \mathtt{J}),\ \mathtt{I1\ is\ I} + 1, \mathtt{between}(\mathtt{I1}, \mathtt{J}, \mathtt{K}).$$

The distributional clause (5.1) *models the number of balls as a Poisson distribution with mean 6. The distributional clause* (5.2) *models a discrete distribution for the random variable color(B). With probability 0.7 the ball is blue and green*

otherwise. Note that the distribution is defined only for the values `B` *for which* `between(1,`$\simeq$`(nballs),B)` *succeeds. Execution of calls to the latter one gives rise to calls to probabilistic facts that are instances of* $dist\_leq(I, \simeq(nballs))$ *and* $dist\_lt(I, \simeq(nballs))$. *Similarly, the distributional clause* (5.3) *defines a gamma distribution that is also conditionally defined. Note that the conditions in the distribution depend on calls of the form* $mean\_diameter(\simeq(color(n)), MD)$ *with* n *a value returned by* `between/3`. *Execution of this call finally leads to calls* $dist\_eq(\simeq(color(n)), b)$ *and* $dist\_eq(\simeq(color(n)), g)$.

The syntax of distributional programs as introduced above is constructed to simplify the theoretical considerations and algorithms in this chapter. For an implementation of this language, particularly for convenience, it will be necessary to allow $\simeq(d)$ terms everywhere and having a simple program analysis insert the special predicates in the appropriate places by replacing $< /2$, $> /2$, $\leq /2$, $\geq /2$ predicates by $dist\_rel/2$ facts. Though extending unification is a bit more complex: as long as a $\simeq(h)$ term is unified with a free variable, standard unification can be performed; only when the other term is bound an extension is required. In this chapter, we assume that the special predicates `dist_eq/2`, `dist_lt/2`, `dist_leq/2`, `dist_gt/2`, and `dist_geq/2` are used whenever the outcome of a random variable needs to be compared with another value and that it is safe to use standard unification whenever a $\simeq$(`h`) term is used in another predicate.

For the basic distribution on facts to be well-defined, a program has to fulfill a set of validity criteria that have to be enforced by the programmer.

Definition 5.2 (Valid program)**.** *A program* P *is called* valid *if:*

(V1) *In the relation* `h` $\sim \mathcal{D}$ *that holds in the least fixpoint (w.r.t. the* $ST_P(S)$ *operator, cf. Def. 5.3) of a program, there is a functional dependency from* `h` *to* $\mathcal{D}$*, such that there exists a unique ground distribution* $\mathcal{D}$ *for each ground random variable* `h`.

(V2) *The program is* distribution-stratified*, that is, there exists a function* $rank(\cdot)$ *that maps ground atoms to* $\mathbb{N}$ *and that satisfies the following properties: (1) for each ground instance of a distribution clause* `h` $\sim \mathcal{D}$ `:-` $\mathtt{b_1}, \ldots \mathtt{b_n}$ *holds* $rank(\mathtt{h} \sim \mathcal{D}) > rank(\mathtt{b_i})$ *(for all* i*); (2) for each ground instance of another program clause:* `h :-` $\mathtt{b_1}, \ldots \mathtt{b_n}$ *holds* $rank(\mathtt{h}) \geq rank(\mathtt{b_i})$ *(for all* i*); (3) for each ground atom* `b` *that contains (the name of) a random variable* `h`*,* $rank(\mathtt{b}) \geq rank(\mathtt{h} \sim \mathcal{D})$ *(with* `h` $\sim \mathcal{D}$ *the head of the distribution clause defining* `h`*).*

(V3) *All ground probabilistic facts or, to be more precise, the corresponding indicator functions (cf.* (5.4)*) are* Lebesgue-measurable.

(V4) *Each atom in the least fixpoint can be derived from a finite number of probabilistic facts (*finite support condition *[Sato, 1995]).*

Together, (V1) and (V2) ensure that a single basic distribution P_F over the probabilistic facts can be obtained from the distributions of individual random variables defined in P. The third requirement (V3) is crucial. It ensures that the series of distributions $P_F^{(n)}$ needed to construct this basic distribution is well-defined. Particularly, it ensures that the probability of the indicator functions can be computed using the Lebesgue integral (cf, [Bartle, 1995]). Finally, the number of facts, over which the basic distribution is defined, needs to be countable. This is true, as we have a finite number of constants and functors: those appearing in the program.

The syntax of distributional programs deviates from that of Hybrid ProbLog introduced in Chapter 4. It is possible to translate a Hybrid ProbLog program into an equivalent distributional program. The continuous fact $(\texttt{X}, \texttt{gaussian}(2,8)) :: \texttt{temp}(\texttt{X})$, for instance, can be represented as the distributional fact $\texttt{temp} \sim \texttt{gaussian}(2,8)$. In turn, each appearance $\texttt{temp}(\texttt{X})$ and the subsequent places, where the variable X is used, have to be replaced by $\simeq \texttt{temp}$ and calls to dist_rel/2 respectively. While it is possible to translate distributional clauses into continuous facts and annotated disjunctions, it is impossible to translate the comparison operators dist_rel/2 in Hybrid ProbLog, as we excluded them to allow for exact inference.

5.1.2 Distribution Semantics

We now define the series of distributions $P_F^{(n)}$ where we fix an enumeration $f_1, f_2, \ldots$ of probabilistic facts such that $i < j \implies rank(f_i) \leq rank(f_j)$ where $rank(\cdot)$ is a *ranking function* showing that the program is distribution-stratified. For each predicate $\texttt{rel}/2 \in dist\_rel$, we define an *indicator function* as follows:

$$I^1_{\texttt{rel}}(X_1, X_2) := \begin{cases} 1 & \text{if } \texttt{rel}(X_1, X_2) \text{ is true} \\ 0 & \text{if } \texttt{rel}(X_1, X_2) \text{ is false} \end{cases} . \tag{5.4}$$

Furthermore, we define $I^0_{\texttt{rel}}(X_1, X_2) := 1.0 - I^1_{\texttt{rel}}(X_1, X_2)$ such that negation of comparison operators, i.e., $\texttt{not}(\texttt{dist\_rel}(\texttt{X}_1, \texttt{X}_2))$, can be mapped on it. We then use the expected value of the indicator function to define probability distributions $P_F^{(n)}$ over finite sets of ground facts $f_1, \ldots, f_n$. If $\{rv_1, \ldots rv_m\}$ is the set of random variables these n facts depend on, ordered such that if $rank(rv_i) < rank(rv_j)$, then $i < j$ (cf. (V2) in Definition 5.2). Furthermore, let $f_i = rel_i(t_{i1}, t_{i2})$, $x_j \in \{1, 0\}$, and $\theta^{-1} = \{\simeq(rv_1)/V_1, \ldots, \simeq(rv_m)/V_m\}$. The latter one replaces all evaluations of

random variables, on which the f_i depend, by variables for integration.

$$
\begin{aligned}
&P_F^{(n)}(f_1 = x_1, \ldots, f_n = x_n) \\
&:= \mathbb{E}[I^{x_1}_{rel_1}(t_{11}, t_{12}) \times \ldots \times I^{x_n}_{rel_n}(t_{n1}, t_{n2})] \\
&= \int \cdots \int \left(I^{x_1}_{rel_1}(t_{11}\theta^{-1}, t_{12}\theta^{-1}) \times \ldots \times I^{x_n}_{rel_n}(t_{n1}\theta^{-1}, t_{n2})\theta^{-1} \right) \\
&\qquad\qquad \times \, d\mathcal{D}_{rv_1}(V_1) \cdots d\mathcal{D}_{rv_m}(V_m) \quad (5.5)
\end{aligned}
$$

Example 5.2 (Basic Distribution)**.** *Let* $f_1, f_2, \ldots = dist\_lt(\simeq(b1), 3), dist\_lt(\simeq(b2), \simeq(b1)), \cdots$*. Then the second distribution in the series is*

$$
\begin{aligned}
&P_F^{(2)}(dist\_lt(\simeq(b1), 3) = x_1, dist\_lt(\simeq(b2), \simeq(b1)) = x_2) \\
&= \mathbb{E}[I^{x_1}_{\texttt{dist\_lt}}(\simeq(b1), 3), I^{x_2}_{\texttt{dist\_lt}}(\simeq(b2), \simeq(b1))] \\
&\qquad = \iint \left(I^{x_1}_{\texttt{dist\_lt}}(V1, 3) \times I^{x_2}_{\texttt{dist\_lt}}(V2, V1) \right) d\mathcal{D}_{b1}(V1)\, d\mathcal{D}_{b2}(V2)
\end{aligned}
$$

Proposition 5.1. *Let P be a valid program. P defines a probability measure P_P over the set of fixpoints of the T_P operator. Hence P also defines for an arbitrary formula* `q` *over atoms in its Herbrand base the probability that* `q` *is true.*

Proof sketch. It suffices to show that the series of distributions $P_F^{(n)}$ over facts (cf. (5.5)) is of the form that is required in the distribution semantics, that is, these are well-defined probability distributions that satisfy the compatibility condition (cf. (2.2)). This is a direct consequence of the definition in terms of indicator functions and the measurability of the underlying facts required for valid programs. □

5.1.3 T_P Semantics

In the following, we give a procedural view onto the semantics by extending the T_P operator (cf. Definition 2.3) to handle probabilistic facts `dist_rel(t1,t2)`. To do so, we introduce a function ReadTable(·) that keeps track of the sampled values of random variables to evaluate probabilistic facts. This is required because interpretations of a program only contain probabilistic facts, but not the values of the associated random variables. Given a probabilistic fact `dist_rel(t1,t2)`, ReadTable returns the truth value of the fact based on the values of the random variables h in the arguments, which are either retrieved from the table or sampled

according to their definition $\mathtt{h} \sim \mathcal{D}$, as included in the interpretation and stored in case they are not available yet.

Definition 5.3 (Stochastic T_P operator)**.** *Let P be a distribution-stratified program and ground(P) the set of all ground instances of clauses in P. Starting from a set of ground facts S the ST_P operator returns*

$$:= \Big\{ \mathtt{h} \;\Big|\; \mathtt{h} :\!- \mathtt{b_1}, \ldots, \mathtt{b_n} \in \mathit{ground}(P) \text{ and } \forall\ \mathtt{b_i} : \text{ either } \mathtt{b_i} \in S \text{ or } \big(\mathtt{b_i} = \mathit{dist\_rel}(t1, t2) \wedge (t_j = \simeq\!(h) \rightarrow (\mathtt{h} \sim \mathcal{D}) \in S) \wedge \textsc{ReadTable}(\mathtt{b_i}) = \mathit{true}\big) \Big\}$$

ReadTable ensures that the basic facts are sampled from their joint distribution as defined in Sec. 5.1.2 during the construction of the standard fixpoint of the logic program. Thus, each fixpoint of the ST_P operator corresponds to a possible world whose probability is given by the distribution semantics.

5.2 Forward Sampling Using Magic Sets And Backward Reasoning

In this section we introduce our new method for probabilistic forward inference. Towards this aim, we first extend the magic set transformation to distributional clauses. We then develop a rejection sampling scheme using this transformation. This scheme also incorporates backward reasoning to check for consistency with the evidence during sampling and thus reduces the rejection rate.

5.2.1 Probabilistic Magic Set Transformation

The disadvantage of forward reasoning in logic programming is that the search is not goal-driven, which might generate irrelevant atoms. The *magic set* transformation [Bancilhon et al., 1986; Nilsson and Małuszyński, 1995] focuses forward reasoning in logic programs towards a goal, to avoid the generation of uninteresting facts. It thus combines the advantages of both reasoning directions.

Definition 5.4 (Magic Set Transformation)**.** *If P is a logic program, then we use* Magic(P) *to denote the smallest program such that if $\mathtt{A_0} :\!- \mathtt{A_1}, \ldots, \mathtt{A_n} \in P$ then*

- $\mathtt{A_0} :\!- \mathtt{c}(\mathtt{A_0}), \mathtt{A_1}, \ldots, \mathtt{A_n} \in$ Magic(P) *and*
- *for each $1 \leq i \leq \mathtt{n}$:* $\mathtt{c}(\mathtt{A_i}) :\!- \mathtt{c}(\mathtt{A_0}), \mathtt{A_1}, \ldots, \mathtt{A_{i-1}} \in$ Magic(P)

The meaning of the additional c/1 atoms (c=call) is that they "switch on" clauses when they are needed to prove a particular goal. If the corresponding switch for the head atom is not true, the body is not true and thus cannot be proven. The magic transformation is both sound and complete [Kemp et al., 1995] for stratified programs. Furthermore, if the SLD tree of a goal is finite, forward reasoning in the transformed program terminates. The same holds if forward reasoning on the original program terminates.

We now extend this transformation to distributional clauses. The idea is that the distributional clause for a random variable h is activated when there is a call to a probabilistic fact $dist\_rel(t_1, t_2)$ depending on h.

Definition 5.5 (Probabilistic Magic Set Transformation)**.** *For program P, let P_L be P without distributional clauses.* $\mathrm{M}(P)$ *is the smallest program s.t.* $\textsc{Magic}(P_L) \subseteq \mathrm{M}(P)$ *and for each* $\mathtt{h} \sim \mathcal{D} :- \mathtt{b_1}, \ldots, \mathtt{b_n} \in P$ *and* $\mathtt{rel} \in \{\mathtt{eq}, \mathtt{lt}, \mathtt{leq}, \mathtt{gt}, \mathtt{geq}\}$*:*

- $\mathtt{h} \sim \mathcal{D} :- (\mathtt{c}(\mathtt{dist\_rel}(\simeq(\mathtt{h}), \mathtt{X})); \mathtt{c}(\mathtt{dist\_rel}(\mathtt{X}, \simeq(\mathtt{h}))), \mathtt{b_1}, \ldots, \mathtt{b_n} \in \mathrm{M}(P)$
- $\mathtt{c}(\mathtt{b_i}) :- (\mathtt{c}(\mathtt{dist\_rel}(\simeq(\mathtt{h}), \mathtt{X})); \mathtt{c}(\mathtt{dist\_rel}(\mathtt{X}, \simeq(\mathtt{h}))), \mathtt{b_1}, \ldots, \mathtt{b_{i-1}} \in \mathrm{M}(P)$.

Then $\textsc{PMagic}(P)$ *consists of:*

- *a clause* $\mathtt{a\_p}(\mathtt{t_1}, \ldots, \mathtt{t_n}) :- \mathtt{c}(\mathtt{p}(\mathtt{t_1}, \ldots, \mathtt{t_n})), \mathtt{p}(\mathtt{t_1}, \ldots, \mathtt{t_n})$ *for each built-in predicate (including* dist_rel/2 *for* $\mathtt{rel} \in \{\mathtt{eq}, \mathtt{lt}, \mathtt{leq}, \mathtt{gt}, \mathtt{geq}\}$*) used in* $\mathrm{M}(P)$.
- *a clause* $\mathtt{h} :- \mathtt{b'_1}, \ldots, \mathtt{b'_n}$ *for each clause* $\mathtt{h} :- \mathtt{b_1}, \ldots, \mathtt{b_n} \in \mathrm{M}(P)$ *where* $\mathtt{b'_i} = \mathtt{a\_b_i}$ *if* $\mathtt{b_i}$ *uses a built-in predicate and else* $\mathtt{b'_i} = \mathtt{b_i}$.

Note that every call to a built-in b is replaced by a call to a_b; the latter predicate is defined by a clause that is activated when there is a call to the built-in (c(b)), which effectively calls the built-in. The transformed program computes the distributions only for random variables whose value is relevant to the query. These distributions are the same as those obtained in a forward computation of the original program. Hence we can show:

Lemma 5.1. *Let P be a program and* $\textsc{PMagic}(P)$ *its probabilistic magic set transformation extended with a seed $c(q)$. The distribution over q defined by P and by* $\textsc{PMagic}(P)$ *is the same.*

Proof sketch. In both programs, the distribution over the query q is determined by the distributions of the atoms $dist\_eq(t_1, t_2)$, $dist\_leq(t_1, t_2)$, $dist\_lt(t_1, t_2)$, $dist\_geq(t_1, t_2)$ and $dist\_gt(t_1, t_2)$, on which q depends in a forward computation of the program P. The magic set transformation ensures that these atoms are called in the forward execution of $\textsc{PMagic}(P)$. In $\textsc{PMagic}(P)$, a call to such an

Algorithm 6 Main loop for sampling-based inference to calculate the conditional probability $p(q|e)$ for query q, evidence e and program L.

```
1: function EVALUATE(L, q, e, Depth)
2:     L* ←PMAGIC(L) ∪ {c(a)|a ∈ e ∪ q}
3:     n+ ← 0
4:     n− ← 0
5:     while Not converged do
6:         (I, w) ←STPMAGIC(L*, L, e, Depth)
7:         if q ∈ I then                          ▷ Query q is true in sample I
8:             n+ ← n+ + w                        ▷ Add sample weight to positive count
9:         else
10:            n− ← n− + w                        ▷ Add sample weight to negative count
11:        end if
12:    end while
13:    return n+/(n+ + n−)
14: end function
```

atom activates the distributional clause for the involved random variable. As this distributional clause is a logic program clause, soundness and completeness of the magic set transformation ensures that the distribution obtained for that random variable is the same as in P. Hence also the distribution over q is the same for both programs. □

5.2.2 Rejection Sampling With Heuristic Lookahead

As discussed in Section 2.3, sampling-based approaches to probabilistic inference estimate the conditional probability $p(q|e)$ of a query q given evidence e by randomly generating a large number of samples or possible worlds (cf. Algorithm 6). The algorithm starts by preparing the program L for sampling by applying the PMAGIC transformation. In the following, we discuss our choice of subroutine STPMAGIC (cf. Algorithm 7) that realizes likelihood weighted sampling. It is used in Algorithm 6, line 6 to generate individual samples. It iterates the stochastic consequence operator of Definition 5.3 until either a fixpoint is reached or the current sample is inconsistent with the evidence. Finally, if the sample is inconsistent with the evidence, it receives weight 0.

Algorithm 8 details the procedure used in line 12 of Algorithm 7 to sample from a given distributional clause. The function READTABLE returns the truth value of the probabilistic fact together with its weight. If the this truth value has not been tabled yet, it is computed and stored in the table. Note that `false` is returned when the outcome is not consistent with the evidence. Involved distributions, if not yet tabled, are sampled in line 5. If the distribution is infinite, SAMPLE

Algorithm 7 Sampling one interpretation together with an associated sample weight from the program L^* (used in Algorithm 6); the original program L is subsequently used by the functions MAYBEPROOF and MAYBEFAIL to ensure that the evidence e is consistent with the sample

```
1: function STPMAGIC(L*, L, e, Depth)
2:     T_pf ← ∅, T_dis ← ∅, w ← 1, I_old ← ∅, I_new ← ∅
3:     repeat
4:         I_old ← I_new
5:         for all (h :- body) ∈ L* do
6:             split body in B_PF (prob. facts) and B_L (the rest)
7:             for all grounding substitution θ such that B_Lθ ⊆ I_old do
8:                 s ← true
9:                 w_d ← 1
10:                while s ∧ (B_PF ≠ ∅) do
11:                    select and remove pf from B_PF
12:                    (b_pf, w_pf) ←READTABLE(pfθ, I_old, T_pf, T_dis, L, e, Depth)
13:                    s ← s ∧ b_pf
14:                    w_d ← w_d · w_pf
15:                end while
16:                if s then
17:                    if hθ ∈ e⁻ then return (I_new, 0)  ▷ check negative evidence
18:                    I_new ← I_new ∪ {hθ}
19:                    w ← w · w_d
20:                end if
21:            end for
22:        end for
23:    until I_new = I_old ∨ w = 0                ▷ Fixpoint or impossible evidence
24:    if e⁺ ⊆ I_new then return (I_new, w)       ▷ check positive evidence
25:    else return (I_new, 0)
26:    end if
27: end function
```

simply returns the sampled value. This case covers both continuous distributions, i.e., Gaussians, and infinite discrete distributions such as the Poisson. In the finite case, the algorithm aims at generating samples that are consistent with the evidence. Firstly, all possible choices that are inconsistent with the negative evidence are removed. Secondly, when there is positive evidence for a particular value, only that value is left in the distribution. Thirdly, it is checked whether each left value is consistent with the rest of the evidence. This consistency check is performed by a simple depth-bounded meta-interpreter. For positive evidence, it attempts a top-down proof of the evidence atom in the original program using the function MAYBEPROOF. Subgoals for which the depth bound is reached, as well as probabilistic facts that are not yet tabled, are assumed to succeed. If this results

Algorithm 8 Evaluating a probabilistic fact pf and storing the sampled value for subsequent calls. The function ComputePF(pf, T_{dis}) returns the truth value and the probability of pf according to the information in T_{dis}.

```
1: function ReadTable(pf, I, T_pf, T_dis, L, e, Depth)
2:     if pf ∉ T_pf then
3:         for all random variable h occurring in pf where h ∉ T_dis do
4:             if h ∼ D ∉ I then return (false, 0)
5:             if not Sample(h, D, T_dis, I, L, e, Depth) then return (false, 0)
6:             (b, w) ← ComputePF(pf,T_dis)
7:         end for
8:         if (b ∧ (pf ∈ e⁻)) ∨ (¬b ∧ (pf ∈ e⁺)) then
9:             return (false, 0)                          ▷ inconsistent with evidence
10:        end if
11:        extend T_pf with (pf, b, w)
12:    end if
13:    return (b, w) as stored in T_pf for pf
14: end function
```

in a proof, the value is consistent, otherwise it is removed. Similarly for negative evidence: in MaybeFail, subgoals for which the depth bound is reached, as well as probabilistic facts that are not yet tabled, are assumed to fail. If this results in failure, the value is consistent, otherwise it is removed. The *Depth* parameter allows one to trade the computational cost, associated with this consistency check, for a reduced rejection rate. Note that the modified distribution is normalized and the weight is adjusted in each of these three cases. The weight adjustment takes into account that the removed elements cannot be sampled. The adjustment is required as it can depend on the distributions sampled so far what elements are to be removed from the distribution sampled in Sample (the clause bodies of the distribution clause are instantiating the distribution).

5.3 Experiments

We set up experiments to answer the following questions:

Q1 Does the lookahead-based sampling improve the performance?

Q2 How do rejection sampling and likelihood weighting compare?

To answer the first question, we used a program that models an urn containing a random number of balls. The number of balls is uniformly distributed between 1 and 10 and each ball is either red or green with equal probability.

Algorithm 9 Sampling the value of h from the distribution $\mathcal{D}$. The function runs a meta interpreter in the program L to identify values that are inconsistent with the evidence e and the part of the interpretation that has been sampled so far.

procedure SAMPLE($\mathtt{h}, \mathcal{D},\ T_{dis},\ I,\ L,\ e,\ Depth$)
  $w_h \leftarrow 1$, $\mathcal{D}' \leftarrow \mathcal{D}$ ▷ Initial weight, temp. distribution
  if $\mathcal{D}' = [p_1 : \mathtt{a_1}, \ldots, p_n : \mathtt{a_n}]$ **then** ▷ finite distribution
    for $p_j : \mathtt{a_j} \in \mathcal{D}'$ where $\mathtt{dist\_eq}(\mathtt{h}, \mathtt{a_j}) \in e^-$ **do** ▷ remove neg. evidence
      $\mathcal{D}' \leftarrow$ NORM($\mathcal{D}' \setminus \{p_j : \mathtt{a_j}\}$)
      $w_h \leftarrow w_h \times (1 - p_j)$
    end for
    if $\exists v : \mathtt{dist\_eq}(\simeq(\mathtt{h}), \mathtt{v}) \in e^+$ and $p_j : \mathtt{a_j} \in \mathcal{D}'$ **then** ▷ pos. evidence
      $\mathcal{D}' \leftarrow [1 : \mathtt{a_j}]$
      $w_h \leftarrow w_h \times p_j$
    end if
    for $p_j : \mathtt{a_j} \in \mathcal{D}'$ **do** ▷ remove choices that make e^+ impossible
      if $\exists b \in e^+$: not MAYBEPROOF($b, Depth, I \cup \{\mathtt{dist\_eq}(\mathtt{h}, \mathtt{a_j})\}, L$) **or**
        $\exists b \in e^-$: not MAYBEFAIL($b, Depth, I \cup \{\mathtt{dist\_eq}(\mathtt{h}, \mathtt{a_j})\}, L$) **then**
        $\mathcal{D}' \leftarrow$ NORM($\mathcal{D}' \setminus \{p_j : \mathtt{a_j}\}$)
        $w_h \leftarrow w_h \times (1 - p_j)$
      end if
    end for
  end if
  if $\mathcal{D}' = \varnothing$ **then return** false ▷ no element in $\mathcal{D}$ is consistent with evidence
  sample x according to $\mathcal{D}'$ and extend T_{dis} with (h, x)
  return true
end procedure

```
numballs ~ uniform([1,2,3,4,5,6,7,8,9,10]).

ball(M) :- between(1, numballs, M).

color(B) ~ uniform([red, green]) :- ball(B).

drawnball(D) ~ uniform(L) :- draw(D), findall(B, ball(B), L).

draw(N) :- between(1, 8, N).

nogreen(0).

nogreen(D) :- dist_eq(≃(color(≃(drawnball(D)))), red),
              D2 is D - 1, nogreen(D2).
```

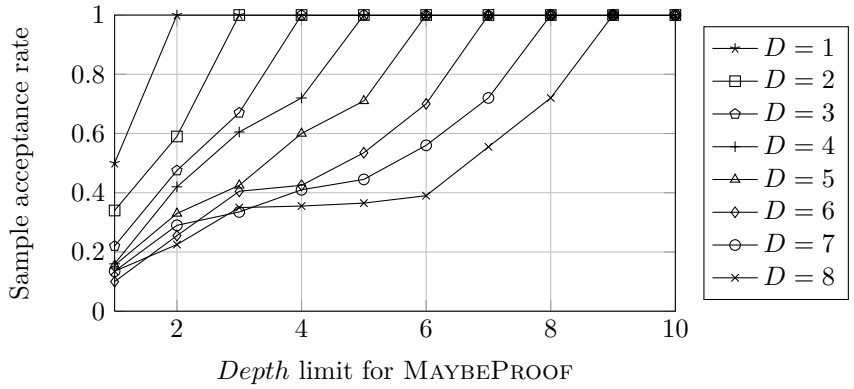
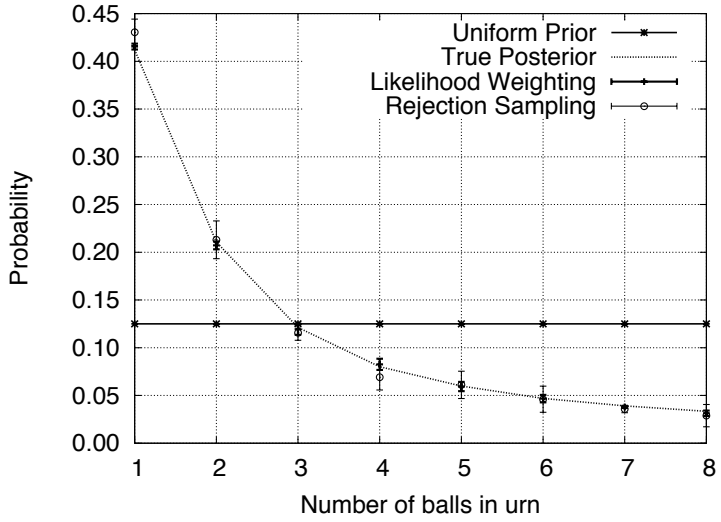


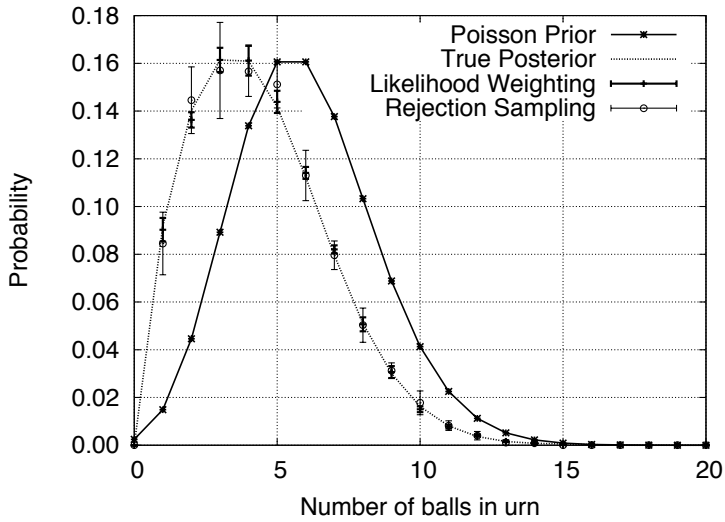
Figure 5.1: Sample acceptance rate when estimating the conditional probability $P(\texttt{dist\_eq}(\simeq(\texttt{color}(\simeq(\texttt{drawnball}(1)))), \texttt{red}) \mid \texttt{nogreen}(D))$ for $D = 1, 2, \ldots, 8$ and for $Depth = 1, 2, \ldots, 10$ using Algorithm 6. The acceptance rate is calculated by generating 200 samples and counting the number of samples that have non-zero weight (**Q1**).

We draw a ball eight times with replacement from the urn and observe its color. We also define the atom $\texttt{nogreen}(D)$ to be true if and only if we did not draw a green ball in draw 1 to D. We evaluated the query $P(\texttt{dist\_eq}(\simeq(\texttt{color}(\simeq(\texttt{drawnball}(1)))), \texttt{red}) \mid \texttt{nogreen}(D))$ for $D = 1, 2, \ldots, 8$. Note that the evidence implies that the first ball drawn is red, hence the probability of the query is 1. However, the number of steps required to prove that the evidence is inconsistent with drawing a green ball first increases with D. The larger the value of D is, the larger $Depth$ is required to reach a 100% acceptance rate for the sample as illustrated in Figure 5.1. It is clear that by increasing the depth limit, each sample will take longer to be generated. The $Depth$ parameter allows one to trade off convergence speed of the sampling and the time each sample needs to be generated. Depending on the program, the query, and the evidence there is an optimal depth for the lookahead.

To answer the second question, we used the standard example for BLOG [Milch et al., 2005a]. An urn contains an unknown number of balls, where every ball can be either green or blue with $p = 0.5$. When drawing a ball from the urn, we observe its color but do not know what ball it is. When we observe the color of a particular ball, there is a 20% chance to observe the wrong one, e.g. green instead of blue. We have some prior belief over the number of balls in the urn. If ten balls are drawn with replacement from the urn and we saw the color green ten times, what is the probability that there are n balls in the urn? We consider two different prior distributions: in the first case, the number of balls is uniformly distributed between 1 and 8, in the second case, it is Poisson-distributed with mean $\lambda = 6$.



(a) Uniform prior over $\{1, \ldots, 8\}$



(b) Poisson prior with mean 6 ($\lambda = 6$)

Figure 5.2: Results of the urn experiment with forward reasoning when using different priors over the number of balls in the urn. In each experiment ten balls with replacement were drawn and each time color green was observed. The graphs show the estimated posterior distribution over the number of balls in the urn together with true posterior and the prior distribution (**Q2**).

One run of the experiment corresponds to sampling the number N of balls in the urn, the color for each of the N balls, and for each of the ten draws both the ball drawn and whether or not the color is observed correctly in this draw. Once these values are fixed, the sequence of colors observed is determined. This implies that for a fixed number N of balls, there are $2^N \cdot N^{10}$ possible proofs. In case of the uniform distribution, exact PRISM inference can be used to calculate the probability for each given number of balls, with a total runtime of 0.16 seconds for all eight cases. In the case of the Poisson distribution, this is only possible up to 13 balls, with more balls PRISM runs out of memory. For inference using sampling, we generate 20,000 samples with the uniform prior and 100,000 with Poisson prior. We report average results over five repetitions. For these priors, PRISM generates 8,015 and 7,507 samples per second respectively, ProbLog backward sampling 708 and 510, BLOG 3,008 and 2,900, and our new forward sampling (with rejection sampling) 760 and 731. The results using our algorithm for both rejection sampling and likelihood weighting with $Depth = 0$ are shown in Figure 5.2. As the graphs show, the standard deviation for rejection sampling is much larger than the one for likelihood weighting.

The result of our experiments confirm that the lookahead step in the sampling algorithm increases the ratio of accepted samples and increases the performance of rejection sampling. However, We also found that running MaybeProof and MaybeFail in every sample step is too inefficient as the number of accepted samples per second in fact decreases with larger *Depth* values. However, in our experiments we compared a prototype implementation of our sampling algorithm with fully-developed and optimized implementations.

5.4 Related Work

Distributional clauses allow one to represent continuous variables and to reason about an unknown number of objects. In this regard this construct is related to languages such as BLOG and Church, however it is strongly embedded in a logic programming context. This embedding allowed us to propose also a novel inference method based on the combination of importance sampling and forward reasoning. This contrasts with the majority of probabilistic logic programming languages, which are typically based on backward reasoning (possibly enhanced with tabling [Sato and Kameya, 2001; Mantadelis and Janssens, 2010]). Furthermore, only few of these techniques employ sampling; see [Kimmig et al., 2011] for a Monte Carlo approach using backward reasoning. Another key difference with the existing probabilistic logic programming approaches is that the described inference method is able to handle evidence. This is due to the magic set transformation that targets the generative process towards the query and evidence by instantiating only the relevant random variables.

P-log [Baral et al., 2009] is a probabilistic language based on Answer Set Prolog (ASP). It uses a standard ASP solver for inference and it is thus based on forward reasoning, but without the use of sampling. Magic sets are also used in probabilistic Datalog [Fuhr, 2000], as well as in Dyna [Eisner et al., 2005], a probabilistic logic programming language that is based on rewrite rules and uses forward reasoning. However, neither of them uses sampling. Furthermore, Dyna and PRISM require the exclusive-explanation assumption. This assumption states that no two different proofs for the same goal can be true simultaneously, that is, they have to rely on at least one basic random variable with a different outcome in each proof. Distributional clauses (and the ProbLog language) do not impose such a restriction. Other related work includes MCMC-based sampling algorithms such as the approach for SLP [Angelopoulos and Cussens, 2003]. Church's inference algorithm is based on MCMC as well, and also BLOG is able to employ MCMC.

5.5 Conclusions And Future Work

The inference algorithm we use for distributional programs can be extended in many ways. Currently, the algorithm does not store the dependencies identified by the lookahead step for subsequent sample generations. For instance, if the evidence can be generated by the program in exactly one possible way, the algorithm has to repeat the lookahead step for each sample in order to identify this determinism. Hence the algorithm does not retain all information it possibly could. The main challenge in this context is to find a representation that can efficiently store such information without grounding the whole program. Furthermore, it is interesting to apply standard logic programming techniques to translate the program into an equivalent program, which is maybe more suitable for sampling. Program transformation and program specialization, for instance, could be used to rule out parts of the program that are inconsistent with the evidence. This preprocessing operation minimizes the work in each sampling step, which in turn increases the sampling speed. Apart from focusing on the logical aspect of the model, one can exploit more efficient sampling techniques. Currently, the inference mechanism is based on likelihood weighting, although using adaptive importance sampling methods is also worth investigating. In addition, using an inference algorithm from the class of Markov Chain Monte Carlo approaches (MCMC) is interesting to explore. Such MCMC approaches are widely used in probabilistic programming languages that are not based on logic programming such as Church [Goodman et al., 2008] or BLOG [Milch et al., 2005a]. Together these enhancements will enable one to apply distributional programs on typical robotics tasks such as localization and tracking, which require a fast inference mechanism in a Bayesian setting [Thrun et al., 2005; De Laet, 2010].

Conclusions of Part II

In this part of the thesis we studied how ProbLog can be extended with continuous distributions.

In Chapter 4 we introduced *Hybrid ProbLog*. This extension of ProbLog allows one to define the continuous distribution over ground instances of so-called *continuous facts*. We developed an exact inference algorithm that is able to identify the relevant areas of the $\mathbb{R}^n$ needed for an automatic query-dependent discretization. In order to make exact inference tractable, the number of possible operations on the continuous facts are limited. This language is best suited for domains that have only a few continuous values and where the model does not require comparison operators between the values.

In Chapter 5 we introduced a more expressive extension of ProbLog in the form of *distributional programs*. The building blocks of this formalism are so-called *distributional clauses*, a generalization of annotated disjunctions (cf. Section 3.3). In order to calculate the conditional probability of queries, we propose an inference algorithm based on rejection sampling. We used the magic sets transformation to guide the sampling process to the relevant parts of the program, i.e., avoiding the instantiation of ground atoms that do not influence the probability. Moreover, we exploit the deterministic information in the program to remove choices that contradict the evidence from the distributional clauses. With regard to future work, it would be interesting to consider evidence on continuous distributions as it is currently restricted to finite distributions. Program analysis and transformation techniques would further optimize the program with respect to the evidence and query could be used to increase the sampling speed.

It is worth noting that the syntax of Hybrid ProbLog and distributional clauses slightly deviate. The former aims at *extending* ProbLog and hence keeps the syntax close to that of ProbLog, while the latter one is a synthesis of different language concepts. It is possible to translate a Hybrid ProbLog program into an equivalent distributional program but not vice versa. Distributional programs are more expressive, which also explains why they require a sampling-based inference algorithm.

Part III

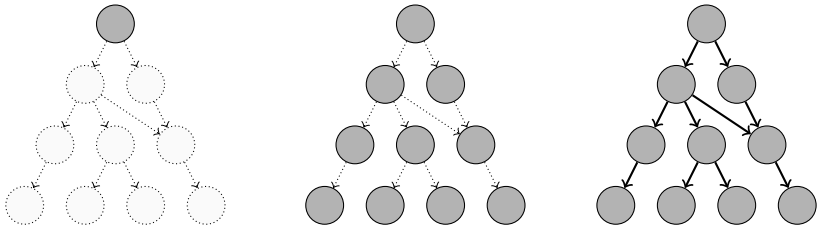
Parameter Learning

Outline of Part III

This part of the thesis is devoted to learning the parameters of a ProbLog theory. We introduce two different approaches to this task. Together they cover the three main learning settings from the logical and relational learning literature: *learning from entailment*, *learning from interpretations* and *learning from proofs* (cf. [De Raedt, 2008]).

Chapter 6 introduces *LFE-ProbLog* that is able to learn from entailment as well as from proofs (cf. Figure 5.3(a) and 5.3(c)). The key idea is to translate the learning task into a *logistic regression* problem and to use a standard gradient search for finding an optimal parameter configuration.

Chapter 7 introduces *LFI-ProbLog* that learns from complete as well as from partial interpretations (see Figure 5.3(b)). This learning approach assumes a generative model that allows us to use the well-known EM algorithm for training. The advantage of this setting over logistic regression is the fact that the resulting parameters can be interpreted, while logistic regression treats the model as a "black box" and merely optimizes the prediction of the model.



(a) Learning from entailment

(b) Learning from interpretations

(c) Learning from proofs

Figure 5.3: One can classify the learning settings in probabilistic logic learning (PLL) by the amount of information each training example contains; dotted elements are not contained in the example (adapted from [Kersting, 2006, Figure 2.6]).

Chapter 6

Learning from Probabilistic Entailment*

Many real-world applications involve managing vast volumes of uncertain data. Such databases arise, for example, when integrating data from various sources or when analyzing social, biological, and chemical networks. They can occur within privacy-preserving data mining where only aggregated data is available as well as within pervasive computing. These are only some of the applications showing the abundance of uncertain data residing in databases. Traditional databases do not allow one to deal with uncertainty. Hence probabilistic extensions of databases are crucial for managing and mining uncertain 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]. Conversely, little attention was directed towards 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]. These probabilistic annotations are then used to define and compute the probability of derived facts, given background knowledge specifying further relationships or predicates. As an 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 consist of a set of facts, each holding with a particular probability [Antanas et al., 2009]. These facts might state, for example, the probability that a certain object in the scene is a pedestrian who is

*This chapter presents joint work with Angelika Kimmig, Kristian Kersting and Luc De Raedt published in [Gutmann et al., 2008, 2010b]. The gradient derivation and the presentation of the algorithms have been extended and adopted towards non-ground facts.

walking in a particular direction. The task could then be, for instance, recognizing certain types of traffic violations. Background knowledge might be used to specify different forms of traffic rules. As second 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 where edges are represented by probabilistic facts about the biological entities [Sevon et al., 2006; De Raedt et al., 2007]. 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 life scientists with better insights into the mutual relationships between the queried entities.

This introduces a novel probabilistic database setting for parameter learning from examples together with their target probability (cf. Definition 6.1). The task is to find parameters that minimize the least squared error with respect to these examples. The examples themselves can be either queries or proofs, where a proof is a conjunction of all facts in the database needed to prove a query. Although ProbLog is a probabilistic programming language, it can be considered as a generalization of a probabilistic database. Both the problem setting introduced in this chapter and the solution developed can easily be integrated in other probabilistic databases. Furthermore, this chapter introduces an efficient learning algorithm, LFE-ProbLog.[1]

This chapter has three core contributions: (1) a novel probabilistic database setting for parameter learning from examples together with their target probability, (2) detailed study of the intuitive meaning of the gradient and (3) an efficient learning algorithm to evaluate the gradient based on BDDs.

We proceed as follows. We formally introduce the parameter estimation problem for probabilistic databases in Section 6.1. Section 6.2 contains the derivation of the gradient, which is used in Section 6.3 to construct the parameter learning algorithm LFE-ProbLog. In Section 6.4 we discuss how the error function can be modified to account for imbalanced datasets. Before concluding, we present the results of an extensive set of experiments on real-world data sets in Section 6.5 as well as related work in Section 6.6.

6.1 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 6.1 (Parameter Learning in Probabilistic Databases)**.** ***Given*** *is a set of examples E, a probabilistic database or probabilistic logic theory D, a probabilistic*

[1] Stands for "learning from probabilistic entailment for ProbLog". In previously published papers it had been called LeProbLog but it was renamed for consistency reasons.

coverage relation $P(e|D)$ that denotes the probability that the database D covers the example $e \in E$ and a scoring function. The goal is to ***find*** *parameters of D such that the scoring is optimal.*

The key difference with purely logical learning approaches is that the coverage relation becomes probabilistic. Furthermore, within probabilistic logical and relational learning the following three settings are typically considered: learning from queries, learning from proofs and learning from interpretations (cf. Figure 5.3). 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. Conversely, a proof does not only show *what* was proven but also *how* it was realized.

As an example consider a probabilistic context-free grammar. 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 more complex 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 complex to apply this to models 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.

In the setting considered in this chapter, the examples themselves have associated probabilities. Such examples naturally arise in various applications. For instance, text extraction algorithms return the confidence, experimental data is often averaged over several runs. As one illustration consider populating a probabilistic database of genes from MEDLINE[2] abstracts 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,

[2]MEDLINE (Medical Literature Analysis and Retrieval System Online) is a bibliographic database of life sciences and biomedical information compiled by the United States National Library of Medicine. It is available through PubMed `http://www.ncbi.nlm.nih.gov/pubmed/`.

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 with 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. In ProbLog the key difference between learning from entailment and learning from proofs is that the DNF formula for a proof is a conjunction of literals, while it is general DNF formula in the case of learning from queries. To the best of our knowledge, this is the first time that learning from proofs and learning from entailment are integrated in one setting.

The setting considered in this chapter differs from the usual statistical relational learning setting with respect to the characteristics of the underlying generative model. 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 since they are variants of probabilistic context-free grammars. As a consequence, they implicitly define a generative model 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 hence does not meet these conditions, as several different facts for the same background knowledge predicate can be true in any possible world. Working with probabilistic training examples has been recently proposed by Chen et al. [2008]. However, in their work, the probabilities associated with examples are viewed as specifying the degree of being sampled from some distribution, thus employing a generative model at the level of examples or queries, which does not hold in our case. Furthermore, Chen et al. consider only learning from entailment and not from proofs.

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

Definition 6.2 (Learning from probabilistic entailment)**.** ***Given*** *a ProbLog theory* $T = F \cup BK$*, where the probabilistic facts* F *have* unknown *parameters* $\mathbf{p} = \langle p_1, ..., p_N \rangle$ *and a set of training examples* $\{q_i, \tilde{p}_i\}_{i=1}^{M}$*, where each* q_i *is a query or proof and* $\tilde{p}_i$ *is the k-probability of* q_i*,* ***find*** *probabilities* $\widehat{\mathbf{p}} = \langle \widehat{p_1}, \ldots, \widehat{p_N} \rangle$ *such that*

the Mean Squared Error of the program on the training set is minimized, where

$$\mathrm{MSE}(T) := \frac{1}{M}\sum_{i=1}^{M}\left(P_k^T(q_i) - \tilde{p}_i\right)^2 \ . \tag{6.1}$$

Note that in this definition we have chosen the k-probability (cf. Eq. (3.9)) as probabilistic coverage relation since this allows for maximal flexibility. For $k = 1$, it is the probability of the best explanation of the query q, for $k = \infty$ it corresponds to the success probability of q. Intermediate values can be used to trade off accuracy for speed. We assume the target probabilities $\tilde{p}_i$ to be given by a domain expert and do not make any explicit assumption as of how they were sampled based on a generative process from the ProbLog theory T.

Our setting is related to the one considered by Kwoh and Gillies [1996] to the degree 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 where the examples are independent of one another. While in our case all the observations are binary, Kwoh and Gillies use training examples that contain several variables. They 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. Afterwards, as long as the error does not converge, the gradient of the error function is calculated, scaled by the learning rate η, and then subtracted from the current parameters.

An unconstrained gradient descent search based on (6.1) cannot be used since the algorithm does not take into account that the search has to be confined to probabilities only. Hence adding the gradient vector to the current probability vector might result in values outside the $[0, 1]$ interval. This can be resolved, for instance, by using a constrained gradient descent search and project the values after each gradient step onto the $[0, 1]^N$ space (cf. [Rosen, 1960, 1961]).

We use a different technique that allows for an unconstrained gradient search: we apply a sigmoid function $\sigma : \mathbb{R} \to (0, 1)$ to the search space similarly to *logistic regression*. Such a function has an s-shaped graph as shown in Figure 6.1. We will use the function

$$\sigma_s(a) := \frac{1}{1 + \exp(-s \cdot a)} \ , \tag{6.2}$$

where the parameter s determines the slope of the curve. For small s, the slope is flat, which in turn can cause a slow convergence. While for large s, the slope is

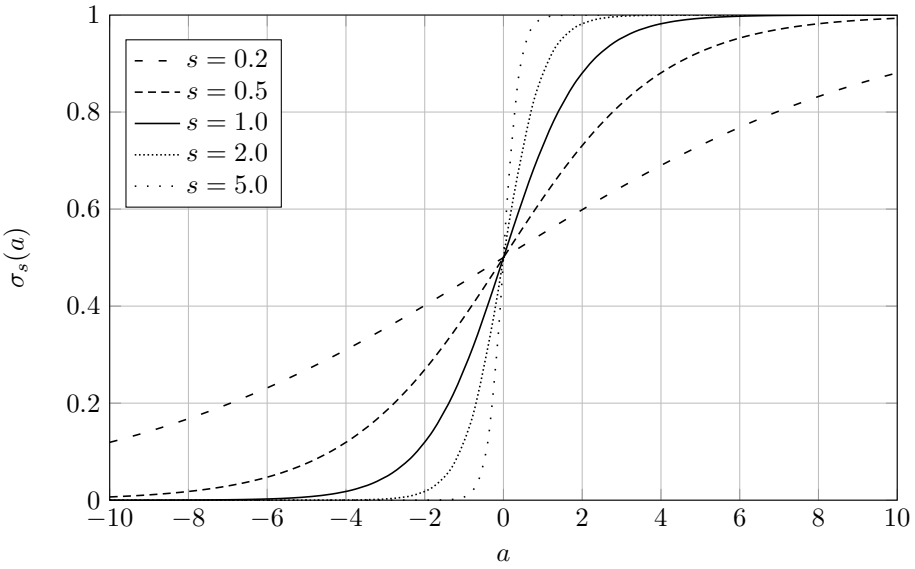


Figure 6.1: The sigmoid function $\sigma_s(a) = (1 + \exp(-s \cdot a))^{-1}$ with different slope parameter $s > 0$. All graphs pass through the point $(0, 0.5)$.

steep, which makes the gradient search more susceptible to local optima. In the rest of this chapter, we assume $s = 1$ and write $\sigma(a)$ whenever the choice of s is clear. In order to apply the sigmoid transformation, we represent the probability of each probabilistic fact $p_j :: f_j$ as $p_j = \sigma(a_j)$, that is, $a_j = \sigma^{-1}(p_j)$. In turn, we have to propagate this representation in the definition of the k-best probability and to the probability of a possible world. Please note that $\sigma_s(a)$ is symmetric with respect to the point (0, 0.5). This allows one to express the complementary probability $1 - \sigma_s(a)$ more compactly as

$$1 - \sigma_s(a) = \sigma_s(-a)$$

The proof for this property can be found in Appendix B. Before proceeding with deriving the gradient of the MSE, we use the following example to illustrate the learning setting we study in this chapter.

Example 6.1 (Parameter learning for the ALARM-4 program)**.** *Consider the* ALARM-4 *program, where the fact probabilities p_i are unknown.*

$$F = \{p_1 :: \texttt{burglary},\ p_2 :: \texttt{earthquake},\ p_3 :: \texttt{hears\_alarm(X)}\}$$

$$\begin{aligned} BK = \{&\texttt{person(mary)},\\ &\texttt{person(john)},\\ &\texttt{person(alice)},\\ &\texttt{person(bob)},\\ &\texttt{alarm} :- \texttt{burglary}\\ &\texttt{alarm} :- \texttt{earthquake}\\ &\texttt{calls(X)} :- \texttt{person(X)},\ \texttt{alarm},\ \texttt{hears\_alarm(X)}\} \end{aligned}$$

The goal is to estimate p_1, p_2, p_3 from the training set comprised of the following examples:

$$q_1 = \texttt{alarm} \qquad \tilde{p}_1 = 0.4$$

$$q_2 = \texttt{calls(john)} \qquad \tilde{p}_2 = 0.2$$

$$q_3 = \texttt{calls(mary)} \vee \texttt{calls(john)} \qquad \tilde{p}_3 = 0.3$$

$$q_4 = \texttt{burglary} \wedge \texttt{hears\_alarm(john)} \qquad \tilde{p}_4 = 0.15$$

After translating the fact probabilities using the $\sigma(\cdot)$ function, the goal is to estimate $a_1, a_2, a_3 \in \mathbb{R}$ where

$$F = \{\sigma(a_1) :: \texttt{burglary},\ \sigma(a_2) :: \texttt{earthquake},\ \sigma(a_3) :: \texttt{hears\_alarm(X)}\}$$

As this example illustrates, one can use atomic queries as well as conjunctions and disjunctions of atoms as training data. This is due to ProbLog's inference algorithm based on SLD resolution. The training example $(q_4, 0.15)$ shows how one can use a conjunctive query for encoding the proof of `calls(john)` that uses the clause `calls(X)` :− `person(X)`, `alarm`, `hears_alarm(X)` and then the clause `alarm` :− `burglary`.

In the following section we formally derive the gradient of the MSE and give an intuitive meaning of the resulting values. Afterwards, in Section 6.3, we introduce an efficient algorithm based on BDDs to compute the gradient and use it in a standard gradient-descent scheme to solve the learning task for a ProbLog program.

6.2 Deriving The Gradient

We derive the gradient of the MSE in three steps. Starting from the gradient of the possible world probability in Lemma 6.1, we derive the gradient of the success probability of a query in Lemma 6.2. This implicitly yields the gradient of the k-best probability, which we then use to obtain the gradient of the MSE in Theorem 6.1. To simplify the notation we introduce the following symbols

$$\delta^+_{j,L} := \big|\{\text{all ground instances of } f_j\} \cap L\big| \tag{6.3}$$

$$\delta^-_{j,L} := \big|\{\text{all ground instances of } f_j\} \setminus L\big| \tag{6.4}$$

where $L \subseteq L^T$ is a possible world and j is the index of the probabilistic fact f_j. In other words, $\delta^+_{j,L}$ counts the true ground instances of f_j in L and $\delta^-_{j,L}$ counts the false ground instances of f_j in L.

Example 6.2 (Meaning of $\delta^+_{j,L}$ and $\delta^-_{j,L}$)**.** *The probabilistic fact f_3 in the* ALARM-4 *program has four ground instances* `hears_alarm(mary)`*,* `hears_alarm(john)`*,* `hears_alarm(alice)`*, and* `hears_alarm(bob)`*. For the possible world $L_1 =$ {`hears_alarm(john)`, `burglary`}, for instance, the count δ^+_{3,L_1} is 1 as there is only one ground instance of f_3 true in L_1. Hence $\delta^-_{3,L_1} = 4 - \delta^+_{3,L_1} = 3$. For the possible world $L_2 =$ {`hears_alarm(john)`, `hears_alarm(alice)`} the count δ^+_{3,L_2} is 2 and $\delta^-_{3,L_2} = 2$.*

6.2.1 Gradient of Possible World Probability

Lemma 6.1 (Gradient of Possible World Probability)**.** *Given a possible world $L \subseteq L^T$ one can compute the gradient of the probability of L with respect to a_j as follows*

$$\frac{\partial}{\partial a_j} P^T(L) = \underbrace{\Big[\delta^+_{j,L} \cdot \sigma(-a_j) - \delta^-_{j,L} \cdot \sigma(a_j)\Big]}_{\textit{Factor 1}} \cdot \underbrace{P^T(L)}_{\textit{Factor 2}} \tag{6.5}$$

where $\sigma(a_j) = p_j$ governs the probability of the probabilistic fact f_j.

Please note that the δ in (6.5) ensure the parameter-tying between different ground instances of a probabilistic fact. Instead of considering individual ground instances, one compresses them into two numbers: the positive and the negative counts.

Before proving Lemma 6.1, let us highlight some properties of the gradient (6.5), which give a more intuitive meaning to the resulting values. Each root of the gradient[3] points to a local optima in $P^T(L)$. As the gradient consists of two factors,

[3] A value x is called a root of a function $f(x)$ if $f(x) = 0$

we can find the gradient's roots by finding roots of either factor. Due to the sigmoid function, Factor 2 is always larger than 0. Hence the local optima of $P^T(L)$ solely depend on Factor 1.

1. If $\delta^+_{j,L} > 0$ and $\delta^-_{j,L} > 0$
   Factor 1 is zero if and only if $\sigma(a_j) = \delta^+_{j,L}/(\delta^+_{j,L} + \delta^-_{j,L})$. Hence any gradient ascent algorithm tends to set a_j such that the resulting probability $\sigma(a_j)$ is the relative frequency of true ground atoms of f_j over all ground atoms of f_j.
2. If $\delta^+_{j,L} > 0$ and $\delta^-_{j,L} = 0$
   Factor 1 is larger than 0 for all a_j. Hence any gradient ascent algorithm tends to increase the value of a_j towards ∞. In turn, the resulting probability $\sigma(a_j)$ will be set closer to 1 in each iteration.
3. If $\delta^+_{j,L} = 0$ and $\delta^-_{j,L} > 0$
   Factor 1 is smaller than 0 for all a_j. Hence any gradient ascent algorithm tends to decrease the value of a_j towards $-\infty$. In turn, the resulting probability $\sigma(a_j)$ will be set closer to 0 in each iteration.

Probabilistic facts that do not have any ground instances ($\delta^+_{j,L} = \delta^-_{j,L} = 0$) can be removed from the program without changing the probability of a possible world. The shape of the gradient's graph is a mirrored "s" as can be seen in Figure 6.2. The slope of the graph depends on the slope of the sigmoid function (in our case it is always 1) and the total number of ground instances of f_j. The function value is limited and converges to $\delta^+_{j,L}$ for $a_j \to -\infty$ and to $-\delta^-_{j,L}$ for $a_j \to \infty$.

Example 6.3 (Gradient of Possible World Probability)**.** *The probabilistic fact* $f_3 =$ `hears_alarm(X)` *in the* ALARM-4 *program has four possible ground instances. Figure 6.2 shows the graph of Factor 1 for all configurations of these ground instances. If* $\delta^+_{3,L} = 4$*, i.e., there are only* true *ground instances, the value is always larger than 0. For* $\delta^+_{3,L} = \delta^-_{3,L} = 2$*, the graph crosses the x-axis at 0, that is, the probability of a possible world has a local optimum at* $a_3 = 0$*, that is, a gradient search will set* $p_3 = \sigma(0) = 0.5$*. For* $\delta^+_{3,L} = 3$ *and* $\delta^-_{3,L} = 1$*, the graph crosses the x-axis at* $\sigma^{-1}(3/(3+1)) \approx 1.01$*.*

Proof. We prove Lemma 6.1 by replacing $P^T(L)$ with its definition (cf. (3.4) in Chapter 3) and write the complementary probability $1 - \sigma(a_j)$ as $\sigma(-a_j)$ (cf. Lemma B.1).

$$\frac{\partial}{\partial a_j} P^T(L) = \frac{\partial}{\partial a_j} \left(\prod_{f_i\theta_{ik} \in L} \sigma(a_i) \prod_{f_i\theta_{ik} \in L^T \setminus L} \sigma(-a_i) \right)$$

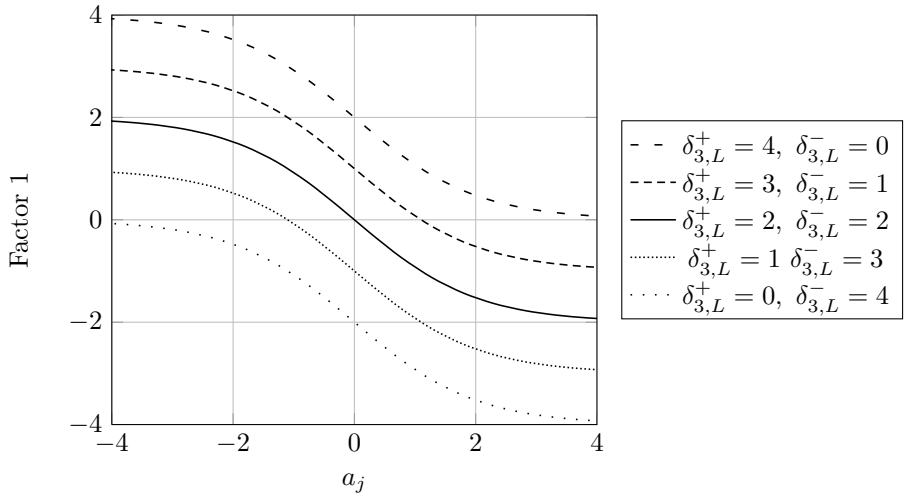


Figure 6.2: The graph of Factor 1 in Lemma 6.1 for all possible combinations of 4 ground instances of the probabilistic fact f_3 in the ALARM-4 program. Notice that $\delta^+_{3,L} + \delta^-_{3L}$ is always 4.

After rearranging the products where $\delta^+_{j,L}$ and $\delta^-_{j,L}$ are chosen accordingly, the products that do not depend on a_j can be treated as constant factor.

$$= \left(\frac{\partial}{\partial a_j} \sigma(a_j)^{\delta^+_{j,L}} \cdot \sigma(-a_j)^{\delta^-_{j,L}} \right) \prod_{\substack{f_i\theta_{ik} \in L \\ i \neq j}} \sigma(a_i) \prod_{\substack{f_i\theta_{ik} \in L^T \setminus L \\ i \neq j}} \sigma(-a_i) \tag{6.6}$$

We have to distinguish **three cases** for the gradient. Case 1 is $\delta^+_{j,L} > 0,\ \delta^-_{j,L} > 0$, case 2 is $\delta^+_{j,L} > 0,\ \delta^-_{j,L} = 0$ and case 3 is $\delta^+_{j,L} = 0,\ \delta^-_{j,L} > 0$. If both would be 0, we could remove the probabilistic fact a_j from the program as there are no ground instances. Please note that the gradient of $\sigma_s(a)$ can be expressed as $\partial/\partial a\ \sigma_s(a) = s \cdot \sigma_s(a) \cdot (1 - \sigma_s(a))$ (cf. Lemma B.1).

Case 1, $\delta^+_{j,L} > 0,\ \delta^-_{j,L} > 0$: applying the product and chain rule on (6.6) yields

$$\begin{aligned} &= \Big[\delta^+_{j,L} \cdot \sigma(a_j)^{\delta^+_{j,L}-1} \cdot \sigma(-a_j)^{\delta^-_{j,L}} \cdot 1 \cdot \sigma(a_j) \cdot \sigma(-a_j) \\ &\quad - \delta^-_{j,L} \cdot \sigma(a_j)^{\delta^+_{j,L}} \cdot \sigma(-a_j)^{\delta^-_{j,L}-1} \cdot 1 \cdot \sigma(a_j) \cdot \sigma(-a_j) \Big] \cdot \\ &\quad \prod_{\substack{f_i\theta_{ik} \in L \\ i \neq j}} \sigma(a_i) \prod_{\substack{f_i\theta_{ik} \in L^T \setminus L \\ i \neq j}} \sigma(-a_i) \end{aligned}$$

Now we simplify the term in the parentheses,

$$= \left[\delta^+_{j,L} \cdot \sigma(-a_j) - \delta^-_{j,L} \cdot \sigma(a_j)\right] \cdot \sigma(a_j)^{\delta^+_{j,L}} \cdot \sigma(-a_j)^{\delta^-_{j,L}} \cdot \prod_{\substack{f_i\theta_{ik} \in L \\ i \neq j}} \sigma(a_i) \prod_{\substack{f_i\theta_{ik} \in L^T \setminus L \\ i \neq j}} \sigma(-a_i)$$

reorganize the products

$$= \left[\delta^+_{j,L} \cdot \sigma(-a_j) - \delta^-_{j,L} \cdot \sigma(a_j)\right] \prod_{f_i\theta_{ik} \in L} \sigma(a_i) \prod_{f_i\theta_{ik} \in L^T \setminus L} \sigma(-a_i)$$

and apply the definition of $P^T(L)$ again

$$= \left[\delta^+_{j,L} \cdot \sigma(-a_j) - \delta^-_{j,L} \cdot \sigma(a_j)\right] \cdot P^T(L) \ .$$

Case 2, $\delta^+_{j,L} > 0,\ \delta^-_{j,L} = 0$: applying product and chain rule on (6.6) yields

$$= \left[\delta^+_{j,L} \cdot \sigma(a_j)^{\delta^+_{j,L}-1} \cdot 1 \cdot \sigma(a_j) \cdot \sigma(-a_j)\right] \cdot \prod_{\substack{f_i\theta_{ik} \in L \\ i \neq j}} \sigma(a_i) \prod_{\substack{f_i\theta_{ik} \in L^T \setminus L \\ i \neq j}} \sigma(-a_i)$$

which can be simplified, similarly as in Case 1, to

$$= \left[\delta^+_{j,L} \cdot \sigma(-a_j)\right] \cdot P^T(L)$$

Since $\delta^-_{j,L} = 0$, subtracting $\delta^-_{j,L} \cdot \sigma(a_j)$ does not change the value. Hence

$$= \left[\delta^+_{j,L} \cdot \sigma(-a_j) - \delta^-_{j,L} \cdot \sigma(a_j)\right] \cdot P^T(L) \ .$$

Case 3, $\delta^+_{j,L} = 0,\ \delta^-_{j,L} > 0$ is analogous to Case 2. □

6.2.2 Gradient of Success Probability

The intuitive meaning of the gradient $\partial/\partial a_j \; P_s^T(q_i)$ is as follows. Each possible world, where the query q_i is true and which has more positive than negative ground instances of f_j, contributes positively to the gradient. In turn, this "pushes" the gradient search towards increasing a_j. Similarly, each possible world, where the query q_i is true and which has more negative than positive ground instances of f_j, "pushes" the gradient search towards decreasing a_j. Possible worlds where q_i is false do not influence the gradient.

The success probability $P_s^T(q)$ of a query q is the sum of the probabilities of all possible worlds, where the query is true (cf. (3.6)). Hence we can easily derive the gradient of $P_s^T(q_i)$ using Lemma 6.1.

Lemma 6.2 (Gradient of Success Probability)**.** *After applying the sigmoid transformation, the gradient of the success probability with respect to a_j is*

$$\frac{\partial P_s^T(q_i)}{\partial a_j} = \sum_{\substack{L \subseteq L^T \\ L \models q_i}} \left[\delta_{j,L}^+ \cdot \sigma(-a_j) - \delta_{j,L}^- \cdot \sigma(a_j) \right] \cdot P^T(L) \;\; , \tag{6.7}$$

where $\delta_{j,L}^+$ and $\delta_{j,L}^-$ are defined as in Lemma 6.1.

Proof.

$$\begin{aligned}
\frac{\partial P_s^T(q_i)}{\partial a_j} &= \frac{\partial}{\partial a_j} \sum_{\substack{L \subseteq L^T \\ L \models q_i}} P^T(L) && \text{by (3.6)} \\
&= \sum_{\substack{L \subseteq L_T \\ L \models q_i}} \frac{\partial}{\partial a_j} P^T(L) && \text{by sum rule} \\
&= \sum_{\substack{L \subseteq L^T \\ L \models q_i}} \left[\delta_{j,L}^+ \cdot \sigma(-a_j) - \delta_{j,L}^- \cdot \sigma(a_j) \right] \cdot P^T(L) && \text{by Lemma 6.1}
\end{aligned}$$

□

Note that one cannot obtain the gradient of the k-best probability $P_k^T(q_i)$ (cf. (3.9)) in a similar manner since $\text{bestproofs}_k(q_i, T)$ depends on the current values of the fact probabilities. In turn, the index set over which the sum in $P_k^T(q_i)$ iterates can change when the value of a_j is changed. We propose an approximation for this gradient where we assume the set of best proofs to be fixed and computed using the fact probabilities obtained in the previous gradient descent step.

Example 6.4 (k best proofs)**.** *Consider the following ProbLog theory that contains two proofs for the query* `alarm`*:*

$$\begin{aligned} F = \{&\sigma(a_1){::}\ \texttt{burglary},\\ &\sigma(a_2){::}\ \texttt{earthquake}\}\\ BK = \{&\texttt{alarm} :\!-\ \texttt{burglary},\\ &\texttt{alarm} :\!-\ \texttt{earthquake}\}\ . \end{aligned}$$

The best proof ($k = 1$) depends on the current values of the fact probabilities $\sigma(a_1)$ and $\sigma(a_2)$, that is,

$$\text{bestproofs}_1(q_i, T) = \begin{cases} \{\texttt{burglary}\} & \text{if } a_1 \geq a_2 \\ \{\texttt{earthquake}\} & \text{otherwise} \end{cases} .$$

If k is large enough, in this case for $k = 2$, the result of bestproofs_k *is independent of the a_j since all possible proofs are obtained.*

Since $\text{bestproofs}_k(q_i, T)$ is a step function (it is step-wise constant) its gradient can have discontinuities at the borders where the set of proofs change. Computing this gradient is computationally expensive and we avoid it by approximating the true gradient of the k-best probability of a query as follows:

$$\frac{\partial P_k^T(q_i)}{\partial a_j} = \frac{\partial}{\partial a_j} \sum_{\substack{L \subseteq L^T \\ L \in \text{bestproofs}_k(q_i, T)}} P^T(L) \ .$$

We treat $\text{bestproofs}_k(q_i, T)$ as constant, that is, independent of a_j while it is actually depending on a_j. We assume the set of best proofs to be computed using the a_j values obtained in the previous iteration of the gradient descent search. On the one hand, this step in the derivation introduces an error. On the other hand, it simplifies the next steps and allows us to apply the sum rule and to proceed as in the proof of Lemma 6.2:

$$\begin{aligned} &\approx \sum_{\substack{L \subseteq L^T \\ L \in \text{bestproofs}_k(q_i, T)}} \frac{\partial}{\partial a_j} P^T(L) \\ &= \sum_{\substack{L \subseteq L^T \\ L \in \text{bestproofs}_k(q_i, T)}} \left[\delta_{j,L}^{+} \cdot \sigma(-a_j) - \delta_{j,L}^{-} \cdot \sigma(a_j)\right] \cdot P^T(L) \ . \end{aligned}$$

Algorithm 11, which computes the gradient of the MSE, compensates for this error by repeatedly searching the set of k best proofs in every iteration before computing the gradient of P_k^T. This raises the questions as to what extent this approach is justified and whether is it necessary to perform this expensive step in every iteration. As the experimental results show, keeping the set of proofs fixed (respectively the BDDs representing this set) does not change the results when k is sufficiently large such that there is an overlap between the proofs.

6.2.3 Gradient of Mean Squared Error

By now we are ready to derive the gradient of the MSE. In the next section we will then show how to compute it efficiently based on binary decision diagrams.

Theorem 6.1 (Gradient of MSE)**.** *The gradient of the MSE with respect to a_j is*

$$\frac{\partial}{\partial a_j}\,\mathrm{MSE}(T) = \frac{2}{M}\sum_{i=1}^{M} \underbrace{\left(P_k^T(q_i) - \tilde{p}_i\right)}_{Part\ 1} \cdot \underbrace{\frac{\partial\ P_k^T(q_i)}{\partial a_j}}_{Part\ 2} \quad . \tag{6.8}$$

Proof. One can derive the gradient (6.8) starting from the definition of the mean squared error (cf. (6.1)) as follows:

$$\begin{aligned}
\frac{\partial}{\partial a_j}\,\mathrm{MSE}(T) &= \frac{\partial}{\partial a_j}\left(\frac{1}{M}\sum_{i=1}^{M}\left(P_k^T(q_i) - \tilde{p}_i\right)^2\right) \\
&= \frac{1}{M}\sum_{i=1}^{M}\left(\frac{\partial}{\partial a_j}\left(P_k^T(q_i) - \tilde{p}_i\right)^2\right) && \text{by sum rule} \\
&= \frac{2}{M}\sum_{i=1}^{M} \underbrace{\left(P_k^T(q_i) - \tilde{p}_i\right)}_{\text{Part 1}} \cdot \underbrace{\left(\frac{\partial P_k^T(q_i)}{\partial a_j}\right)}_{\text{Part 2}} && \text{by chain rule}
\end{aligned}$$

□

Part 1 is the k-best probability of the training example q_i under the current model parameters compared to the target success probability. Intuitively, this factor weights the influence of each query gradient (cf. Part 2), that is, for small differences the influence of Part 2 on the gradient of the MSE is small, while for larger differences the influence of the query gradient grows. Hence for training examples $(q_i, \tilde{p}_i)$, where the predicted probability $P_k^T(q_i)$ is close to the target $\tilde{p}_i$, the gradient is close to 0, while it is larger for those examples, where the prediction deviates more. Part 2 is the partial derivative of the k-best probability (cf. (3.9)).

Algorithm 10 Evaluating the gradient of a query 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. We use the sigmoid function $\sigma(\cdot)$ to translate $a_n \in \mathbb{R}$ into a probability.

```
1: function GRADIENT(node n, target fact n_j)
2:     if n is the 1-terminal then return (1, 0)                ▷ Base Case
3:     if n is the 0-terminal then return (0, 0)                ▷ Base Case
4:     let h and l be the high and low children of n             ▷ Inductive Case
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) + σ(−a_n) · prob(l)
8:     grad ← σ(a_n) · grad(h) + 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_n)σ(−a_n)
11:    end if
12:    return (prob, grad)
13: end function
```

It is computationally infeasible to loop over all subprograms $L \subseteq L^T$ when computing the gradient as there are exponentially many L. As shown in Chapter 3, there is an efficient algorithm to compute $P_k^T(q_i)$ relying on BDDs [De Raedt et al., 2007]. In the following section we update this towards computing the gradient and integrate it with a standard gradient descent search to learn the parameters of a ProbLog program based on a training set.

6.3 Computing the Gradient By Means of BDDs

As discussed in Section 3.2, the algorithm of [De Raedt et al., 2007] (cf. also Algorithm 1) computes the success probability P_s^T and the k-best probability P_k^T for a query q efficiently by collecting all proofs and compactly representing them as a Binary Decision Diagram (BDD) [Bryant, 1986]. When the BDD is built, Algorithm 2 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. When the BDD represents the Boolean formula corresponding to the k best proofs the resulting probability is the k-best probability.

This algorithm can be extended to compute the gradient with respect to a particular a_j as shown in Algorithm 10. GRADIENT(n, n_j) calculates the gradient with respect to 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

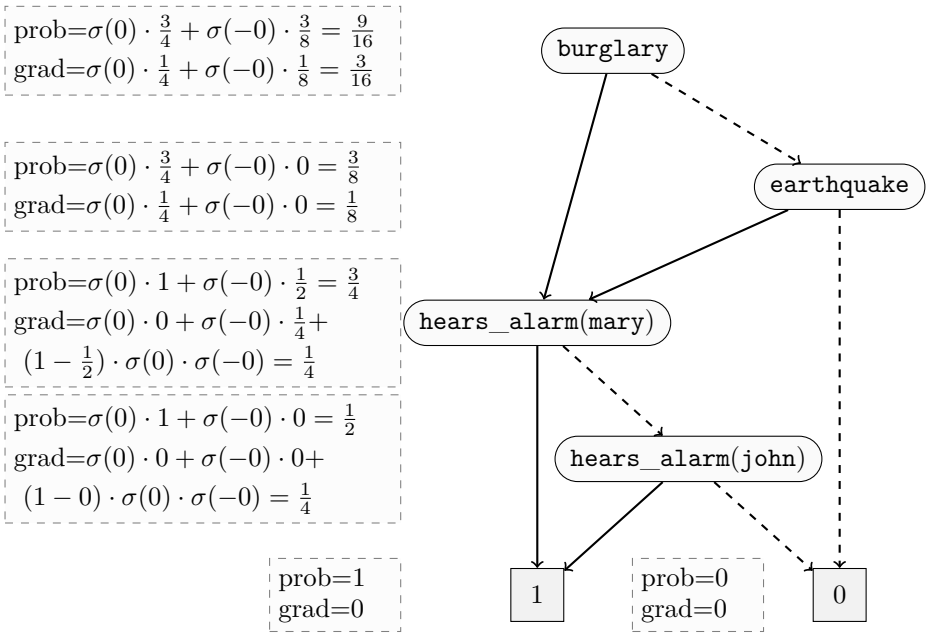


Figure 6.3: Intermediate results when calculating the gradient $\partial P_k^T(\texttt{calls(mary)} \vee \texttt{calls(john)})/\partial \mathtt{a_3}(\texttt{X})$ using Algorithm 10. The parameter a_3 governs the probability of the probabilistic fact `hears_alarm(X)`. The result is returned at the root node of the BDD.

theta-subsumption that expresses an *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 `hears_alarm(john)` $\subseteq_\theta$ `hears_alarm(john)`, `hears_alarm(john)` $\subseteq_\theta$ `hears_alarm(X)` but `hears_alarm(john)` $\not\subseteq_\theta$ `hears_alarm(mary)`. The correctness proof for Algorithm 10 can be found in Appendix D.

Example 6.5 (Gradient of a query probability)**.** *For the ease of calculation let us assume that the* Learn *algorithm (cf. Line 2 in Alg. 11) initialized all fact parameters of the* ALARM-4 *program from Example 6.1 with the value* 0*, that is,*

$$F = \{\sigma(0){::}\ \texttt{burglary},\ \sigma(0){::}\ \texttt{earthquake},\ \sigma(0){::}\ \texttt{hears\_alarm(X)}\}\ .$$

In order to calculate the gradient of the MSE (cf. (6.8)*) the algorithm evaluates the partial derivative for every probabilistic fact and every training example. Figure 6.3 illustrates how the partial derivate* $\partial P_k^T(\texttt{calls(mary)} \vee \texttt{calls(john)})/\partial \mathtt{a_3}(\texttt{X})$ *is obtained by running Algorithm 10. As one can see, the gradient is* $\frac{3}{16} > 0$*, which indicates that the probability of the query can be increased by increasing the value of* a_3*. This is plausible, as it is more likely that somebody calls about an alarm when the likelihood of hearing the alarm is increased.*

Algorithm 11 Gradient descent for ProbLog minimizing the MSE on the training data; T is a ProbLog program with unknown fact probabilities, $\{(q_j, \tilde{p}_j)|1 \leq j \leq M\}$ is the training set containing queries q_j and their target probabilities $\tilde{p}_j$, η is the learning rate, and $k \in \mathbb{N}$ is the number of proofs used to build the BDDs.

```
1: function LFE-ProbLog(T, {(q_j, p̃_j)|1 ≤ j ≤ M}, η, k)
2:     initialize all a_j randomly
3:     while not converged do
4:         Δa ← 0                                          ▷ Set gradient to 0
5:         for 1 ≤ i ≤ M do                                ▷ Loop over training examples
6:             BDD_i ←find k best proofs for q_i generate BDD ▷ See Equation 3.9
7:             y ← (2/M) · (P_k^T(q_i) − p̃_i)              ▷ cf. Part 1 in (6.8)
8:             for 1 ≤ j ≤ N do                            ▷ Loop over probabilistic facts
9:                 Δa_j ← Δa_j + y · ∂P_k^T(q_i)/∂a_j       ▷ Call Gradient(BDD_i, n_j)
10:            end for
11:        end for
12:        a ← a − η · Δa                                  ▷ Update model
13:    end while
14:    return {σ(a_j) :: c_j | c_j ∈ F} ∪ BK
15: end function
```

To obtain the final learning algorithm LFE-ProbLog, 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 11 shows the pseudocode of this gradient descent search.

6.4 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 6.1 performs poorly in terms of area under the precision-recall curve as we discovered in our experiments. To account for this we use a weighted MSE

$$\mathrm{MSE}_{\mathrm{cost}}(T) := \frac{1}{M} \sum_{i=1}^{M} \mathrm{cost}(\tilde{p}_i) \cdot \left(P_k^T(q_i) - \tilde{p}_i \right)^2 , \tag{6.9}$$

where $\text{cost}(1) := 1$ and $\text{cost}(0) := \alpha$. Thus negative training examples have only α times the influence on the MSE compared to positive training examples. When deriving the gradient (cf. Theorem 6.1) the $\text{cost}(\cdot)$ factor can be treated as constant resulting in

$$\frac{\partial \,\text{MSE}_{\text{cost}}(T)}{\partial p_j} = \frac{2}{M} \sum_{i=1}^{M} \text{cost}(\tilde{p}_i) \cdot \left(P_k^T(q_i) - \tilde{p}_i \right) \cdot \frac{\partial \; P_k^T(q_i)}{\partial p_j} \quad . \tag{6.10}$$

Introducing the $\text{cost}(\cdot)$ factor corresponds to an asymmetric loss matrix

$$\begin{bmatrix} P_k^T(q_i) \\ 1 - P_k^T(q_i) \end{bmatrix}^T \cdot \begin{bmatrix} 0 & \alpha \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} \tilde{p}_i \\ 1 - \tilde{p}_i \end{bmatrix} \quad . \tag{6.11}$$

This technique is often used for classification problems with non-uniform class distributions [Bishop, 2006]. Since the $\text{cost}(\cdot)$ factor introduces an additional parameter it is reasonable to ask what value to use for α. We suggest to set

$$\alpha = \frac{M_p}{M_n} \quad , \tag{6.12}$$

where M_P and M_n are the numbers of positive and negative training examples respectively. This particular value ensures 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. The experimental results indicate that this choice is justified (cf. Figure 6.9).

6.5 Experiments

In this section we empirically evaluate the proposed approach to parameter estimation. The experiments are designed to get insights 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 in every iteration)

C the interplay between learning from proofs and learning from entailment, and

D the performance of the approach compared to state-of-the-art statistical relational learning systems such as Alchemy.

Before presenting the results of our experimental investigation of the four questions outlined above, we will now describe the datasets, the evaluation criteria used as performance measure and the initialization of the parameters.

6.5.1 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. As working with the full Biomine graph would involve estimating several millions of parameters, we extracted two subgraphs, one around Alzheimer disease 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. Other relations stem from EntrezGene, String, UniProt, HomoloGene, Gene Ontology, and OMIM databases. Weights were assigned to edges as described in [Sevon et al., 2006]. In our experiments, 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 length not more than four, 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, the one around Asthma 127 nodes and 241 edges.

- 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 pages of the computer science department of the University of Washington[4] and the BibServ database[5]. 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 four universities: Cornell, Texas, Washington and Wisconsin. Next to the links, each page is annotated with the words occurring in the page's text. 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] `http://www.cs.washington.edu`
[5] `http://www.bibserv.org`

6.5.2 Evaluation Metrics

We use the following metrics to assess the results

- The mean squared error (MSE), cf. Equation (6.1), 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 zero they differ. We report $\sqrt{\text{MSE}_{\text{Test}}}$, 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].

6.5.3 Methodology

Before the gradient descent algorithm can be used, all the fact probabilities need to be initialized with some values. In our experiments, we sampled The initial fact probabilities 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 that yielded probability values in the interval $[0.43, 0.57]$. For the WebKB dataset we sampled the initial fact probabilities uniformly in $[0.03995, 0.04005]$. We experimentally found that this improved the convergence of the gradient search in terms of iterations compared to sampling from $[0.43, 0.57]$. However, it did not influence the predictive performance of the model. 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, this led to different numbers of iterations of gradient descent due to the maximum job duration on the PC cluster. On the UW-CSE dataset we ran 200 iterations of gradient descent and on the WebKB we ran 60 iterations of gradient descent.

6.5.4 Quality of Estimated Probabilities

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

Q1 Does LFE-ProbLog reduce the MSE both on training and test set?

Q2 Can LFE-ProbLog 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, both subgraphs of the Biomine graph. We sampled 500 random node pairs (a, b) in these graphs, estimating the query probability for path(a,b) using P_5^T, the probability of the five best proofs. We used the same approximation $k = 5$ in the LFE-ProbLog, where the set of proofs to build the BDD is determined anew in every iteration, as stated in Algorithm 11. 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 of gradient descent takes. In the same amount of time, LFE-ProbLog therefore performs less iterations when using more training examples.

The right column of Figure 6.4 shows the change of $\sqrt{\text{MSE}_{\text{Test}}}$ during learning. LFE-ProbLog 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 6.5. 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.

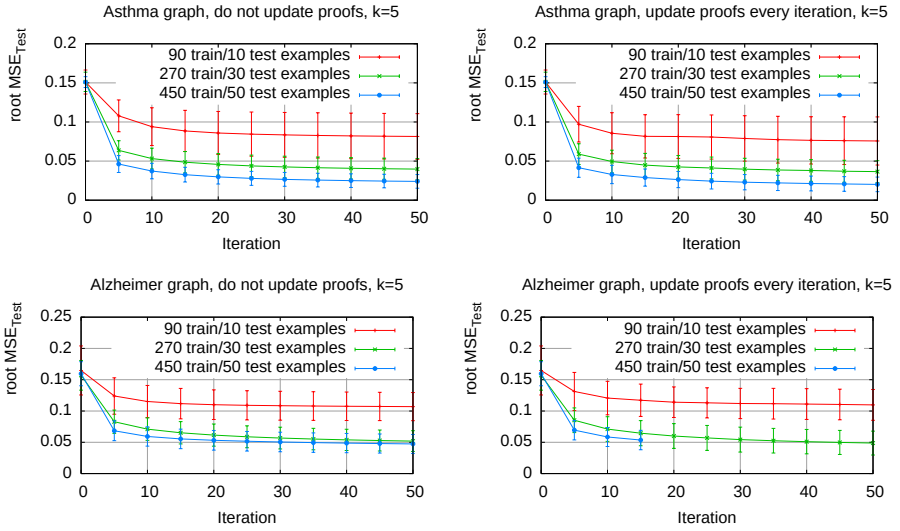


Figure 6.4: $\sqrt{\text{MSE}_{\text{Test}}}$ for Asthma and Alzheimer using the 5 best proofs ($k = 5$); when the BDDs and proofs are not updated (left column); when they are updated every iteration (right column) (**Q2** and **Q3**)

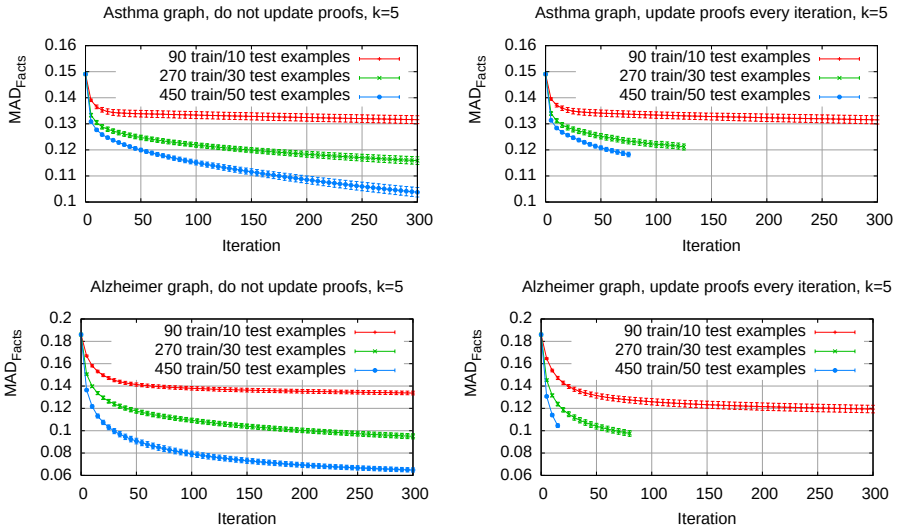


Figure 6.5: MAD_{facts} for Asthma and Alzheimer using the 5 best proofs ($k = 5$); when the BDDs and proofs are not updated (left column); when they are updated every iteration (right column) (**Q2** and **Q3**)

6.5.5 Influence of Approximations

LFE-ProbLog relies on several compute-intensive operations. First, recomputing the proofs and constructing the associated BDDs for each query in each iteration is expensive (see Line 6 in Algorithm 11). Conversely, 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 insights into the influence of such approximations, we set up experiments to answer the following questions:

Q3 Can we obtain good results even though we do not 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 experiments as before, but now without updating the set of proofs used for constructing the BDDs. The change of $\sqrt{\text{MSE}_{\text{Test}}}$ as well as of MAD_{Facts} are plotted in the left column of Figures 6.4 and 6.5 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 that had to be done only once in the current experiments. Retaining the initially-created BDDs gave a speedup of 10 for the Alzheimer graph. In this 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. When compared against runtime, though, this is clearly not the case. These results indicate that BDDs can safely be kept fixed during learning in the Biomine domain, which answers **Q3**. However, this result might be due to characteristics of the dataset and the theory, e.g., the amount of "overlap" between proofs and the amount of probability mass in shorter proofs. For the general case we suggest to evaluate the influence of keeping the BDDs fixed on a hold-out dataset prior to training the final model.

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 6.6 shows the results for this experiment after 50 iterations of gradient descent. The average absolute error per fact (MAD_{facts}) decreases slightly with higher k. The difference

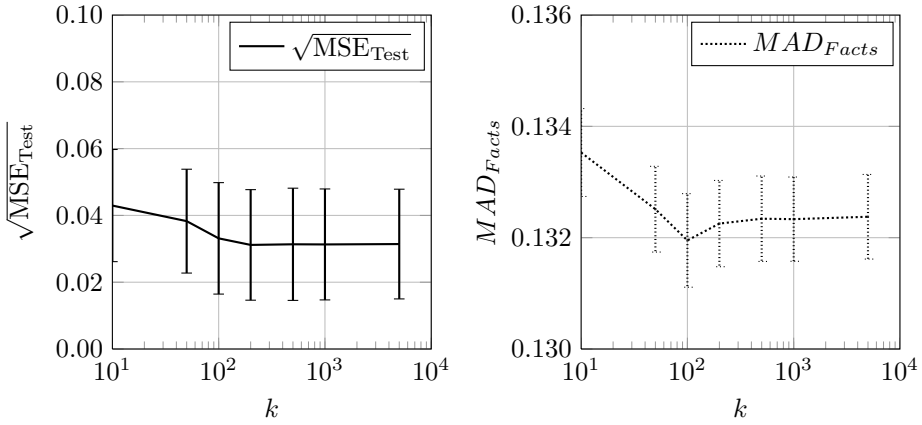


Figure 6.6: MAD_{facts} and $\sqrt{\text{MSE}_{\text{Test}}}$ after 50 iterations of LFE-ProbLog for different k (number of best proofs used) on the Asthma graph where training examples carry P_∞ probabilities (**Q4**)

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, $\alpha = 0.05$) when comparing the values for $k = 10$ and $k = 200$. Increasing k beyond 200 proofs has no significant influence the MSE. 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 hence answer **Q4**. For the general case we suggest to find a good value for k on a hold-out dataset prior to training the final model.

6.5.6 Learning From Entailment And From Proofs

LFE-ProbLog is able to simultaneously learn from proofs and queries as training examples. Hence we are interested in the following question:

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^T for path(a,b), the probability of the best path between a and b from the Asthma 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 6.7 shows the average value of MAD_{facts} and $\sqrt{\text{MSE}_{\text{Test}}}$ after 40 iterations of gradient descent. The graphs confirm that both error measures decrease as the fraction of proofs in the training data is increased. Figure 6.8 shows the learning curve for MAD_{facts} in the same experiment and indicates that the convergence is faster when the fraction of proofs in the training data is increased. These results confirm that proofs do carry more information and answer **Q5**.

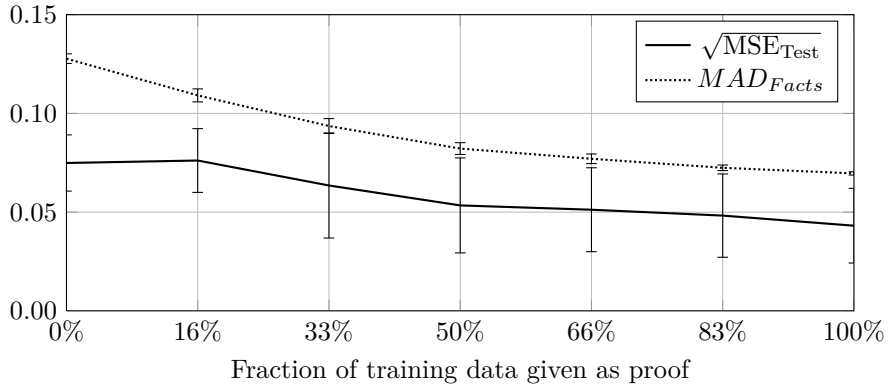


Figure 6.7: MAD_{facts} and $\sqrt{\text{MSE}_{\text{Test}}}$ after 40 iterations of gradient descent using LFE-ProbLog on the Asthma graph for different fractions of the training data given as proofs (**Q5**).

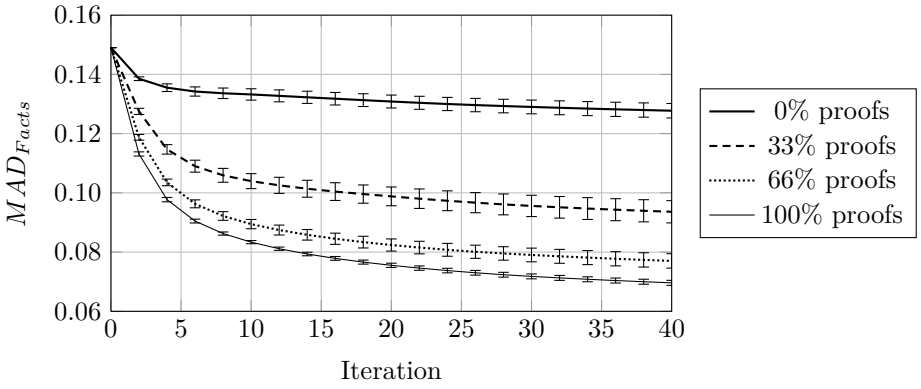


Figure 6.8: Learning curve for MAD_{Facts} on the Asthma graph for different fractions of the training data given as proofs (**Q5**).

6.5.7 Comparison to State-Of-The-Art

So far the experiments were designed to study various aspects of the learning algorithm and the system itself, while the performance of the trained model was of secondary interest. This brings us on to the following question:

Q6 Does LFE-ProbLog perform as well as Markov Logic Networks in terms of prediction?

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 on this dataset, we translated the Markov logic network (MLN) used by Richardson and Domingos into ProbLog clauses using Algorithm 16 (see Appendix C). The ProbLog theory contained 113 probabilistic facts whereas the original MLN contained 94 clauses. The translation yielded a ProbLog program with unknown fact probabilities, which were then learned using LFE-ProbLog and *leave-one-department-out* cross validation. Given n person constants in a sub-database, we generated n^2 training examples, one for every possible grounding of `advisedBy(X,Y)`. The examples got the probability 1 if the particular `advisedBy` tuple was contained in the sub database, otherwise 0. The resulting training sets are imbalanced. Averaged over all departments, the ratio M_P/M_n of positive to negative examples was ≈ 0.00756. Test runs with the standard MSE showed poor performance, namely all fact probabilities were set to values close to zero. Hence we minimized MSE_{cost} with $\alpha = 0.00756$ to account for the imbalance (cf. Section 6.4).

The clauses generated by converting the MLN contain cycles that are incompatible with 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, that is, all proofs were 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 $P_s^T(\texttt{advisedBy}(\texttt{X},\texttt{Y}))$ for every possible grounding of `advisedBy(X,Y)`, using all person constants from the test set. We then chose a threshold τ and classified those atoms as true that had a success probability of at least τ. Others were classified as negative. We obtained the precision-recall curve by varying τ from 0 to 1.

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 in turn made learning and inference more difficult. Table 6.1 shows the results of this experiment. We used the same setup

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

Table 6.1: Area under the precision-recall curve when predicting `advisedBy(X, Y)` when all other predicates are known (All Info) and when `student(X)` and `professor(X)` are unknown (Partial Info) (**Q6**).

of Richardson and Domingos and computed the AUC and the standard deviation using their evaluation scripts. In their experiments MLN(KB) and MLN(KB+CL) performed best. We extracted the graphs for those two systems from their plots and included them in our plots (Figure 6.10-6.15). 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 (**Q6**).

To assess the influence of the α parameter in MSE_{cost} on the outcome, we repeated the experiment with different values (cf. Figure 6.9). For $\alpha = 1$ we get the standard MSE that performs worst. For $\alpha = M_P/M_N$ (cf. Equation 6.12) 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 *All Info* case and with a rather good value in the *Partial Info* case. This result indicates that our choice of α is justified. The results also suggest to use a hold-out dataset to tune α for increasing the performance.

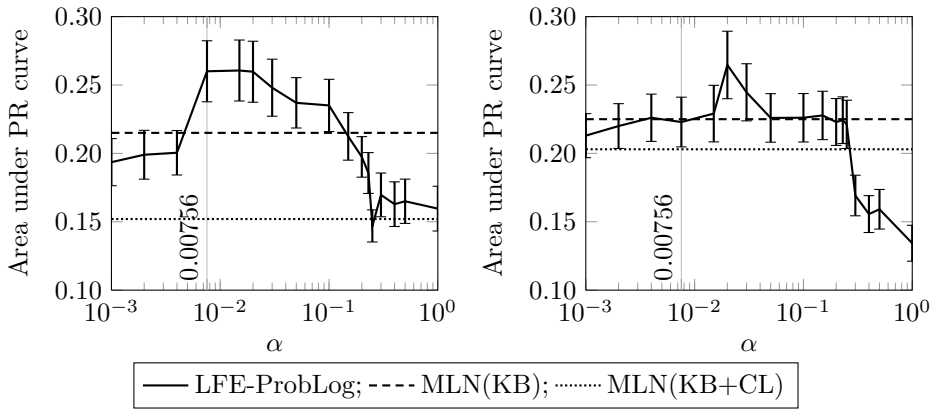


Figure 6.9: The area under the PR curve for different α values when everything is available (All Info, left graph) and when `student(X)` and `professor(X)` are unknown (Partial Info, right graph). The horizontal lines indicate the results obtained with MLNs reported in [Richardson and Domingos, 2006] (**Q6**).

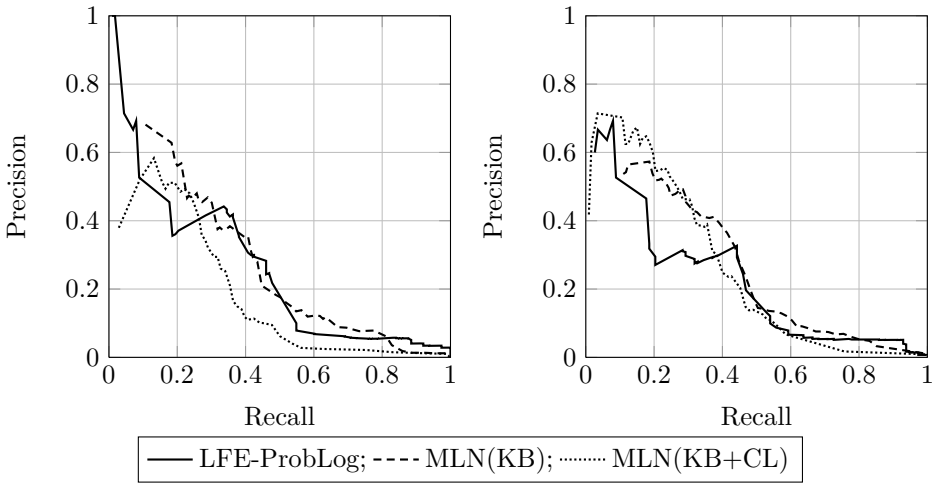


Figure 6.10: Precision and recall for all areas: All Info (left) and Partial Info (right) (**Q6**).

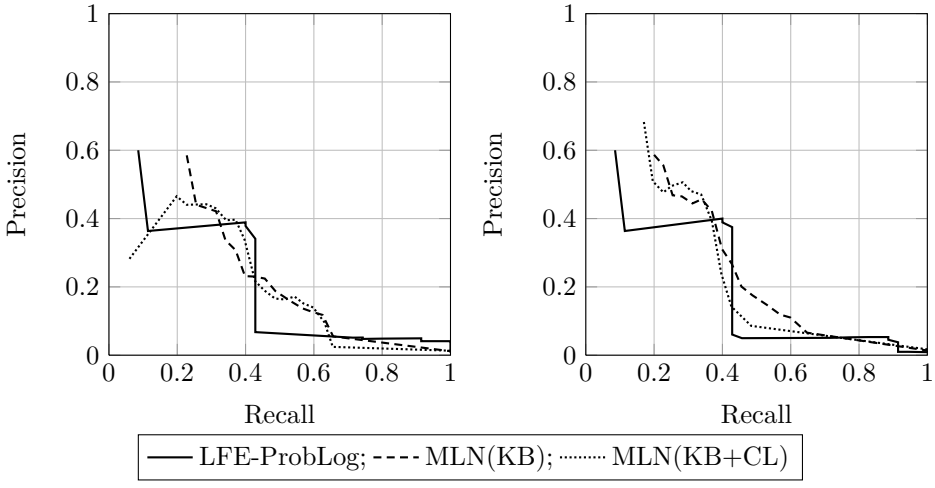


Figure 6.11: Precision and recall for the AI area: All Info (left) and Partial Info (right) (**Q6**).

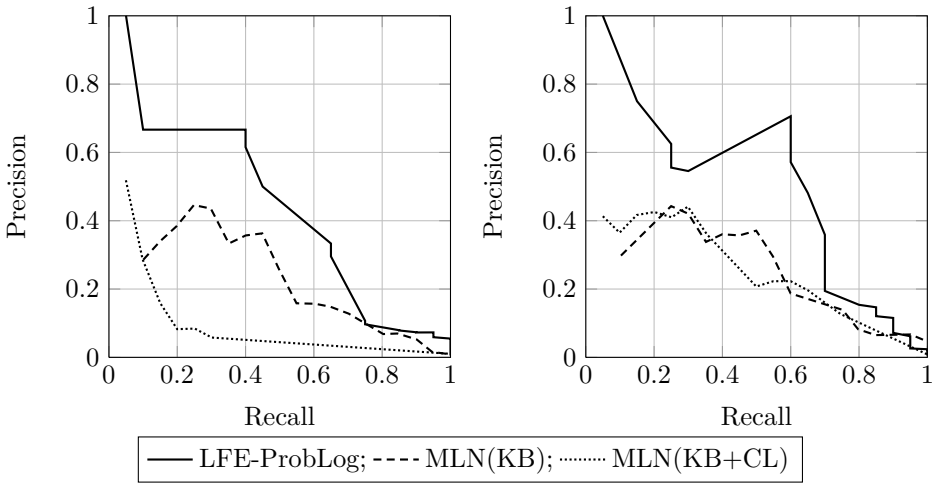


Figure 6.12: Precision and recall for the Graphics area: All Info (left) and Partial Info (right) (**Q6**).

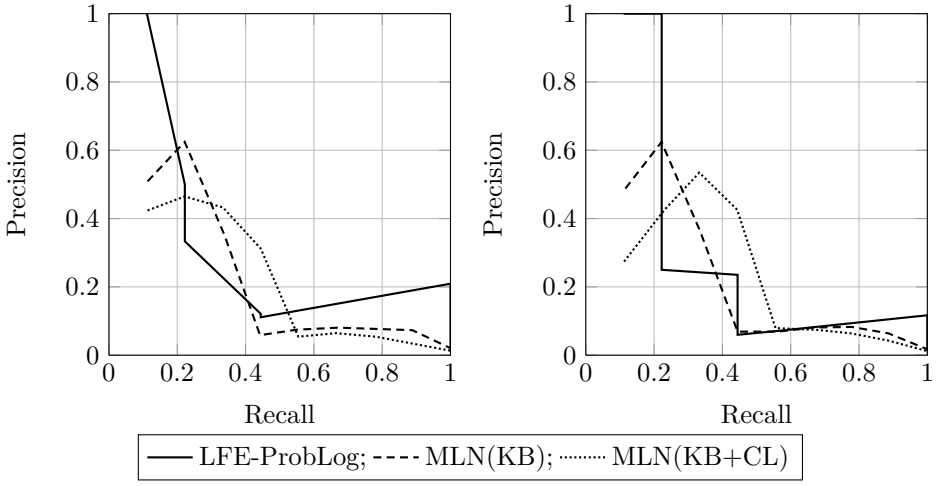


Figure 6.13: Precision and recall for the Languages area: All Info (left) and Partial Info (right) (**Q6**).

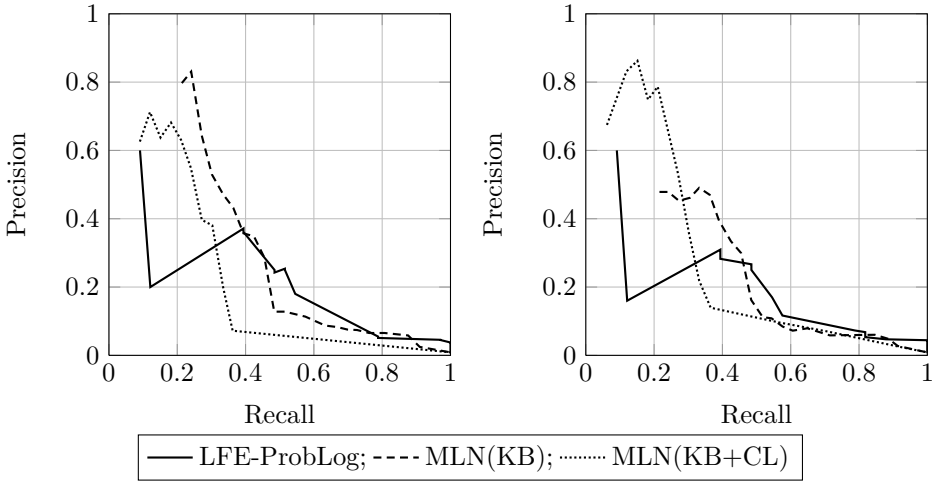


Figure 6.14: Precision and recall for the Systems area: All Info (left) and Partial Info (right) (**Q6**).

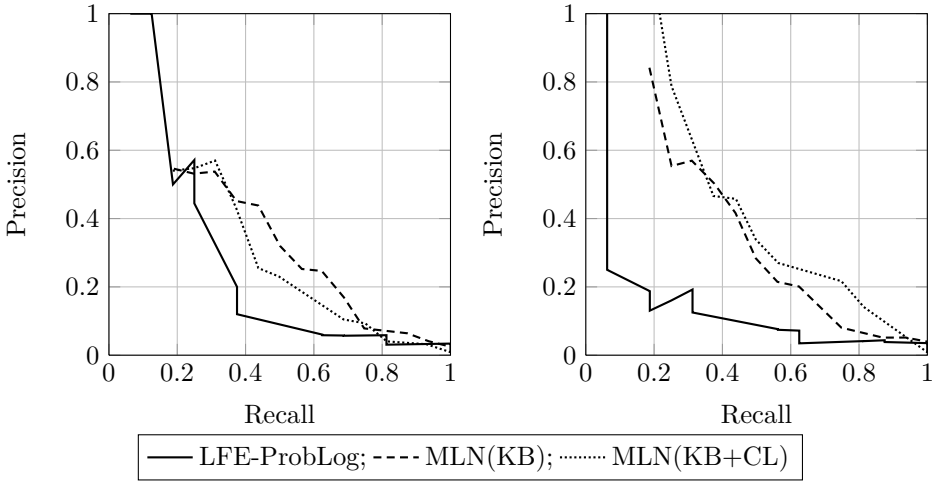


Figure 6.15: Precision and recall for the Theory area: All Info (left) and Partial Info (right) (**Q6**).

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]. In comparison to their model we ignore words that are absent on a page, while they explicitly take them into account. Our model consists 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).

$$p_{Word,\,Class} :: \texttt{word\_class}(Word,Class).$$

$$\texttt{class}(\texttt{Page},\texttt{C},\texttt{Depth}) :- \texttt{word\_class}(\texttt{W},\texttt{C}),\ \texttt{has\_word}(\texttt{Page},\texttt{W}).$$

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 `link_class`(`P1`, `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 `Depth` $= 1$ that restricts the search to the direct neighborhood of each page.

$$p_{Class1,\,Class2} ::\texttt{link\_class}(\texttt{P1},\texttt{P2},Class1,Class2).$$

$$\begin{aligned}\texttt{class}(\texttt{Page},\texttt{C},\texttt{Depth}) :-\ & \texttt{Depth} > 0,\\ & \texttt{Depth2 is Depth} - 1,\\ & \texttt{links\_to}(\texttt{OtherPage},\texttt{Page}),\\ & \texttt{class}(\texttt{OtherPage},\texttt{COther},\texttt{Depth2}),\\ & \texttt{link\_class}(\texttt{OtherPage},\texttt{Page},\texttt{COther},\texttt{C}).\end{aligned}$$

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

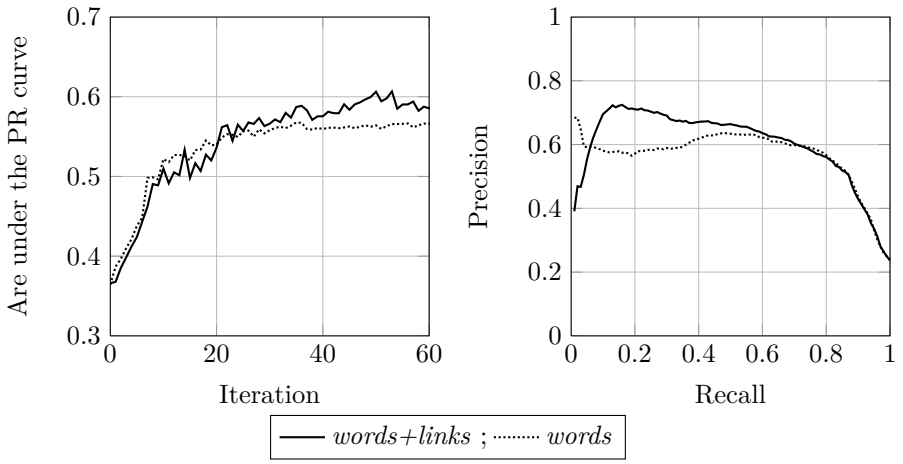


Figure 6.16: Area under the PR curve (AUC) after each learning step for WebKB (left), PR curve after 50 iterations of gradient descent (right) (**Q6**)

the first part of our model that considers the words appearing on a page. In the second run we used the full model that considers both words and links. We refer to the runs as *words* and *words+links* respectively. The results are shown in Figure 6.16. As expected, the *words+links* model outperforms the version restricted to words. It reaches its maximum AUC 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). Lowd and Domingos were able to improve the AUC up to ≈ 0.73 using second-order gradient techniques such as scaled conjugated gradients. These results answer **Q6** and show an interesting direction for future work: applying second order gradient techniques for LFE-ProbLog.

6.6 Related Work

Probabilistic relational models (PRMs) [Friedman et al., 1999b] and Bayesian logic programs (BLPs) [Kersting and De Raedt, 2008] are relational extensions of Bayesian networks using entity relationship models or logic programming respectively. Similarly 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 large. 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 do not need to be equal 1. These difficulties explain, partially, why so far only few learning techniques for probabilistic databases have been developed.

The ALLPAD system [Riguzzi, 2008] for learning ground LPADs is related to the work presented here since it also uses training examples with associated probabilities. However, the training examples in this setting are interpretations. 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. 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 little 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 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 randomly labeled with "0" or "1" according to the observed probabilities, whereas we use the observed probabilities directly. Finally, there are also differences in the algorithmic approach. Our learning algorithm follows a principled gradient approach, while 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.

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 [1997] structural EM learning for Bayesian networks, as it also reuses structures computed for similar problems. However, in contrast to structural EM, we do not evaluate the changes and use the old structure as an approximation of the new one.

Finally, the new setting and algorithm compromise a natural and useful addition to the existing learning algorithms for ProbLog. It is most closely related to the

theory compression setting of De Raedt et al. [2008b]. The task there 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 while the structure is fixed.

6.7 Conclusions And Future Work

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, though interpretations are typically too large to be used as training examples for parameter learning.

The LFE-ProbLog algorithm relies on gradient descent to find optimal parameters. While the approach works well in the domains we used in our experiments, there might be cases where the convergence speed is too slow. To account for that, we implemented a line search algorithm that tries to scale the gradient in order to find an optimal learning rate. This computation is almost as costly as computing the gradient itself, as it involves repeated evaluations of the BDDs. A more promising approach to increase the convergence speed would be the use of second-order gradient methods. They have been proven to be very efficient for Markov Logic Networks [Lowd and Domingos, 2007] and it is straightforward to adopt the gradient computation towards a second order gradient by using existing libraries like libLBFGs [Okazaki, 2007].

Another interesting extension is the combination with Hybrid ProbLog in order to learn the parameters of distributions governing the continuous facts. The first step, that is, deriving the gradient of the success probability with respect to the parameters has already been made (cf. Appendix E). A proof-of-concept implementation showed the feasibility of this approach.

Chapter 7

Learning from Interpretations*

Statistical relational learning [Getoor and Taskar, 2007] and probabilistic logic learning [De Raedt et al., 2008a; De Raedt, 2008] have contributed various representations and learning schemes. Popular approaches include BLPs [Kersting and De Raedt, 2007], ICL [Poole, 2008], Markov Logic [Richardson and Domingos, 2006], PRISM [Sato and Kameya, 2001], PRMs [Getoor et al., 2001] and ProbLog [De Raedt et al., 2007; Gutmann et al., 2008]. These approaches differ not only in the underlying representations but also in the learning settings they employ.

For learning knowledge-based model construction approaches (KBMC), such as Markov Logic, PRMs and BLPs, one usually employs relational state descriptions as training examples. This setting is also known as *learning from interpretations*. For training probabilistic programming languages one typically uses *learning from entailment* [De Raedt and Kersting, 2004; De Raedt et al., 2008a]. PRISM and ProbLog, for instance, are probabilistic logic programming languages that are based on Sato 's [1995] distribution semantics. They use training examples in the form of labeled facts, where labels are either the truth values of these facts or target probabilities.

In the learning from entailment setting, one usually starts from observations for a single target predicate. In the learning from interpretations setting, however, the observations specify the value for some of the random variables in a state description. Probabilistic grammars and graphical models are illustrative examples for each setting. Probabilistic grammars are trained on training examples in the form of sentences. Each training example states whether a particular sentence was derived or not, but it does not explain *how* it was derived. In contrast, Bayesian

*This chapter presents joint work with Ingo Thon and Luc De Raedt published in [Gutmann et al., 2010c, 2011a]

networks are typically trained on partial or complete state descriptions, which specify the value for some random variables in the network. This also implies that training examples for Bayesian networks can contain much more information. These differences in learning settings also explain why the KBMC and PLP approaches have been applied to different kinds of data sets and applications. Entity resolution and link prediction are examples of domains where KBMC has been successfully applied (cf. [Singla and Domingos, 2006]). This chapter aims at bridging the gap between these two types of approaches to learning. We study how the parameters of ProbLog programs can be learned from partial interpretations.

This chapter has two core contributions: (1) a parameter estimation algorithm for learning from partial interpretations for ProbLog and (2) an optimization that identifies independent parts of the theory such that learning in real-world domains becomes feasible.

The chapter is organized as follows: Section 7.1 formalizes the problem of learning the parameters of ProbLog programs from interpretations. Section 7.2 introduces LFI-ProbLog for finding the maximum likelihood parameters. We report on experimental results in Section 7.3. Before concluding, we discuss related work in Section 7.4.

7.1 Learning From Interpretations

Learning from (possibly partial) interpretations is a common setting in statistical relational learning that has not yet been studied in its full generality for probabilistic programming languages. The semantics of a ProbLog program does not encode a generative process at the level of individual predicates. Hence one cannot learn in a generative setting from individual queries. Thus we needed to formulate the learning task for LFE-ProbLog (cf. Chapter 6) as a regression problem. At the level of interpretations, however, ProbLog does encode a generative model as we argued in Chapter 3. It follows from the fact that each total choice generates a unique possible world through its least Herbrand model. Therefore, it is much more natural to learn from interpretations in ProbLog; this is akin to typical KBMC approaches. In a generative setting, one normally is interested in the maximum likelihood parameters given the training data. This can be formalized as follows.

Definition 7.1 (Max-Likelihood Parameter Estimation)**.** ***Given*** *a ProbLog program $T = F \cup BK$ where the probabilistic facts F have* unknown *parameters $\mathbf{p} = \langle p_1, \ldots, p_N \rangle$ and a set of partial or complete interpretations $D = \{I_1, \ldots, I_M\}$ (the training examples).* ***Find*** *probabilities $\widehat{\mathbf{p}} = \langle \widehat{p_1}, \ldots, \widehat{p_N} \rangle$ such that*

$$\widehat{\mathbf{p}} = \arg\max_{\mathbf{p}} P(D|T) = \arg\max_{\mathbf{p}} \prod_{m=1}^{M} P_w^T(I_m) \ .$$

Thus we are given a ProbLog program and a set of partial interpretations and the goal is to find the maximum likelihood parameters. We use the previously introduced ALARM-2 program to illustrate the concepts and algorithms in this chapter:

$$F = \{0.1{::}\ \texttt{burglary},\ 0.2{::}\ \texttt{earthquake},\ 0.8{::}\ \texttt{hears\_alarm(X)}\}$$

$$\begin{aligned} BK = \{&\texttt{person(mary)},\\ &\texttt{person(john)},\\ &\texttt{alarm} :- \texttt{burglary}\\ &\texttt{alarm} :- \texttt{earthquake}\\ &\texttt{calls(X)} :- \texttt{person(X)},\ \texttt{alarm},\ \texttt{hears\_alarm(X)}\} \end{aligned}$$

A partial interpretation I specifies the truth value for some but not necessarily for all atoms. We represent partial interpretations as $I = (I^+, I^-)$, where I^+ contains all *true* atoms and I^- all *false* atoms and $I^+ \cap I^- = \varnothing$. The probability of a partial interpretation is the sum of the probabilities of all possible worlds consistent with the known atoms. This is the success probability of the query $(\bigwedge_{a_j \in I^+} a_j) \wedge (\bigwedge_{a_j \in I^-} \neg a_j)$. In the ALARM-2 program the probability of the following partial interpretation

$$\begin{aligned} I^+ = \{&\texttt{person(mary)}, \texttt{person(john)}, \texttt{burglary}, \texttt{alarm},\\ &\texttt{hears\_alarm(john)}, \texttt{calls(john)}\} \end{aligned}$$

$$I^- = \{\texttt{calls(mary)}, \texttt{hears\_alarm(mary)}\}$$

is $P_w^T((I^+, I^-)) = 0.1 \times 0.8 \times (1 - 0.8) \times \big(0.2 + (1 - 0.2)\big)$ as there are two total choices that result in this partial interpretation.

For instructive reason we first discuss the fully-observable case in the following section. For complete interpretations, where everything is observable, one can compute $\widehat{\mathbf{p}}$ by counting. In the more complex case of partial interpretations, one has to use an approach that is capable of handling partial observability (cf. Section 7.1.2).

7.1.1 Full Observability

In the fully-observable case the maximum likelihood estimators $\widehat{p_n}$ for the probabilistic facts $p_n :: f_n$ can be obtained by counting the number of true ground

instances in every interpretation, that is,

$$\widehat{p_n} = \frac{1}{Z_n} \sum_{m=1}^{M} \sum_{k=1}^{K_n^m} \delta_{n,k}^m \qquad \text{where} \qquad \delta_{n,k}^m := \begin{cases} 1 & \text{if } f_n \theta_{n,k}^m \in I_m \\ 0 & \text{else} \end{cases} \tag{7.1}$$

and $\theta_{n,k}^m$ is the k-th possible ground substitution for the fact f_n in the interpretation I_m and K_n^m is the number of such substitutions. The sum is normalized by $Z_n = \sum_{m=1}^{M} K_n^m$, the total number of ground instances of the fact f_n in all training examples. If Z_n is zero, i.e., no ground instance of f_n is used, $\widehat{p_n}$ is undefined and one must not update the fact probability p_n.

Before moving on to the partially-observable case, let us consider the issue of determining the possible substitutions $\theta_{n,k}^m$ for a fact $p_n :: f_n$ and an interpretation I_m. To resolve this, we assume that the facts f_n are typed and that each interpretation I_m contains an explicit definition of the different types in the form of fully-observable unary predicates.[1] In the ALARM-2 example, the predicate `person/1` can be regarded as the type of the (first) argument of `hears_alarm(X)` and `calls(X)`. This predicate may have different ground atoms in each interpretation. One person, i.e., can have `john` and `mary` as neighbors, another one `ann`, `bob` and `eve`.

7.1.2 Partial Observability

In many applications the training examples are partially observed. In the ALARM-2 example, we may receive a phone call but we may not know whether an earthquake has in fact occurred. In the partially-observable case – similar to Bayesian networks – a closed-form solution of the maximum likelihood parameters is infeasible. Instead, one has to replace in (7.1) the term $\delta_{n,k}^m$ by $E_T[\delta_{n,k}^m | I_m]$, that is, the conditional expectation under the current model T given the partial interpretation I_m,

$$\widetilde{p_n} := \frac{1}{Z_n} \sum_{m=1}^{M} \sum_{k=1}^{K_n^m} E_T[\delta_{n,k}^m | I_m] \ . \tag{7.2}$$

The $\widetilde{p_n}$ values can then be used by the EM algorithm described in the following section to compute the maximum likelihood estimates $\widehat{p_n}$. Before we describe this algorithm, we highlight a crucial property of (7.2) that allows us to ignore irrelevant ground facts in order to speed up the computation. Consider the partial interpretation $I^+ = \{\texttt{person(mary)}, \texttt{person(john)}, \texttt{alarm}\}$ and $I^- = \varnothing$ for the ALARM-2 example. Only the atoms in {`burglary`, `earthquake`, `hears_alarm(john)`, `hears_alarm(mary)`} $\cup\, I^+$ are relevant for calculating the

[1] This assumption can be relaxed if the types are computable from the ProbLog program and the current interpretation.

marginal probability (expected counts) of all probabilistic facts. This is due to the fact that the remaining atoms {`calls(john)`, `calls(mary)`} are not used in any proof for the facts observed in the interpretations. Therefore, they do not influence the probability of the partial interpretation. Such atoms play a role similar to that of barren nodes in Bayesian networks [Jensen, 2001]. This motivates the following definition.

Definition 7.2 (Dependency Set of Atom)**.** *Let $T = F \cup BK$ be a ProbLog theory and x a ground atom then the* dependency set of x *is defined as*

$$\mathrm{dep}_T(x) := \{f \text{ ground fact} \mid \text{a ground SLD-proof in } T \text{ for } x \text{ contains } f\} \ .$$

Thus, $\mathrm{dep}_T(x)$ contains all ground atoms that appear in any possible proof of the atom x. This can be generalized to partial interpretations:

Definition 7.3 (Dependency Set of Interpretation)**.** *Let $T = F \cup BK$ be a ProbLog theory and $I = (I^+, I^-)$ a partial interpretation then the* dependency set of the partial interpretation I *is defined as*

$$\mathrm{dep}_T(I) := \left(\bigcup_{x \in I^+} \mathrm{dep}_T(x) \right) \cup \left(\bigcup_{x \in I^-} \mathrm{dep}_T(x) \right) \ .$$

Our goal is to restrict the probability calculation to the dependent atoms only. Before doing so, we first need the notion of the restricted ProbLog theory.

Definition 7.4 (Interpretation-Restricted ProbLog theory)**.** *Let $T = F \cup BK$ be a ProbLog theory and $I = (I^+, I^-)$ a partial interpretation. Then we define the* interpretation-restricted *ProbLog theory $T^r(I)$ as follows*

$$T^r(I) := F^r(I) \cup BK^r(I) \ .$$

The set of facts $F^r(I)$ is defined as $\{p :: f\theta \mid p :: f \in F \wedge f\theta \in dep_T(I)\}$ and $BK^r(I)$ is obtained by computing all ground instances of clauses in BK in which all atoms appear in $\mathrm{dep}_T(I)$*.*

In other words, the interpretation-restricted theory $T^r(I)$ *grounds* the part of the original theory T that is relevant for generating I. The rest, that is, ground atoms that do not appear in any proof for any atom in I, is ignored. In turn, this allows one to apply the learning algorithm on programs that otherwise would not fit in memory. More importantly, $T^r(I)$ is always finite as we assume T having the *finite support property.*

Example 7.1 (Interpretation-Restricted ProbLog theory)**.** *For the partial interpretation $I = (\{$`burglary`, `alarm`$\}, \varnothing)$, for instance, $F^r(I)$ is* {`0.1 :: burglary`, `0.2 :: earthquake`} *and $BK^r(I)$ is* {`alarm :- burglary`, `alarm :- earthquake`}*.*

It can be shown that the conditional probability of ground instances of the probabilistic fact f_n given I calculated in the theory T is equivalent to the probability calculated in $T^r(I)$

$$E_T[\delta^m_{n,k}|I_m] = \begin{cases} E_{T^r(I_m)}[\delta^m_{n,k}|I_m] & \text{if } f_n \in \text{dep}_T(I_m) \\ p_n & \text{otherwise} \end{cases} . \tag{7.3}$$

This property guarantees that the subsequent steps of the learning algorithm yield the correct result when applied on the interpretation-restricted theory. The proof for (7.3) can be found in Appendix F.

Non-ground probabilistic facts such as 0.8 :: `hears_alarm(X)` have a parameter-tying between all ground instances. For instance, both `hears_alarm(john)` and `hears_alarm(mary)` are *true* with probability 0.8, independently of each other. When learning the probabilities, this parameter-tying has to be taken into account after grounding the theory. LFI-Problog enforces parameter-tying by aggregating the $\delta^m_{n,k}$ counts per non-ground fact (cf. Eq. 7.1), where k identifies the ground instances of the probabilistic fact f_n.

7.2 The LFI-ProbLog Algorithm

We now develop the LFI-ProbLog algorithm for finding the maximum likelihood parameters $\widehat{\mathbf{p}}$ defined in (7.1). One of the key ideas is the transformation of the program T into an equivalent Boolean formula by Clark's completion. In order to yield correct results, this step requires T to be *acyclic*. A definite logic program is *acyclic* if there exists a mapping function from atoms to natural numbers, such that for each clause the mapping of the head is larger than the mapping of all body atoms (cf. [Apt and Bezem, 1991]).

Example 7.2. *The following program is acyclic*

```
day(0).    day(s(X)) :- day(X).
```

and a possible mapping function is $f($`day(0)`$) := 0$ *and* $f($`day(s(X))`$) := 1 + f(f($`day(s(X))`$))$. *The program*

```
q :- p.    p :- q.
```

is cyclic as there exists no mapping function.

Algorithm 12 shows the main loop of LFI-ProbLog. In Line 3, each training example I_m, that is, each partial interpretation is translated into a Binary Decision Diagram and atoms with known truth value using the GenerateBDD(T, I_m) function.

Algorithm 12 The main loop of LFI-ProbLog. Each training example $(I_m^+, I_m^-) \in I$ is represented as a BDD together with the set of ground probabilistic facts that are known to be true or false. After the initialization of the BDDs, the algorithm follows an EM update scheme, that is, using the current model to complete the data and then estimating the new model parameters from this until convergence.

```
1: function LFI-ProbLog(T = F ∪ BK, I)
2:     for 1 ≤ m ≤ M do                                  ▷ Loop over training examples
3:         (bbd_m, known_m^true, known_m^false) ←GenerateBDD(T, (I_m^+, I_m^-))
4:     end for
5:     initialize all fact probabilities f_n in F randomly
6:     while not converged do                                          ▷ EM algorithm
7:         for 1 ≤ m ≤ M do                              ▷ Loop over training examples
8:             α_m ←Alpha(bbd_m)                                          ▷ E Step
9:             β_m ←Beta(bbd_m)                                           ▷ E Step
10:        end for
11:        compute E_T[δ_{n,k}^m | I_m] for all m, k and n using Eq. 7.4   ▷ E Step
12:        for 1 ≤ n ≤ N do                              ▷ Loop over probabilistic facts
13:            p_n~ ← (1/Z_n) Σ_{m=1}^{M} Σ_{k=1}^{K_n^m} E_T[δ_{n,k}^m | I_m]   ▷ cf. Eq. 7.2 M Step
14:        end for
15:        Replace all p_n with p_n~                          ▷ Update model parameters
16:    end while
17:    return {p_n :: f_n | f_n ∈ F} ∪ BK
18: end function
```

This information is then used in Line 5-16 to estimate the fact probabilities using the expectation maximization (EM) algorithm (cf. [Dempster et al., 1977]). After randomly initializing the model parameters, the EM algorithm repeatedly performs the following two operations until the parameters have converged:

- *E Step*: Use the current model to determine the conditional distribution of the unobserved random variables.
- *M Step*: Use the observed random variables together with the distribution of the unobserved random variables to estimate the model parameters.

We now discuss the details of the auxiliary functions used by LFI-ProbLog. In the following section we describe the details of this translation employed by GenerateBDD while Section 7.2.2 explains the computation of the expected counts based on Alpha and Beta. Please note that we employ two optimizations to minimize the size of the resulting BDDs. Firstly, we use *unit propagation* to identify ground probabilistic facts that are certainly true or false and do not add them to the formulae represented as BDD. Secondly, we use an algorithm that

Algorithm 13 This algorithm translates a ProbLog theory T and a set of evidence atoms $I = (I^-, I^+)$ into a BDD. Unit propagation is used in the SIMPLIFY step to identify atoms with known truth values. Such atoms are not contained in the BDD but returned separately.

```
1: function GENERATEBDD(T, I)
2:     dep ← dep_T(I)                    ▷ Find the dependency set of I (cf. Definition 7.3)
3:     F ← ∅                             ▷ Set of Boolean formulae to encode T and I
4:     for a ∈ dep where a is not a probabilistic fact do
5:         (a ↔ body_a) ←CLARKSCOMPLETION(a, dep, T)
6:         F ← F ∪ {a ↔ body_a}
7:     end for
8:     (F, known^true, known^false) ←PROPAGATEEVIDENCEANDSIMPLIFY(F, I)
9:     bdd ←CONSTRUCTBDD(φ_1 ∧ φ_2 ∧ ... ∧ φ_k)      ▷ the φ_i ∈ F are formulae
10:    return (bdd, known^true, known^false)
11: end function
```

exploits independencies in the Boolean formulae to split the BDD into smaller parts. The details of this *splitting algorithm* are discussed in Section 7.2.3.

7.2.1 Computing the BDD For An Interpretation

In this section we describe how LFI-ProbLog translates a partial interpretation I together with a ProbLog theory T into a BDD. We shall do this by executing Algorithm 13 on the partial interpretation $I^+ = \{\texttt{alarm}\}$, $I^- = \{\texttt{calls(john)}\}$ and the ALARM-2 program.

1. The algorithm computes $\text{dep}_T(I)$ in Line 2 by executing a tabled meta-interpreter for all atoms in I and collecting all ground atoms in all proofs. The set $\text{dep}_T(I)$ contains all atoms that may have an influence on the probability of the partial interpretation I.

$$dep_T(\texttt{alarm}) = \{\texttt{alarm}, \texttt{earthquake}, \texttt{burglary}\}$$

$$dep_T(\texttt{calls(john)}) = \{\texttt{alarm}, \texttt{earthquake}, \texttt{burglary}, \texttt{person(john)}$$
$$\texttt{hears\_alarm(john)}, \texttt{calls(john)}\}$$

   Based on this, one can obtain the interpretation-restricted theory (cf. Definition 7.4) by generating all ground clauses from BK and probabilistic facts that can be constructed by replacing the atoms with matching ground instances from $dep_T(I) = dep_T(\texttt{alarm}) \cup dep_T(\texttt{calls(john)})$. In terms of

logic programming this step is called *grounding*. It yields the following theory:

$$
\begin{aligned}
F^r(I) = \{&0.1 :: \texttt{burglary},\\
&0.2 :: \texttt{earthquake},\\
&0.8 :: \texttt{hears\_alarm}(\texttt{john})\}\\
BK^r(I) = \{&\texttt{person}(\texttt{john}),\\
&\texttt{alarm} :\!- \texttt{burglary},\\
&\texttt{alarm} :\!- \texttt{earthquake},\\
&\texttt{calls}(\texttt{john}) :\!- \texttt{person}(\texttt{john}), \texttt{alarm},\\
&\qquad\qquad \texttt{hears\_alarm}(\texttt{john})\}
\end{aligned}
$$

The algorithm does not explicitly generate the interpretation-restricted theory. Instead, it performs the grounding simultaneously with Clark's completion in the next step.

2. The algorithm computes Clark's completion of $BK^r(I)$ in Line 5–7. This is a well-known transformation (cf. [Nilsson and Małuszyński, 1995]) that translates a logic program into a Boolean formula. This step requires the program to be *acyclic* in order to be sound. If the program is cyclic, it can be the case that the Boolean formula can be satisfied by a truth value assignment that is not a model of the original program.

   Clark's completion of $BK^r(I)$ is computed by replacing all clauses with the same head $h :\!- \mathit{body}_1, \ldots, h :\!- \mathit{body}_n$ by the corresponding formula $h \leftrightarrow \mathit{body}_1 \vee \ldots \vee \mathit{body}_n$. The result in our example is the conjunction of

$$
\begin{aligned}
\texttt{person}(\texttt{john}) &\leftrightarrow \texttt{true}\\
\texttt{alarm} &\leftrightarrow (\texttt{burglary} \vee \texttt{earthquake})\\
\texttt{calls}(\texttt{john}) &\leftrightarrow \texttt{person}(\texttt{john}) \wedge \texttt{alarm} \wedge \texttt{hears\_alarm}(\texttt{john})
\end{aligned}
$$

   Clark's completion allows one to propagate values from the head to the bodies and vice versa. It states that the head is true if *and only if* at least one of its bodies is true, which captures the least Herbrand model semantics when the logic program is *acyclic*.

   Note that we generate Clark's completion for the background knowledge BK but not for probabilistic facts, for instance, we do not generate $\texttt{burglary} \leftrightarrow \texttt{true}$. If we would do that, the following unit propagation algorithm could simplify the formula and remove all probabilistic facts. Doing so, we make the *open world assumption* for the probabilistic facts.

3. The function PROPAGATEEVIDENCEANDSIMPLIFY in Line 8 propagates the evidence into the formulae $\phi_i \in F$, that is, all atoms that occur in I^+ get replaced by `true` and all atoms that occur in I^- get replaced by `false`:

$$\texttt{person(john)} \leftrightarrow \texttt{true}$$

$$\texttt{true} \leftrightarrow (\texttt{burglary} \vee \texttt{earthquake})$$

$$\texttt{false} \leftrightarrow \texttt{person(john)} \wedge \texttt{true} \wedge \texttt{hears\_alarm(john)}$$

Then the function simplifies the Boolean formula by applying basic term-equivalent operations, for instance, removing `true` from conjunctions and De Morgan's laws. Moreover, the function employs *unit propagation* to replace atoms by `true` or `false` if their value is certain, i.e., the atom `person(john)` in the example above.

It is worth noting that the simplification step is not mandatory and one could stop after propagating the truth values from I. In addition, the simplification is expensive as it requires repeated operations on terms represented in the form of trees. Nonetheless, it is crucial for performance of BDD operations later on. By using unit propagation to identify atoms that are certainly true or false, we minimize the amount of variables the BDD has to represent. In turn, it can be build faster and less memory is required.

When no more simplification is possible the algorithm returns the conjunction of the formulae $\phi_i \in F$ as well as the ground probabilistic facts with known true value:

$$(\texttt{burglary} \vee \texttt{earthquake})$$

$$known^{true} = \varnothing,\ known^{false} = \{\texttt{hears\_alarm(john)}\}$$

While the resulting formula in this example contains only atoms that are also probabilistic facts, it can be the case that it also contains derived atoms from BK. In turn, such atoms will then be represented in the BDD and we shall refer to nodes that correspond to such atoms as *deterministic nodes* since they represent atoms in the deterministic background knowledge BK.

4. In Line 9 the algorithm constructs the BDD shown in Figure 7.1. This data structure represents the Boolean formula and is used by Algorithm 14 and 15 to compute the expected counts of the probabilistic facts.

In the following section, we show the computation of the expected counts $E_T[\delta^m_{n,k}|I_m]$ using the BDD and the $known^{true}$ and $known^{false}$ sets. Based on the counts, one can then compute the estimated fact probabilities $\widetilde{p_n}$ (cf. Eq. 7.2).

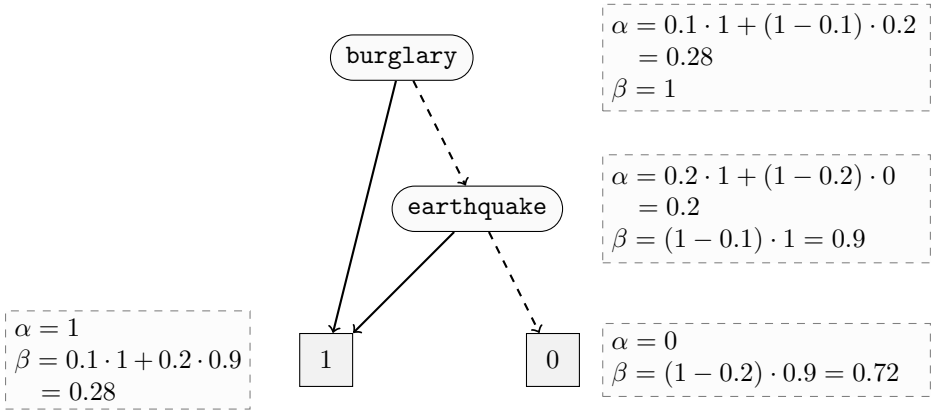


Figure 7.1: The BDD generated by LFI-ProbLog for the partial interpretation $I^+ = \{\texttt{alarm}\}$, $I^- = \{\texttt{calls(john)}\}$ of the ALARM-2 program. The α and β values are computed in Algorithm 14 and 15 and are used for calculating the expected counts of the probabilistic facts.

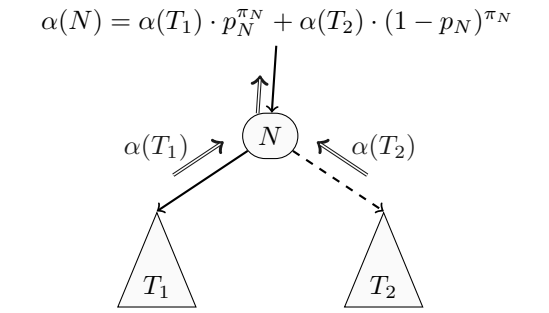
7.2.2 Calculating Expected Counts

We compute the expected counts $E_T[\delta^m_{n,k}|I_m]$ by a dynamic programming algorithm on BDD_m. The computation is similar to the forward-backward algorithm for Hidden Markov Models (cf. [Rabiner, 1989]) or Baker's [1979] inside-outside algorithm for probabilistic context-free grammars. Our algorithm computes the *upward probability* $\alpha(N)$ as well as the *downward probability* $\beta(N)$ for each node N in the BDD. These values are then used to compute the expected count of the kth ground instance of the probabilistic fact $p_n :: f_n$ given the interpretation I_m as

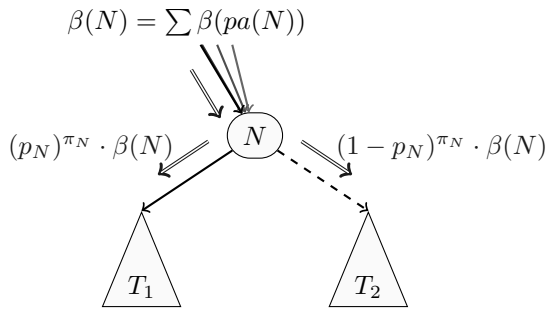
$$E_T[\delta^m_{n,k}|I_m] = \frac{1}{P(BDD)} \sum_{\substack{N \in BDD_m \\ N \text{ represents } f_n\theta^m_{n,k}}} \beta(N) \cdot p_n \cdot \alpha(h(N)) \ . \tag{7.4}$$

We have to take into account that we removed ground probabilistic facts due to unit propagation and they are not represented in the BDD. Thus if $f_n\theta^m_{n,k} \in known^{true}_m$ then $E_T[\delta^m_{n,k}|I_m] = 1$ and if $f_n\theta^m_{n,k} \in known^{false}_m$ then $E_T[\delta^m_{n,k}|I_m] = 0$. We also have to consider the probabilistic facts that are neither represented in the BDD nor in the *known* sets. Their expected count is the prior probability of the fact itself, that is, $E_T[\delta^m_{n,k}|I_m] = p_n$ (cf. Eq. 7.3). In the remainder of this section we discuss the details of the $\alpha(\cdot)$ and $\beta(\cdot)$ values and the algorithms for computing them.

The intuitive meaning of (7.4) is as follows. Each path from the root of the BDD to the 1-terminal corresponds to an assignment of values to the variables that satisfies the Boolean formula underlying the BDD. The probability that such a path passes



(a) Upward probability



(b) Downward probability

Figure 7.2: Information flow in the BDD when calculating $\alpha(\cdot)$ and $\beta(\cdot)$. Solid lines represent *high* edges and dashed lines represent *low* edges. The direction of the information flow is indicated by the double-lined arrows.

through the node N can be computed as $\alpha(N) \cdot \beta(N) \cdot (P(BDD))^{-1}$, where $\alpha(N)$ is the probability that a random walker starting from N will reach the 1-terminal and $\beta(N)$ is the probability that a random walker starting at the root will reach N. Hence in (7.4) we compute the probability of "a random walker starting at the root reaches N, and ends up in the 1-terminal when leaving N trough the high child".

The $\alpha(\cdot)$ probabilities express the likelihood of reaching the 1-terminal when starting from a particular node N and proceeding to the high child $h(N)$ with probability p_N and to the low child with probability $1 - p_N$. For the terminal nodes this *upward probability* is defined as

$$\alpha(0) := 0 \qquad \alpha(1) := 1$$

and for inner nodes, it is defined as

$$\alpha(N) := \alpha(h(N)) \cdot p_N^{\pi_N} + \alpha(l(N)) \cdot (1 - p_N)^{\pi_N} \ ,$$

Algorithm 14 Calculating the upward probability $\alpha(\cdot)$ for each node in a BDD

```
1: function ALPHA(bdd)
2:     Global α, visited                                   ▷ Global Variables
3:     α ← 0̂, visited ← 0̂          ▷ Array of 0's with one entry per BDD node
4:     α(1) ← 1, α(0) ← 0, visited(1) ← 1, visited(0) ← 1          ▷ Base Cases
5:     ALPHATRAVERSE(root(bdd))
6:     return α
7: end function
8: function ALPHATRAVERSE(N)
9:     Global α, visited                                   ▷ Global Variables
10:    if visited(N) = 0 then
11:        visited(N) ← 1
12:        Let h(N) and l(N) be the high and low child of N respectively
13:        ALPHATRAVERSE(h(N))
14:        ALPHATRAVERSE(l(N))
15:        α(N) ← (p_N)^{π_N} · α(h(N)) + (1 − p_N)^{π_N} · α(l(N))
16:    end if
17: end function
```

where π_N is an indicator function that takes the value 1 for probabilistic nodes and 0 for deterministic nodes and p_N is the fact probability of the fact represented by the node N. The $\alpha(\cdot)$ values can be computed for each node in the BDD by a level-wise algorithm starting from the terminal nodes (cf. Algorithm 14). The intermediate values are propagated *upwards* as illustrated in Figure 7.2(a).

The *downward probabilities* $\beta(\cdot)$ express the likelihood of reaching a node N when starting from the root node and following edges, similarly to $\alpha(\cdot)$, according to their probability. It is defined as $\beta(\mathit{Root}) := 1$ for the root node of the BDD. For any other node N it is defined as:

$$\beta(N) := \left[\sum_{\substack{M \in nodes(BDD) \\ N = h(M)}} \beta(M) \cdot (p_M)^{\pi_M} \right] + \left[\sum_{\substack{M \in nodes(BDD) \\ N = l(M)}} \beta(M) \cdot (1 - p_M)^{\pi_M} \right]$$

The first sum considers all possible parent nodes M of N that are connected by a *high* edge. When being in M the random walker will choose this edge with probability p_M. The second sum considers all possible parents nodes that are connected to N by a *low* edge, where the random walker will proceed from M to N with probability $1 - p_M$. The $\beta(\cdot)$ values can be computed for each node in the BDD by a level-wise algorithm starting from the root node as shown in Algorithm 15. It uses a priority queue for the BDD nodes to ensure that the values are computed level-wise starting from the root. This is crucial as the values on

Algorithm 15 Calculating the downward probability $\beta(\cdot)$ for each node in a BDD

```
1: function BETA(bdd)
2:     β ← 0̂                                  ▷ Array of 0's with one entry per BDD node
3:     q ←EMPTYPRIORITYQUEUE()                ▷ Sort according to BDD variable order
4:     ENQUEUE(q, root(bdd))                                          ▷ Base Case
5:     β(root(bdd)) ← 1                                               ▷ Base Case
6:     repeat
7:         N ←DEQUEUE(q)
8:         Let h(N) and l(N) be the high and low child of N respectively
9:         β(h(N)) ← β(h(N)) + β(N) · (p_N)^{π_N}
10:        β(l(N)) ← β(l(N)) + β(N) · (1 − p_N)^{π_N}
11:        ENQUEUEIFNOTYETCONTAINED(q, h(N))
12:        ENQUEUEIFNOTYETCONTAINED(q, l(N))
13:    until q is empty
14:    return β
15: end function
```

level n depend, due to the sum over all parent nodes, on the values level $n-1$. The intermediate values are propagated *downwards* as illustrated in Figure 7.2(b).

The upward probabilities $\alpha(\cdot)$ are identical to the values computed for inference in ProbLog by Algorithm 2 (cf. Chapter 3). Hence the probability $P(BDD)$ that is needed in Equation 7.4 is identical to $\alpha(Root)$. Furthermore, one can also show that $P(BDD) = \beta(1)$ and $1 - P(BDD) = \beta(0)$.

Moreover, for computing the expected counts, we also need to consider the nodes that have been removed in the construction of the *reduced* BDD. For instance, the probabilistic fact `earthquake` in Figure 7.1 is represented by the node on the left side, while the high edge of the root directly points to the 1-terminal. Implicitly, the fact `earthquake` is also represented on that path and we have to take this into account for the expected counts. Hence the sum in (7.2) needs to consider the removed nodes. If we would not consider such nodes, we would underestimate the expected counts. In the algorithms for computing $\alpha(\cdot)$ and $\beta(\cdot)$, we have to treat the missing atoms at a particular level as if they were there as illustrated in Figure 7.3. Since the BDD is ordered, that is, variables appear on all paths in the same order, we can easily detect missing nodes and output their probability. For the ease of notation, this step is not contained in the pseudocode of the algorithms.

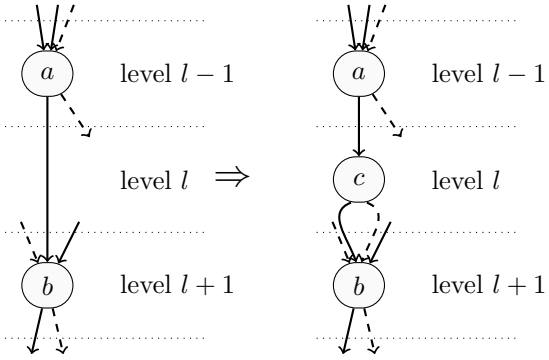


Figure 7.3: The algorithms need to compute $\alpha(c)$ and $\beta(c)$ even though the node c has been removed when reducing the BDD during its construction. It can be shown that $\beta(c) = \beta(a)$ and $\alpha(c) = \alpha(b)$.

7.2.3 Automated Theory Splitting

For large ground theories the corresponding BDDs are often too big to fit in memory. BDD tools use heuristics to find a variable order that minimizes the size of the BDD. The runtime of this step is exponential in the size of the input, which is prohibitive for parameter learning. We propose an algorithm that identifies independent parts of the grounded theory $clark(BK^r(I))$. The key observation is that the BDD for the Boolean formula $A \wedge B$ can be decomposed into two BDDs, one for BDD for A and one for B respectively, if A and B do not share a common variable. Since each variable is contained in at most one BDD, the expected counts of variables can be computed as the union of the expected count calculation on both BDDs.

This property can be exploited by the following algorithm that constructs an undirected graph as follows:

1. Add one node per clause in $clark(BK^r(I))$.
2. Add an edge between two nodes if the corresponding clauses share an atom.
3. Identify the connected components in the resulting graph.
4. Build for each of the connected components one BDD representing the conjunction of the clauses in the component.

The connected components of a graph can be computed in linear time [Hopcroft and Tarjan, 1973]. The experimental results show that the splitting is crucial for the applicability of LFI-ProbLog on large real-world datasets.

7.3 Experiments

We used two datasets to evaluate LFI-ProbLog. The *WebKB* dataset serves as test case to compare with state-of-the-art systems. The *Smokers* dataset is used to test the algorithm in terms of the learned model, that is, how close the parameters are to the original ones. The experiments were run on an Intel Core 2 Quad machine (2.83 GHz) with 8GB RAM.

7.3.1 WebKB

The goal of this experiment is to answer the following questions:

Q1 *Is LFI-ProbLog competitive with existing state-of-the-art frameworks?*

Q2 *Is LFI-ProbLog insensitive to the initial probabilities?*

Q3 *Is the theory splitting algorithm capable of handling large data sets?*

In this experiment, we used the WebKB [Craven and Slattery, 2001] dataset. It contains four folds, each describing the link structure of websites from one of the following universities: Cornell, Texas, Washington, and Wisconsin. WebKB is a collective classification task, that is, one wants to predict the class of a page depending on the classes of the pages that link to it and depending on the words used in the text. To allow for an objective comparison with Markov Logic networks and the results of Domingos and Lowd [2009], we used their version of WebKB, where each page is assigned exactly one of the classes "course", "faculty", "other", "researchproject", "staff" or "student". Furthermore, the class "person", present in the original version, has been removed.

We use the following model that contains one non-ground probabilistic fact for each pair of CLASS and WORD. To account for the link structure, it contains one non-ground probabilistic fact for each pair of CLASS1 and CLASS2.

P:: pfWoCla(Page, CLASS, WORD). P:: pfLiCla(Page1, Page2, CLASS1, CLASS2).

The probabilities P are unknown and have to be learned by LFI-ProbLog. As there are 6 classes and 771 words, our model has $6 \times 771 + 6 \times 6 = 4662$ parameters. In order to combine the probabilistic facts and predict the class of a page we add the following background knowledge.

cl(Pa, C) :− hasWord(Pa, Word), pfWoCla(Pa, Word, C).

cl(Pa, C) :− linksTo(Pa2, Pa), pfLiCla(Pa2, Pa, C2, C), cl(Pa2, C2).

This program is cyclic since the underlying graph formed by the `linksTo/2` predicate is cyclic. However, during learning the `cl/2` atoms are fully observable such that the resulting ground program is acyclic. This is due to the fact that our grounding algorithm replaces evidence atoms with their respective truth values before Clark's completion is computed.

We performed a 4-fold cross validation, that is, we trained the model on three universities and then tested it on the fourth one. We repeated this for all four universities and averaged the results. We measured the area under the precision-recall curve (AUC-PR), the area under the ROC curve (AUC-ROC), the log likelihood (LLH) and the accuracy after each iteration of the EM algorithm. Our model does not express that each page has exactly one class. To account for this, we normalize the probabilities per page.

Figure 7.4 (middle) shows the AUC-ROC plotted against the average training time. The initialization phase, that is running steps 1-4 of LFI-ProbLog, takes ≈ 330 seconds, and each iteration of the EM algorithm takes ≈ 62 seconds. We initialized the probabilities of the model randomly with values sampled from the uniform distribution between 0.1 and 0.9, which is shown as the graph for LFI-ProbLog [0.1-0.9]. After 10 iterations (≈ 800 s) the AUC-ROC is 0.950 ± 0.002, the AUC-PR is 0.828 ± 0.006, and the accuracy is 0.769 ± 0.010.

We compared LFI-ProbLog with Alchemy [Domingos and Lowd, 2009]. Alchemy is an implementation of Markov Logic networks. We use the model suggested by Domingos and Lowd, which uses the same features as our model, and we train it according to their setup.[2] The learning curve for AUC-ROC is shown in Figure 7.4 (middle). Alchemy achieves an AUC-ROC of 0.923 ± 0.016, an AUC-PR of 0.788 ± 0.036, and an accuracy of 0.746 ± 0.032 **Q1**.

We tested how sensitive LFI-ProbLog is for the initial fact probabilities by repeating the experiment with values sampled uniformly between 0.1 and 0.3 and sampled uniformly between 0.0001 and 0.0003 respectively. As the graphs in Figure 7.4 indicate, the convergence is initially slower and the initial LLH values differ. This is due to the fact that the ground truth probabilities are small, and if the initial fact probabilities are also small, one obtains a better initial LLH. All settings converge to the same results, in terms of AUC and LLH. This suggests that LFI-ProbLog is insensitive to the start values (cf. **Q2**).

The BDDs for the WebKB dataset are too large to fit in memory and the automatic variable reordering is unable to construct the BDD in a reasonable amount of time. We used two different approaches to resolve this. In the first approach, we manually split each training example, that is, the grounded theory together with the known class for each page, into several training examples. The results shown

[2] Daniel Lowd provided us with the original scripts for the experiment setup. We report on the evaluation based on the rerun of the experiment.

in Figure 7.4 are based on this manual split. In the second approach, we used the automatic splitting algorithm presented in Section 7.2.3. The resulting BDDs are identical to the manual split setting, and the subsequent runs of the EM algorithm converge to the same results. Hence when plotting against the iteration, the graphs are identical. The resulting ground theory is much larger and the initialization phase therefore takes 247 minutes. However, this is mainly due to the overhead for indexing, database access and garbage collection in the underlying Prolog system YAP [Santos Costa et al., 2011]. Grounding and Clark's completion take only 6 seconds each, the term simplification step takes roughly 246 minutes, and the final splitting algorithm runs in 40 seconds. As we did not optimize the implementation of the term simplification, we see a big potential for improvement, for instance by tabling intermediate simplification steps (**Q3**).

7.3.2 Smokers

We set up an experiment on an instance of the *Smokers* dataset (cf. [Domingos and Lowd, 2009]) to answer the following question

Q4 *Is LFI-Problog able to recover the parameters of the original model with a reasonable amount of data?*

Missing and incorrect values are two types of noise occurring in real-world data. While incorrect values can be compensated by additional data, missing values cause local maxima in the likelihood function. In turn, they cause the learning algorithm to yield parameters different from the ones used to generate the data. LFI-ProbLog computes the maximum likelihood parameters given some evidence. Hence the algorithm should be capable of recovering the parameters used to generate a set of interpretations. We analyze how the amount of required training data increases along with the size of the model. Furthermore, we test for the influence of missing values on the results. We assess the quality of the learned model, that is, the difference to the original model parameters by computing the Kullback-Leibler (K-L) divergence. ProbLog allows for an efficient computation of this measure due to the independence of the probabilistic facts. The details can be found in Appendix G.

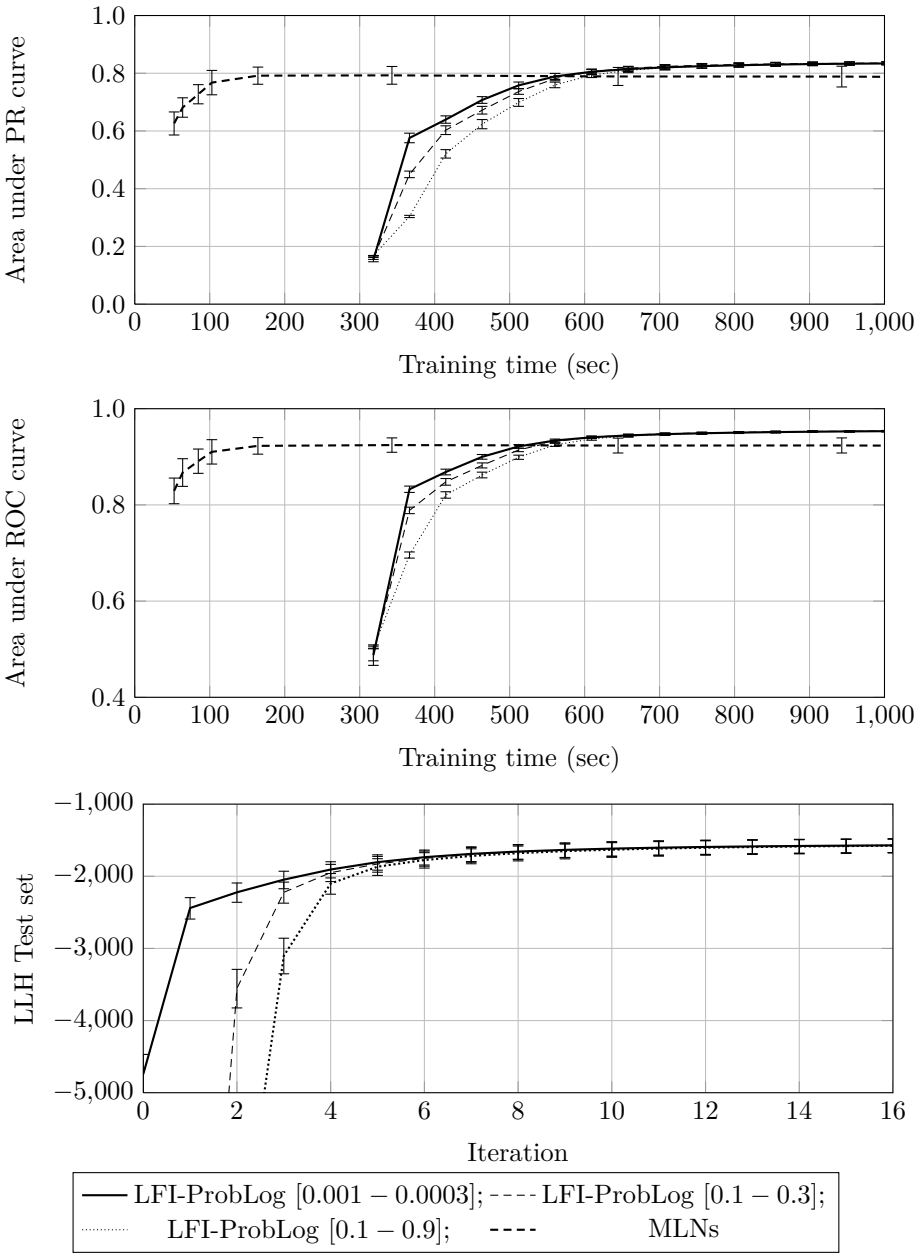


Figure 7.4: Results with LFI-ProbLog on WebKB. Area under the PR curve against the learning time (top); Area under the ROC curve against learning time (middle); Test set log likelihood against number of EM steps (bottom) (**Q1** and **Q2**).

In this experiment, we use a variant of the "Smokers" model that can be represented in ProbLog as follows:

$$
\begin{aligned}
F = \{ & p_{si} :: \texttt{smokes\_i(X,Y)}, && \text{// person X smokes due to smoking friend Y} \\
& p_{sp} :: \texttt{smokes\_p(X)}, && \text{// person X spontaneously starts smoking} \\
& p_{cs} :: \texttt{cancer\_s(X)}, && \text{// person X gets cancer due to smoking} \\
& p_{cp} :: \texttt{cancer\_p(X)}\} && \text{// person X spontaneously gets cancer}
\end{aligned}
$$

$$
\begin{aligned}
BK = \{ & \texttt{smokes(X)} :- \texttt{friend(X,Y)},\ \texttt{smokes(Y)},\ \texttt{smokes\_i(X,Y)} \\
& \texttt{smokes(X)} :- \texttt{smokes\_p(X)} \\
& \texttt{cancer(X)} :- \texttt{smokes(X)},\ \texttt{cancer\_s(X)} \\
& \texttt{cancer(X)} :- \texttt{cancer\_p(X)}\}
\end{aligned}
$$

This program is cyclic due to the `friend/2` relation. As not all relevant atoms are observed during learning, we cannot rely on the grounder to "break the cycles" and applying Clark's completion would yield incorrect results. Hence we used an acyclic version of this program. The underlying idea is, similarly to the path program used for probabilistic graphs, to extend each atom with an extra argument carrying the list of persons that have been visited so far within the current proof and fail the proof if a person is revisited. The loop breaking constructs have to be added to each clause, which make the resulting programing rather complex. Hence we use the path program to illustrate the idea, that is, instead of using the definition

```
path(X,Y) :- edge(X,Y).
path(X,Y) :- edge(X,Z), path(Z,Y).
```

one uses

```
path(X,Y) :- path(X,Y,[X],_).

path(X,X,A,A).
path(X,Y,A,R) :- X\==Y, edge(X,Z),
                 absent(Z,A), path(Z,Y,[Z|A],R).

absent(_,[]).
absent(X,[Y|Z]) :- X \= Y, absent(X,Z).
```

The resulting program is acyclic and Clark's completion can be applied.

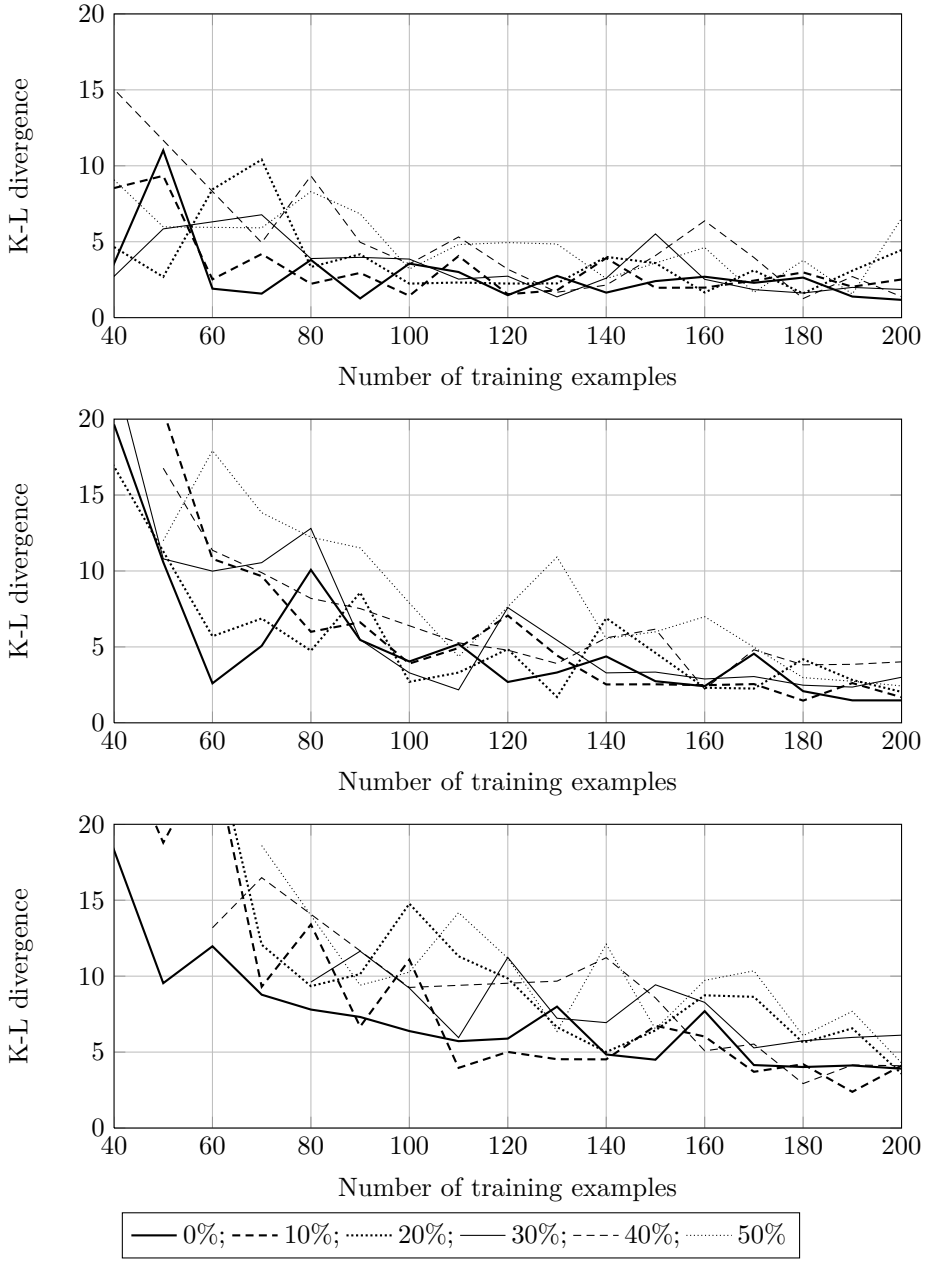


Figure 7.5: K-L divergence of the model trained using LFI-ProbLog on a Smokers data set with 3, 4 and 5 people (top, middle, bottom). The graphs represent different amounts of missing values in the training set (**Q4**).

This program does not model the `friend/2` relation but assumes this predicate to be given. We set the number of persons to 3, 4 and 5 respectively and sampled from the resulting models up to 200 interpretations each. From these datasets we derived new instances by randomly removing $10 - 50\%$ of the atoms. The size of an interpretation grows quadratically with the number of persons. The model, as described above, has an implicit parameter tying between ground instances of non-ground facts. Hence the number of model parameters (4, that is, p_{si}, p_{sp}, p_{cs}, and p_{cp}) does not change with the number of persons. To measure the influence of the model size, we trained grounded versions of the model, where the grounding depends on the number of persons. In turn number of ground probabilistic facts grows with the number of persons. Furthermore, we did not enforce parameter-tying between ground instances. For each dataset we ran LFI-ProbLog for 50 iterations of EM. Manual inspection showed that the probabilities stabilized after a few, typically 10, iterations. Figure 7.5 shows the K-L divergence for 3, 4 and 5 persons respectively. The closer the K-L divergence is to 0, the closer the learned model is to the original parameters. As the graphs show, the learned parameters approach the parameters of the original model as the number of training examples grows. Moreover, the amount of missing values has little influence on the distance between the true and the learned parameters. Hence LFI-ProbLog is capable of recovering the original parameters with a reasonable amount of training data and it is robust against missing values (**Q4**).

7.4 Related Work

Existing parameter learning approaches for ProbLog [De Raedt et al., 2007], PRISM [Sato and Kameya, 2001] and SLPs [Muggleton, 1996] are mainly based on learning from entailment. Sato and Kameya have contributed various interesting and advanced learning algorithms that have been incorporated in PRISM. However, all of them learn from entailment.

Ishihata et al. [2008] consider a parameter learning setting based on Binary Decision Diagrams (BDDs) [Bryant, 1986]. In contrast to our work, they assume the BDDs to be given, whereas LFI-ProbLog, constructs them in an intelligent way from evidence and a ProbLog theory. Ishihata et al. suggest that their approach can be used to perform learning from entailment for PRISM programs. This approach has been recently adopted by Bellodi and Riguzzi [2011] for learning CP-Logic programs.

The BDDs constructed by LFI-ProbLog are a compact representation of all possible worlds that are consistent with the evidence. LFI-ProbLog estimates the marginals of the probabilistic facts in a dynamic programming manner on the BDDs. While this step is inspired by [Ishihata et al., 2008], we tailored it towards the specifics of LFI-ProbLog, that is, we allow deterministic nodes to be present in the BDDs.

This extension is crucial as the removal of deterministic nodes can result in an exponential growth of the Boolean formulae underlying the BDD construction.

Ishihata et al. mention the possibility of using their approach in the context of parameter learning for PRISM. As they seem to suggest to adopt the BDD generation of the original ProbLog system towards PRISM, it implies a system for learning from entailment. On the other hand, it would allow one to relax the exclusive explanation assumption made by PRISM. This assumption requires that different proofs for a particular fact are disjoint. It allows the current PRISM system to avoid the use of BDDs and optimizes both learning and inference algorithms. Conversely, ProbLog does not make this assumption, which simplifies modeling.

The upward/downward procedure used in our algorithm to estimate the parameters from a set of BDDs is essentially an extension of that of the approaches of Ishihata et al. and of Thon et al. [2008, 2011] that have been independently developed. The algorithm of Ishihata et al. learns the probability of literals for arbitrary Boolean formulae from examples using a BDD, while that of Thon et al. is tailored towards learning with BDDs for the sequential logic CPT-L.

Riguzzi [2007] uses a transformation of ground ProbLog programs to Bayesian networks in order to learn ProbLog programs from interpretations. Such a transformation is also employed in the learning approaches for CP-logic [Vennekens et al., 2006] like the setting considered by Meert et al. [2008]. CPT-L is a variant of CP-Logic tailored towards sequences. Thon et al. [2011] studied how it can be learned from sequences of interpretations. CPT-L is closely related to LFI-ProbLog. However, CPT-L is targeted towards the sequential aspect of the theory, whereas we consider a more general settings with arbitrary theories. Thon et al. assume full observability, which allows them to split the sequence into separate transitions. They build one BDD per transition, which is much easier to construct than one large BDD per sequence. Our splitting algorithm (cf. Sect 7.2.3) is capable of exploiting arbitrary independence. This improves the applicability and, when applied on sequential data, it detects the independence between time steps. Exploiting independence in Boolean formulae is known in the verification literature, for instance in the Deterministic Decomposable Negation Normal form (d-DNNF) [Darwiche, 2004].

Our approach can also be related to the work on knowledge-based model construction approaches in statistical relational learning such as BLPs, PRMs and MLNs [Richardson and Domingos, 2006]. While the setting explored in this chapter is standard for the aforementioned formalisms, our approach has significant representational and algorithmic differences from the algorithms used in those formalisms. In BLPs, PRMs and CP-logic, each training example is typically used to construct a ground Bayesian network, on which a standard learning algorithm is applied. Although the representation generated by Clark's completion is quite close to the representation of Markov Logic, there are subtle differences. While

Markov Logic uses weights on clauses, we use probabilities attached to single facts. Moreover, our approach has a clear probabilistic semantics. Using BDDs for performing inference in Markov Logic networks is not entirely obvious. However, it seems to be an interesting research direction.

7.5 Conclusions And Future Work

We have introduced a novel parameter learning algorithm from interpretations for the probabilistic logic programming language ProbLog. This has been motivated by the differences in the learning settings and applications of typical knowledge-based model construction and probabilistic logic programming approaches. The LFI-ProbLog algorithm tightly couples logical inference with an EM algorithm at the level of BDDs. The chapter provides an empirical evaluation which demonstrates the applicability of the proposed algorithm to the types of problems commonly tackled by knowledge-based model construction approaches to statistical relational learning.

The LFI-ProbLog algorithms use Clark's completion to translate the original theory into Boolean formulae. This operation requires acyclic programs in order to guarantee semantic consistency between the theory and the resulting formula. Using a translation that does not depend on the program to be acyclic will make LFI-ProbLog better suited for many applications. The transformation by Janhunen [2004], for instance, is a potential candidate for this task. The intuition behind it is the augmentation of Clark's completion with loop-breaking constraints. In order to improve the performance in terms of speed, one could also use other data structures apart from BDDs. So-called d-DNNF graphs [Darwiche, 2004], for instance, represent Boolean formulae similarly to BDDs. They introduce special nodes that allow them to exploit more and different kinds of independence between the Boolean variables. For instance, the theory splitting in LFI-ProbLog identifies parts of the ground theory that are independent given the evidence. This is a special case of the independencies considered in d-DNNFs and not every theory contains them. To some extent, these two ideas have already been studied by Fierens et al. [2011]. While it is theoretically straightforward to transfer the results to LFI-ProbLog, there are open questions with respect to the implementation.

Conclusions of Part III

In this part of the thesis we studied how the parameters of ProbLog programs can be estimated using machine learning techniques.

In Chapter 6, we developed the LFE-ProbLog algorithm. It is able to learn from queries as well as from proofs. The algorithm optimizes the fact probabilities of the Problog program such that the predictions on the training set match the target probabilities. A ProbLog program does not specify a generative model at the level of proofs or queries. Hence we mapped the learning problem for LFE-ProbLog on a logistic regression problem, which was solved using a gradient descent search.

In Chapter 7, we developed the LFI-ProbLog algorithm that learns from both complete and partial interpretations. Since a ProbLog program defines a probability distribution over interpretations, we could solve this learning task by applying an expectation maximization (EM) algorithm. This algorithm builds a BDD based on a transformation of the program as a whole, in contrast to the exact inference algorithm that uses individual proofs.

Part IV

Round-Up

Chapter 8

Summary and Future Work

Thesis Summary

We studied two important questions that had previously not received a lot of attention in the context of probabilistic logic programming languages.

> (Q1) How can continuous distributions be integrated in probabilistic logic programming languages?
>
> (Q2) How can the parameters of probabilistic programming languages be estimated?

Continuous-valued information is crucial in many real-world applications that can benefit from using logical representations, such as robotics and human activity recognition. Spatial and temporal relations in these domains are naturally expressed using numbers, while the background knowledge is often specified in terms of logical expressions. Moreover, parameter learning methods allow domain experts to focus on the qualitative aspects of the model while the strength of dependencies, for instance, can be tuned by means of parameter learning algorithms.

We used ProbLog as a representative of the probabilistic logic programming paradigm, although our results can easily be translated into other languages and frameworks. We introduced two different extensions of ProbLog with continuous distributions. We deliberately restricted the expressivity of the first extension, Hybrid ProbLog, in order to allow for exact inference. The second extension does not impose any restrictions on continuous distributions. Hence the resulting distributional programs are more expressive and in turn more suitable for real-world applications. With respect to parameter learning we proposed two algorithms that

cover all relevant learning settings known in the logical and relational literature (cf. [De Raedt, 2008]): learning from proofs, queries, and interpretations. In the following we summarize the four contributions mentioned above.

Contribution 1: Hybrid ProbLog Hybrid ProbLog introduces *continuous facts* to ProbLog. One denotes a continuous fact by $(X, D) :: Atom$, where D specifies the distribution of the variable X in $Atom$. For instance, one can model the height of an average male human as

$$(\texttt{H}, \texttt{gaussian}(\texttt{1.78}, \texttt{0.2})) :: \texttt{height}(\texttt{male}, \texttt{H}) \ .$$

While the changes in syntax are rather small when augmenting ProbLog with such facts, the extra efforts to account for them in the inference algorithms are much more complex. We had to limit the possible use cases of the values stemming from continuous facts, i.e., the variable `H` in the example above. We allowed only for comparison against number constants but not of two continuous values against one another. Doing so, we avoided the coupling of two more variables and the resulting complex integration problems. This allows for an efficient inference algorithm that uses the number constants occurring in the proofs as the basis for a dynamic discretization and the subsequent representation as Binary Decision Diagram. To the best of the author's knowledge, this technique was not applied on probabilistic programming languages before.

Contribution 2: Distributional Programs We developed the language of distributional clauses as an extension of ProbLog, Hybrid ProbLog and CP-Logic such that its semantics can be mapped onto Sato's [1995] distribution semantics. Distributional programs are more expressive than Hybrid ProbLog, which in turn makes them most suitable for real-world applications. As a result we needed to rely on an approximate inference algorithm based on rejection sampling. The additional information given by the structure of the logic program can be used to guide the sampling process, which allows for a more efficient inference algorithm compared to naïve rejection sampling. Akin to SampleSearch [Gogate and Dechter, 2011], we developed a lookahead-based algorithm that locally modifies the distributions, while checking the evidence globally. Moreover, we adopted the Magic sets transformation [Bancilhon et al., 1986] to distributional clauses. This enabled us to restrict the sample generation to the necessary finite part of the ground program and more importantly, it allows for inference although the complete ground program is infinite such as in temporal models.

Contribution 3: Learning from Probabilistic Entailment In this setting, we train a ProbLog theory on examples comprising queries and their target probability. In biomedical applications, for instance, such an example can be a relation between

a gene and a disease annotated with the probability that this relation holds. Additionally, this setting supports learning from proofs, since proofs can be represented as the conjunction of the probabilistic facts they use. The learning algorithm optimizes the parameters of the model such that the predicted query probabilities match the target probabilities. As a ProbLog program does not specify a generative model on the level of individual queries, we represented the learning task as a logistic regression problem and solved it using a gradient-descent search. The gradients can be computed efficiently on the BDDs, which in turn allows one to use parameter learning in all domains where inference is possible.

Contribution 4: Learning from Interpretations A major drawback of not having a generative setting, when learning from queries, is that the learner treats the model as a black box. Hence the estimated parameters, despite fitting the training data perfectly, do not necessarily have an intuitive meaning. On the contrary, ProbLog does specify a generative model at the level of interpretations. Such possible worlds are at the core of Sato's [1995] distribution semantics. Consequently, it is possible to estimate the parameters of a model based on a set of *complete* interpretations; simply counting the number of true ground instances and dividing by the number of examples yields the correct estimates. However, complete interpretations, especially in real-world applications, are often too large to fit in memory and often substantial parts of the world cannot be observed. Thus, when learning from interpretations in the context of ProbLog, one actually needs to be able to learn from *partial interpretations*. We developed LFI-ProbLog that is able to accomplish this task. While this learning setting has been considered for other probabilistic relational frameworks such as PRISM [Sato, 1995], relational Bayesian networks [Jaeger, 2007], and Markov Logic Networks [Richardson and Domingos, 2006], it was not considered before for ProbLog programs.

Hence the parameters of ProbLog programs can now be estimated in all three learning settings considered in probabilistic logic programming. The two main ideas underlying LFI-ProbLog are: translating the theory together with the evidence into a BDD and using it for estimating the number of ground instances of the probabilistic facts. The experimental evaluation of the resulting system LFI-ProbLog demonstrated its performance as being on par with state-of-the-art frameworks, while having an interpretable model.

Future Work

Currently, the BDDs are the bottleneck in the parameter learning algorithms that we developed. Evaluating them in every iteration takes a lot of time and for larger datasets it is impossible to store all of the them in memory. Hence the BDDs have to be reconstructed anew from the underlying Boolean functions in each iteration.

In [Fierens et al., 2011], we have started to investigate the use of alternative data structures such as d-DNNFs and we are currently working on integrating them into the ProbLog system. However, on challenging and particularly large-scale real-world tasks the performance improvements in terms of runtime might not be sufficient as the size of the Boolean functions typically grows exponentially. Lifted inference algorithms (cf. [Poole, 2003; Milch et al., 2008]) aim at grounding as few atoms as possible exploiting the fact that most ground atoms share the same distribution. Adopting lifted techniques for parameter learning is an interesting research direction that will broaden the applicability of probabilistic logic languages.

More work needs to be done on making the learning algorithms applicable on *noisy* training data. Currently, the algorithms implicitly assume the model to take noise into account. Particularly LFI-ProbLog, which heavily relies on the consistency of the training data with the theory, requires carefully-designed models. Integrating noise handling techniques in the learning algorithms will thus simplify the modeling process and make probabilistic logic languages easier to use. It is interesting to adopt ideas from Support Vector Regression [Smola and Schölkopf, 2004] and, for instance, automatically weight the influence of each training example based on the estimated noise.

Another natural direction to proceed is the development of structure learning techniques for ProbLog and probabilistic logic languages in general. De Raedt and Thon [2011] have considered a structure learning approach that generates non-recursive rules using probabilistic training examples. Continuing this line of research and upgrading "classical" inductive logic programming algorithms to the probabilistic languages would be worth investigating.

Using probabilistic logical learning techniques for robotics applications has recently become popular due to the maturity of the field and the available frameworks. The problems considered within robotics [Thrun et al., 2005], such as localization and people tracking, do require continuous-valued information. The models applied to robotics tasks are often based on the Bayesian principle [De Laet, 2010], that is, they express the agent's belief in terms of a posterior distribution computed from the *prior belief* and the evidence. To be applicable on such tasks, probabilistic logic programming languages need to be able to express such regularities and they need to provide the corresponding inference mechanisms. More work has to be done, in order to apply Hybrid ProbLog or distributional programs on such tasks. Exact inference algorithms are often not applicable as the resulting inference problems are too complex. Conversely, the sampling algorithm we have developed for distributional programs is currently not able to handle continuous-valued evidence. An interesting direction for future work is to adopt the ideas behind dynamic discretization to distributional programs.

One more open question is how to estimate the parameters of continuous distributions together with the fact probabilities. It is straightforward to extend

LFE-ProbLog's gradient descent search towards Gaussian distributions as shown in Appendix E. Hence it is theoretically possible to apply LFE-ProbLog on continuous facts. In practice, this will require a second-order gradient algorithm such as LBFGs due to small values of the gradient in the tail regions of the distributions. Parameter estimation for distributional programs is conceptually much simpler. Any MCMC approach for inference also provides a learning algorithm as it can be used to estimate the parameters of the distributions given the evidence.

Appendix A

Correctness of Mapping Annotated Disjunctions

In the following we show that the mapping of annotated disjunctions introduced in Section 3.3 is correct, that is, the success probabilities in the resulting program are identical to probabilities attached to the head atoms of the AD. For the ease of notation we assume that the AD is ground, that the body of the AD is empty and that all atoms in the head are different.

Theorem A.1. *Let* $p_1 :: \mathtt{h}_1;\ \ldots;\ p_N :: \mathtt{h}_N :- \mathtt{true}$ *be a ground annotated disjunction; let* $T = F \cup BK$ *be a ProbLog theory with the probabilistic facts* $F = \{\tilde{p}_1 :: \mathtt{msw}(1), \ldots, \tilde{p}_N :: \mathtt{msw}(N)\}$ *where* $\tilde{p}_1 = p_1$ *and for* $i > 1$

$$\tilde{p}_i = \begin{cases} p_i \cdot \left(1 - \sum_{k=1}^{i-1} p_k\right)^{-1} & \text{if } p_i > 0 \\ 0 & \text{if } p_i = 0 \end{cases}$$

and the background knowledge BK *consisting of the clauses (* for $i \in \{1, \ldots, N\}$*)*

$$\mathtt{h}_i :- \mathtt{msw}(i),\ \mathtt{not}(\mathtt{msw}(i-1)),\ \ldots,\ \mathtt{not}(\mathtt{msw}(1)).$$

Then $P_s^T(\mathtt{h}_i) = p_i$ *for all* $i \in \{1, \ldots, N\}$*, that is, the success probability of the query* $\mathtt{h}_i$ *in the ProbLog theory* T *is equal the probability of the corresponding head atom in the AD.*

Proof. The theory T has exactly one proof for $\mathtt{h_i}$. Hence the success probability of the query is identical to the probability of its single proof, that is, $\mathcal{P}_i := \left(\prod_{k=1}^{i-1}(1-\tilde{p}_k)\right) \cdot \tilde{p}_i$. The factors $(1 - \tilde{p}_k)$ stem from the negated part

$\texttt{not}(\texttt{msw}(\texttt{i}-\texttt{1})),\ldots,\texttt{not}(\texttt{msw}(\texttt{1}))$ and the last factor $\tilde{p}_i$ stems from the first atom in the body. It is sufficient to prove that $\mathcal{P}_i = p_i$. We show this by induction over the position of the choice i. For the base case $i = 1$ we get

$$\mathcal{P}_1 = \underbrace{\left(\prod_{k=1}^{0}(1-\tilde{p}_k)\right)}_{\text{product with no factors}} \cdot\ \tilde{p}_1 = 1 \cdot p_1 = \ p_1\ .$$

And for the inductive case $i \to i+1$ we get for $\tilde{p}_i > 0$:

$$\begin{aligned}\mathcal{P}_{i+1} &= \left(\prod_{k=1}^{i}(1-\tilde{p}_k)\right)\cdot\tilde{p}_{i+1}\\ &= \left(\prod_{k=1}^{i-1}(1-\tilde{p}_k)\right)\cdot(1-\tilde{p}_i)\cdot\tilde{p}_{i+1}\end{aligned}$$

where applying the induction assumption yields

$$\begin{aligned}&= \frac{p_i}{\tilde{p}_i}\cdot(1-\tilde{p}_i)\cdot\tilde{p}_{i+1}\\ &= \left(\frac{p_i}{\tilde{p}_i} - p_i\right)\cdot\tilde{p}_{i+1}\end{aligned}$$

We use the definition of $\tilde{p}_i$ from Equation 3.10 and get

$$\begin{aligned}&= \left(\frac{p_i}{p_i\cdot\left(1-\sum_{j=1}^{i-1}p_j\right)^{-1}} - p_i\right)\cdot\tilde{p}_{i+1}\\ &= \left(1-\sum_{j=1}^{i}p_j\right)\cdot\tilde{p}_{i+1}\end{aligned}$$

We use the definition of $\tilde{p}_{i+1}$ from Equation 3.10 and get

$$= p_{i+1}$$

If $\tilde{p}_i = 0$ then we have to apply the inductive assumption for $i-1$, that is, use $\mathcal{P}_{i-1}$ for rewriting the term. □

Appendix B

Properties of The Sigmoid Function

In Chapter 6 we develop a gradient-descent method for estimating the fact probabilities of a ProbLog theory. Akin to logistic regression we represented probabilities using the sigmoid function

$$\sigma_s(a) = \frac{1}{1 + \exp(-s \cdot a)} \ , \tag{B.1}$$

where $a \in \mathbb{R}$ and $s > 0$. In the following we show two properties of this function that we used while deriving the gradient.

Lemma B.1 (Properties of Sigmoid function)**.** $\forall a \in \mathbb{R}\ \forall s > 0$

$$1 - \sigma_s(a) = \sigma_s(-a) \tag{B.2}$$

$$\frac{\partial}{\partial a}\ \sigma_s(a) = s \cdot \sigma_s(a) \cdot (1 - \sigma_s(a)) \tag{B.3}$$

Proof. Eq. B.2 can be shown as follows:

$$
\begin{aligned}
1 - \sigma_s(a) &= 1 - \frac{1}{1 + \exp(-s \cdot a)} && \text{by (B.1)} \\
&= \frac{1 + \exp(-s \cdot a)}{1 + \exp(-s \cdot a)} - \frac{1}{1 + \exp(-s \cdot a)} && \text{by reordering} \\
&= \frac{\exp(-s \cdot a)}{1 + \exp(-s \cdot a)} && \text{by reordering} \\
&= \frac{\exp(-s \cdot a) \cdot \exp(s \cdot a)}{\exp(s \cdot a) + \exp(-s \cdot a) \cdot \exp(s \cdot a)} && \text{expand with } \exp(s \cdot a) \\
&= \frac{1}{\exp(s \cdot a) + 1} && \text{by reordering} \\
&= \sigma_s(-a) && \text{by (B.1)}
\end{aligned}
$$

Eq. B.3 can be shown as follows:

$$
\begin{aligned}
\frac{\partial}{\partial a}\, \sigma_s(a) &= \frac{\partial}{\partial a}\, \frac{1}{1 + \exp(-s \cdot a)} && \text{by (B.1)} \\
&= (-1) \cdot \left(\frac{1}{1 + \exp(-s \cdot a)} \right)^2 \cdot \exp(-s \cdot a) \cdot (-s) && \text{by chain rule} \\
&= s \cdot \frac{1}{1 + \exp(-s \cdot a)} \cdot \frac{\exp(-s \cdot a)}{1 + \exp(-s \cdot a)} && \text{by reordering} \\
&= s \cdot \sigma_s(a) \cdot \frac{\exp(-s \cdot a)}{1 + \exp(-s \cdot a)} && \text{by (B.1)} \\
&= s \cdot \sigma_s(a) \cdot \frac{1 - 1 + \exp(-s \cdot a)}{1 + \exp(-s \cdot a)} && \text{by reordering} \\
&= s \cdot \sigma_s(a) \cdot \left(\frac{1 + \exp(-s \cdot a)}{1 + \exp(-s \cdot a)} - \frac{1}{1 + \exp(-s \cdot a)} \right) && \text{by reordering} \\
&= s \cdot \sigma_s(a) \cdot (1 - \sigma_s(a)) && \text{by (B.1)}
\end{aligned}
$$

□

Appendix C

Translating Markov Logic Clauses Into ProbLog Clauses

A Markov logic network (MLN) is a set of weighted first-order clauses [Richardson and Domingos, 2006]. 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 clause, $g_i = 1$ if the ith clause is true and $g_i = 0$ otherwise. Inference can be carried out by generating the ground corresponding Markov network and running any Markov network inference algorithm such as belief propagation (cf. [Bishop, 2006]).

To convert the Markov logic clauses into ProbLog clauses (see the experiments on UW-CSE dataset in Section 6.5), we use the function ConvertMLNClause shown in Algorithm 16. It takes the clause c as input 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. Clause weights are not transformed, the corresponding fact probabilities are set to p_i indicating that they have to be estimated using parameter learning.

Algorithm 16 The function ConvertMLNClause translates a clause c, which is a disjunction of positive c^+ and negative c^- literals, into a set of clauses and probabilistic facts. If the clause does not contain positive literals, we use an error predicate and an additional training example to preserve the meaning of the clause.

```
1: function ConvertMLNClause(c)
2:     if c+ = ∅ then return {error :− c1−, c2−, ..., ck−}
3:     result ← ∅
4:     for atom ∈ c+ do
5:         n ← UniqueNumber()
6:         result ← result ∪ {pn :: fact(n)}
7:                         ∪ {atom :− fact(n), c1−, c2−, ..., ck−}
8:     end for
9:     return result
10: end function
```

Table C.1 shows some examples of translated clauses. ProbLog cannot represent clauses that contain only negative literals, i.e., the third example in the table

$$\neg\texttt{student(X)} \vee \neg\texttt{professor} \ .$$

To represent such clauses in ProbLog, one can rewrite them as follows

$$\neg\texttt{student(X)} \vee \neg\texttt{professor} \vee \neg\texttt{false}$$

such that they can be translated into

```
false :− student(X), professor(X) .
```

Instead of `false`, we use the predicate `error`. In contrast to the other clauses generated by our algorithm, `error` clauses do not contain probabilistic facts in the body. Instead, we have to provide one additional training example for each department `D` of the form `example(error(D),0.0)`. 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 the 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. Nonetheless, it is possible to use non-ground facts. The first clause in Table C.1, for instance, can be translated into:

```
p1 :: fact(1,P,S).
professor(Department,P) :−
   advisedBy(Department,P,S),
   fact(1,P,S).
```

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 C.1: Clauses in Markov Logic and their translation using Algorithm 16. Note that in order to deal with clauses consisting only out of negated atoms, like in the third example, we have to provide additional training examples. The atoms contain an argument `Department` that enables us to store all examples in a single database.

MLN Clause	Resulting ProbLog Code
w_1 $\neg$advisedBy(P, S) $\vee$ professor(P)	p_1 :: fact(1). professor(Department, P) :− fact(1), advisedBy(Department, P, S).
w_2 $\neg$tempAdvisedBy(X, Y) $\vee$ yearsInProgram(X, 1) $\vee$ yearsInProgram(X, 2)	p_2 :: fact(2). yearsInProgram(Department, X, 1) :− fact(2), tempAdvisedBy(Department, X, Y). p_3 :: fact(3). yearsInProgram(Department, X, 1) :− fact(3), tempAdvisedBy(Department, X, Y).
w_3 $\neg$student(X) $\vee$ $\neg$professor(X)	error(Department) :− student(Department, X), professor(Department, X). example(error(Department), 0.0).

Appendix D

Correctness of Gradient Computation Algorithm

In Chapter 6 we develop the LFE-ProbLog algorithm that trains a ProbLog program on a given training set by means of gradient descent algorithm. This approach uses the function GRADIENT(node n, target fact n_j) to compute the gradient of the k-best probability of a query, see Algorithm 10 and Equation 3.9 respectively. In the following we show the correctness of this algorithm.

Theorem D.1. GRADIENT($root(BDD_F), n_j$) *returns the probability* $P(F = true)$ *and the partial derivative* $\partial P(F = true)/\partial a_j$ *if* BDD_F *is a BDD representing the Boolean function* F*.*

Proof. In order to prove Theorem D.1 and to show that Algorithm 10 computes the gradient of the k-best probability it suffices to show that it computes the correct gradient of the function represented by the BDD. We do this by induction over the structure of the BDD that is given by the Shannon expansion [Shannon, 1948] of the underlying Boolean formula F.

Base case 1 If the BDD consists only of the 1-terminal, it represents the function $F = true$ and the probability that F is true is 1. The partial derivate $\partial P(F)/\partial a_j$ is 0.

Base case 2 If the BDD consist only of the 0-terminal, it represents the function $F = false$ and the probability of F being true is 0. The partial derivative $\partial P(F)/\partial a_j$ is 0.

Inductive case Due to the Shannon expansion used by the BDD, the function F is represented as $F = (n \wedge F_{True}) \vee (\neg n \wedge F_{False})$, where both F_{True} and F_{False} are BDDs and the variable n does neither appear in F_{True} nor in F_{False}. Since both branches are disjoint with respect to n, one can calculate the probability that F yields true as

$$P(F = true) = P(n) \cdot P(F_{True} = true) + (1 - P(n)) \cdot P(F_{False} = true) \ .$$

The probability $P(n) = \sigma(a_n)$ is defined by the corresponding probabilistic fact

$$\sigma(a_n) :: f_n \ .$$

The complementary probability $1 - P(n)$ is $\sigma(-a_n)$ (cf Lemma B.1). See Equation 6.2 for the definition of the sigmoid function σ. We apply the inductive hypothesis that says, the algorithm returns the correct probabilities when applied to F_{True} and F_{False}. Hence the value for *prob*, calculated in Line 7 of Algorithm 10, is correct.

The partial derivative of $P(F)$ with respect to a_j, in the inductive case

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

requires one to distinguish the following two cases.

Case 1 If $n \not\trianglelefteq_\Theta n_j$, namely the current node n does not represent a ground instance of the target fact f_j, then one can treat $P(n)$ as constant factor:

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

We apply the inductive hypothesis that says, the algorithm returns the correct gradient when applied to F_{True} and F_{False}. This part of the proof covers Line 8 of Algorithm 10.

Case 2 If $n \subseteq_\Theta n_j$, namely the current node n represents a ground instance of the target fact f_j, we get

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

we replace $P(n)$ by the corresponding probability $\sigma(a_j)$

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

and apply the product rule on both summands

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

and reorganize the terms to

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

We apply the inductive hypothesis that says, the algorithm returns the correct partial derivative and probability when applied to F_{True} and F_{False}. This part of the proof covers Line 9 of Algorithm 10. □

Appendix E

Gradient Computation for Hybrid ProbLog

In Chapter 4 we introduced Hybrid ProbLog and showed how exact inference can be carried out in programs with continuous facts. The algorithm relies on dynamic discretization and introduces so-called auxiliary facts that are annotated with a cumulative density function. For instance, the auxiliary fact $\texttt{call\_temp(T)}_{[0,5)}$ expresses that the value of the variable T is within the interval $[0, 5)$. These auxiliary facts get annotated with a probability that is computed by integrating the density of corresponding continuous fact, i.e., $P(\texttt{call\_temp(T)}_{[0,5)}) = \int_0^5 D(x)dx$. The details can be found in Section 4.2.

In this appendix, we show how one can obtain the partial derivative of their probability with respect to the parameters of the distribution. This allows one to extend the gradient-descent search used by LFE-ProbLog (cf. Chapter 6) towards continuous distributions and to automatically estimate their parameters. Please note that we restrict ourselves to the partial derivative of the cumulative density function, while ignoring dependencies between the auxiliary facts, that is, the bodies of the auxiliary clauses. In the following we assume the continuous facts to be annotated with a Gaussian. Hence the probability attached to auxiliary facts is defined as

$$P(\texttt{call\_f}_{[X_0,X_1)}) = \int_{X_0}^{X_1} \varphi_{\mu;\sigma}(x)\, dx \tag{E.1}$$

where $\varphi_{\mu;\sigma}$ is the density function of a Gaussian distribution

$$\varphi_{\mu;\sigma}(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right) \tag{E.2}$$

with mean $\mu \in \mathbb{R}$ and standard deviation $\sigma > 0$. The following theorem states that the partial derivatives of (E.1) can be computed efficiently by evaluating the density at the borders of the interval $[X_0, X_1)$.

Theorem E.1. *Let* $\texttt{call\_f}_{[X_0,X_1)}$ *be an auxiliary fact introduced by the Hybrid ProbLog inference algorithm. The partial derivatives of* $P(\texttt{call\_f}_{[X_0,X_1)})$ *are*

$$\frac{\partial}{\partial\mu} P(\texttt{call\_f}_{[X_0,X_1)}) = \frac{1}{\sigma}\varphi_{\mu;\sigma}(X_0) - \frac{1}{\sigma}\varphi_{\mu;\sigma}(X_1) \tag{E.3}$$

$$\frac{\partial}{\partial\sigma} P(\texttt{call\_f}_{[X_0,X_1)}) = \frac{X_0-\mu}{\sigma}\varphi_{\mu;\sigma}(X_0) - \frac{X_1-\mu}{\sigma}\varphi_{\mu;\sigma}(X_1)\ . \tag{E.4}$$

In the proof of this theorem we assume the points X_0, X_1 to be finite. Extending this to the border cases $\texttt{call\_f}_{(-\infty,X_1)}$ and $\texttt{call\_f}_{[X_0,\infty)}$ will be straightforward. Please note that the standard deviation σ has to be larger than 0, which needs to be taken into account by the gradient-descent algorithm. A possible solution is the use of a transformation function. For instance, one can represent the standard deviation as $\sigma = \exp(a)$ with an arbitrary $a \in \mathbb{R}$. This is akin to the use of the sigmoid function (cf. (6.2)) in LFE-ProbLog for representing fact probabilities. We omit this transformation for the ease of readability of the proofs. In order to show Theorem E.1 we require the following lemma.

Lemma E.1. *The partial derivatives of* $\varphi_{\mu;\sigma}(x)$ *with respect to the parameters* μ *and* σ *of the normal density* (E.2) *are*

$$\frac{\partial}{\partial\mu}\varphi_{\mu;\sigma}(x) = \varphi_{\mu;\sigma}(x)\cdot\frac{x-\mu}{\sigma^2} \tag{E.5}$$

$$\frac{\partial}{\partial\sigma}\varphi_{\mu;\sigma}(x) = \varphi_{\mu;\sigma}(x)\cdot\frac{(x-\mu)^2-\sigma^2}{\sigma^3}\ . \tag{E.6}$$

Proof. The derivation of (E.5) can be shown as follows

$$\begin{aligned}
\frac{\partial}{\partial\mu}\varphi_{\mu;\sigma}(x) &= \frac{\partial}{\partial\mu}\left[\frac{1}{\sigma\sqrt{2\pi}}\exp\left(-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right)\right] && \text{by (E.2)}\\
&= \frac{1}{\sigma\sqrt{2\pi}}\exp\left(-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right)\frac{\partial}{\partial\mu}\left[-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right] && \text{by chain rule}\\
&= \varphi_{\mu;\sigma}(x)\cdot\frac{\partial}{\partial\mu}\left[-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right] && \text{by (E.2)}\\
&= \varphi_{\mu;\sigma}(x)\cdot\left[-\frac{1}{2}\cdot 2\cdot\left(\frac{x-\mu}{\sigma}\right)\cdot\left(\frac{-1}{\sigma}\right)\right] && \text{by chain rule}\\
&= \varphi_{\mu;\sigma}(x)\cdot\frac{x-\mu}{\sigma^2} && \text{by reordering}
\end{aligned}$$

The derivation of (E.6) can be shown as follows

$$\begin{aligned}
\frac{\partial}{\partial\sigma}\varphi_{\mu;\sigma}(x) &= \frac{\partial}{\partial\sigma}\left[\frac{1}{\sigma\sqrt{2\pi}}\exp\left(-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right)\right] && \text{by (E.2)}\\
&= \frac{-1}{\sigma^2\sqrt{2\pi}}\exp\left(-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right)\\
&\quad + \frac{1}{\sigma\sqrt{2\pi}}\frac{\partial}{\partial\sigma}\left[\exp\left(-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right)\right] && \text{by product rule}\\
&= \frac{-1}{\sigma}\varphi_{\mu;\sigma}(x) && \text{by (E.2)}\\
&\quad + \frac{1}{\sigma\sqrt{2\pi}}\exp\left(-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right)\frac{\partial}{\partial\sigma}\left[-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right] && \text{by chain rule}\\
&= \frac{-1}{\sigma}\varphi_{\mu;\sigma}(x) + \varphi_{\mu;\sigma}(x)\cdot\left[-\frac{1}{2}\frac{(x-\mu)^2}{\sigma^3}\cdot(-2)\right] && \text{by chain rule}\\
&= \varphi_{\mu;\sigma}(x)\frac{(x-\mu)^2-\sigma^2}{\sigma^3} && \text{by reordering}
\end{aligned}$$

□

Proof. We now prove Theorem E.1 using Lemma E.1. First we show how to obtain the partial derivative with respect to μ (cf. (E.3)). Then we prove the second equation (E.4) and show how to obtain the partial derivative with respect to σ.

$$\frac{\partial}{\partial\mu} P(\texttt{call\_f}_{[X_0,X_1)}) = \frac{\partial}{\partial\mu} \int_{X_0}^{X_1} \varphi_{\mu;\sigma}(x)\,dx \qquad \text{by (E.1)}$$

The partial derivation can be moved inside the integral since $\varphi_{\mu;\sigma}$ is a smooth function and we integrate it over a finite interval. This step requires X_0 and X_1 to be finite. If they are infinite, that is, in case of an improper integral, we would need to argue differently.

$$= \int_{X_0}^{X_1} \frac{\partial}{\partial\mu} \varphi_{\mu;\sigma}(x)\,dx$$

$$= \int_{X_0}^{X_1} \varphi_{\mu;\sigma}(x) \cdot \frac{x-\mu}{\sigma^2}\,dx \qquad \text{by (E.5)}$$

$$= \int_{X_0}^{X_1} \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right) \cdot \frac{x-\mu}{\sigma^2}\,dx \qquad \text{by (E.2)}$$

$$= \frac{-1}{\sigma^2\sqrt{2\pi}} \int_{X_0}^{X_1} \exp\left(-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right) \cdot \frac{\mu-x}{\sigma}\,dx \qquad \text{by reordering}$$

One can write this integral as $c\int f(g(x)) \cdot g'(x)\,dx$, where $f(x) = \exp(x)$, $g(x) = -\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2$ and $g'(x) = \frac{\mu-x}{\sigma}$. Hence we can apply the *integration by substitution*

method. To this end we have to substitute $u = -\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2$ and $du = \frac{\mu - x}{\sigma}\,dx$.

$$
\begin{aligned}
&= \frac{-1}{\sigma^2\sqrt{2\pi}} \int_{g(X_0)}^{g(X_1)} \exp(u)\,du \\
&= \frac{-1}{\sigma^2\sqrt{2\pi}} \exp(u)\Big|_{u=g(X_0)}^{g(X_1)} && \text{by } c\int \exp(u)\,du = \exp(u) \\
&= \frac{-1}{\sigma^2\sqrt{2\pi}} \exp\left(-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right)\Big|_{x=X_0}^{X_1} && \text{by resubstituting } u \\
&= \frac{-1}{\sigma}\varphi_{\mu;\sigma}(x)\Big|_{x=X_0}^{X_1} && \text{by (E.2)} \\
&= \frac{1}{\sigma}\varphi_{\mu;\sigma}(X_0) - \frac{1}{\sigma}\varphi_{\mu;\sigma}(X_1)\ .
\end{aligned}
$$

The second part of the theorem can be shown as follows

$$
\begin{aligned}
&\frac{\partial}{\partial\sigma} P(\texttt{call\_f}_{[X_0,X_1)}) = \frac{\partial}{\partial\sigma}\int_{X_0}^{X_1} \varphi_{\mu;\sigma}(x)\,dx \\
&= \int_{X_0}^{X_1} \frac{\partial}{\partial\sigma}\varphi_{\mu;\sigma}(x)\,dx && \text{by smoothness of } \varphi \\
&= \int_{X_0}^{X_1} \varphi_{\mu;\sigma}(x)\frac{(x-\mu)^2-\sigma^2}{\sigma^3}\,dx && \text{by (E.6)} \\
&= \int_{X_0}^{X_1} \frac{1}{\sigma\sqrt{2\pi}}\exp\left(-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right)\frac{(x-\mu)^2-\sigma^2}{\sigma^3}\,dx && \text{by (E.2)} \\
&= \int_{X_0}^{X_1} \left[\frac{-1}{\sigma^2\sqrt{2\pi}}\exp\left(-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right)\right. \\
&\qquad \left. + \frac{\mu-x}{\sigma^2\sqrt{2\pi}}\exp\left(-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right)\frac{\mu-x}{\sigma^2}\right]dx && \text{by reordering}
\end{aligned}
$$

We define $u(x) \coloneqq \frac{\mu - x}{\sigma^2\sqrt{2\pi}}$ and $v(x) \coloneqq \exp\left(-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right)$ and substitute.

$$= \int_{X_0}^{X_1} \Big[u'(x) \cdot v(x) + u(x) \cdot v'(x)\Big] dx$$

Using the product rule and the fundamental theorem of calculus we get

$$\begin{aligned}
&= u(x) \cdot v(x)\Big|_{x=X_0}^{X_1} \\
&= \frac{\mu - x}{\sigma^2\sqrt{2\pi}} \exp\left(-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right)\Bigg|_{x=X_0}^{X_1} && \text{by re-substituting} \\
&= \frac{\mu - x}{\sigma}\varphi_{\mu;\sigma}(x)\Big|_{x=X_0}^{X_1} && \text{by (E.2)} \\
&= \frac{\mu - X_1}{\sigma}\varphi_{\mu;\sigma}(X_1) - \frac{\mu - X_0}{\sigma}\varphi_{\mu;\sigma}(X_0) \\
&= \frac{X_0 - \mu}{\sigma}\varphi_{\mu;\sigma}(X_0) - \frac{X_1 - \mu}{\sigma}\varphi_{\mu;\sigma}(X_1) \ . && \text{by reordering}
\end{aligned}$$

This concludes the proof and shows that the partial derivatives can be calculated very efficiently and exact by evaluating the density function at the borders of the interval attached to the auxiliary fact. □

Appendix F

Computing Expected Counts on Interpretation-Restricted Theory

The following theorem shows that the conditional probability of f_n given I calculated in the theory T is equivalent to the probability calculated in $T^r(I)$. This guarantees that subsequent steps of LFI-ProbLog (cf. Chapter 7) yield the correct result.

Theorem F.1. *For all ground probabilistic facts f_n and partial interpretations I_m*

$$E_T[\delta^m_{n,k}|I_m] = \begin{cases} E_{T^r(I_m)}[\delta^m_{n,k}|I_m] & \text{if } f_n \in \mathrm{dep}_T(I_m) \\ p_n & \text{otherwise} \end{cases},$$

where $T^r(I)$ is the interpretation-restricted ProbLog theory of T and p_n is the probability of the fact f_n.

Proof. For the ease of notation we assume that the ProbLog program T is ground. Furthermore, we drop the m index identifying individual interpretations, hence $I_m = I$. We then have to consider the following **two cases**.

Case 1 $f_n \notin dep_T(I)$: Due to the definition of $\mathrm{dep}_T(I)$ the ground probabilistic fact f_n is independent of I and we get $P^T_w(\{f_n\} \mid I) = P^T_w(\{f_n\}) = p_n$.

Case 2 $f_n \in \mathrm{dep}_T(I)$: In order to prove that $E_T[\delta_{n,k}|I] = E_{T^r(I)}[\delta_{n,k}|I]$ it is sufficient to show that $P^T_w(\{f_n\} \mid I) = P^{T^r(I)}_w(\{f_n\} \mid I)$ as this is the probability

used by the expectation computation. In turn, it is sufficient to show that

$$\frac{P_w^T(\{f_n\}, I)}{P_w^T(I)} = \frac{P_w^{T^r(I)}(\{f_n\}, I)}{P_w^{T^r(I)}(I)}$$

due to the definition of a conditional probability. The following argumentation similar for numerator and denominator, we will restrict to the latter one.

$$P_w^T(I) = \sum_{\substack{L \subseteq L^T \\ L \models I}} P^T(L)$$

Now we split the set L into L_1 and L_2 such that L_1 contains all atoms of L, which are in the dependency set of I, and L_2 contains the rest. Let T_1 and T_2 the corresponding theories.

$$= \sum_{L_1 \subseteq L^T \cap \mathrm{dep}_T(I)} \sum_{\substack{L_2 \subseteq L^T \setminus \mathrm{dep}_T(I) \\ L_1 \cup L_2 \models I}} P^{T_1}(L_1) \cdot P^{T_2}(L_2)$$

By definition, L_2 has no influence on provability of I and can in turn be shifted outside of the inner sum.

$$= \sum_{\substack{L_1 \subseteq L^T \cap \mathrm{dep}_T(I) \\ L_1 \models I}} P^{T_1}(L_1) \sum_{L_2 \subseteq L^T \setminus \mathrm{dep}_T(I)} P^{T_2}(L_2)$$

The inner sum is 1 as it iterates over all subsets.

$$= \sum_{\substack{L_1 \subseteq L^T \cap \mathrm{dep}_T(I) \\ L_1 \models I}} P^{T_1}(L_1)$$

$$= P_w^{T^r(I)}(I)$$

□

Appendix G

Kullback-Leibler Divergence Between ProbLog Programs

The Kullback-Leibler divergence $D(P||Q)$ is a non-symmetric measure for the difference of two probability distributions P and Q (cf. [Wasserman, 2003]). It is used in probability theory as well as in information theory where it is also known as information gain. The K-L divergence aggregates the difference of the two distributions on all elements of the outcome space. It is only defined if the support of Q is larger than the one of P, that is, for all i where $P(i) > 0$ also $Q(i) > 0$.

We use the K-L divergence to evaluate the LFI-ProbLog learning algorithm (cf. Chapter 7) and measure how close the learned program T_2 is to the ground truth program T_1. We are doing parameter estimation, that is, the structure of the program is fixed and only the fact probabilities change. Hence we can restrict the definition of the K-L divergence to programs that are identical except for the fact probabilities.

Definition G.1 (K-L Divergence)**.** *Let $T_1 = F_1 \cup BK$ and $T_2 = F_2 \cup BK$ be ground ProbLog programs such that the probabilistic facts are identical except for the probabilities, that is, $F_1 = \{p_i :: f_i | 1 \leq i \leq n\}$ and $F_2 = \{q_i :: f_i | 1 \leq i \leq n\}$. Then the K-L Divergence between T_1 and T_2 is defined as*

$$D(T_1||T_2) := \sum_{L \subseteq L^{T_1}} P^{T_1}(L) \log \frac{P^{T_1}(L)}{P^{T_2}(L)}$$

where $P^{T_1}(L)$ and $P^{T_2}(L)$ are the probabilities that the subprogram L is sampled from T_1 and T_2 respectively (cf. Eq. 3.4).

Please note, that the set of ground facts L^{T_1} in the definition above is identical to the ground facts L^{T_2} (cf. Equation 3.2). Hence the value of $D(T_1||T_2)$ does not change if the sum iterates over $L \subseteq L^{T_2}$ instead of $L \subseteq L^{T_1}$.

There are exponentially many subprograms $L \subseteq L^{T_1}$, which makes evaluating the K-L divergence as defined above impossible in practice. However, the probabilistic facts in a ProbLog program are independent, which can be exploited to compute the K-L divergence in linear time by looping once over F.

Theorem G.1. *Let $T_1 = F_1 \cup BK$ and $T_2 = F_2 \cup BK$ be ground ProbLog programs such that the probabilistic facts are identical except for the probabilities, that is, $F_1 = \{p_i :: f_i | 1 \leq i \leq n\}$ and $F_2 = \{q_i :: f_i | 1 \leq i \leq n\}$. Then the K-L Divergence between T_1 and T_2 can be calculated as*

$$D(T_1||T_2) = \sum_{i=1}^{n} \left(p_i \log \frac{p_i}{q_i} + (1-p_i) \log \frac{1-p_i}{1-q_i} \right) \quad .$$

It is possible to extend the K-L divergence and the theorem to non-ground facts. To do so, one needs to multiply each summand $p_i \log \frac{p_i}{q_i} + (1-p_i) \log \frac{1-p_i}{1-q_i}$ with the number of ground instances of the probabilistic fact f_i.

Proof. We prove Theorem G.1 by induction over the number of probabilistic facts.

Base case $n = 1$.

$$\begin{aligned}
D(T_1||T_2) &= \sum_{L \subseteq L^{T_1}} P^{T_1}(L) \log \frac{P^{T_1}(L)}{P^{T_2}(L)} \\
&= P^{T_1}(\{f_1\}) \log \frac{P^{T_1}(\{f_1\})}{P^{T_2}(\{f_1\})} + P^{T_1}(\varnothing) \log \frac{P^{T_1}(\varnothing)}{P^{T_2}(\varnothing)} \\
&= p_1 \log \frac{p_1}{q_1} + (1-p_1) \log \frac{1-p_1}{1-q_1} \\
&= \sum_{i=1}^{n} \left(p_i \log \frac{p_i}{q_i} + (1-p_i) \log \frac{1-p_i}{1-q_i} \right)
\end{aligned}$$

Inductive case $n \to n{+}1$. To simplify the notation, we define $T_1^{n+1} := T_1 \cup \{p_{n+1} :: f_{n+1}\}$ and $T_2^{n+1} := T_2 \cup \{q_{n+1} :: f_{n+1}\}$

$$D(T_1^{n+1}||T_2^{n+1}) = \sum_{L \subseteq (L^{T_1} \cup \{f_{n+1}\})} P^{T_1^{n+1}}(L) \log \frac{P^{T_1^{n+1}}(L)}{P^{T_2^{n+1}}(L)}$$

$$= \left[\sum_{L \subseteq L^{T_1}} P^{T_1^{n+1}}(L \cup \{f_{n+1}\}) \log \frac{P^{T_1^{n+1}}(L \cup \{f_{n+1}\})}{P^{T_2^{n+1}}(L \cup \{f_{n+1}\})} \right] +$$

$$\left[\sum_{L \subseteq L^{T_1}} P^{T_1^{n+1}}(L) \log \frac{P^{T_1^{n+1}}(L)}{P^{T_2^{n+1}}(L)} \right]$$

Probabilistic facts are independent and thus we can factorize the probabilities

$$= \left[\sum_{L \subseteq L^{T_1}} p_{n+1} \cdot P^{T_1}(L) \log \frac{p_{n+1} \cdot P^{T_1}(L)}{q_{n+1} \cdot P^{T_2}(L)} \right] +$$

$$\left[\sum_{L \subseteq L^{T_1}} (1 - p_{n+1}) \cdot P^{T_1}(L) \log \frac{(1 - p_{n+1}) \cdot P^{T_1}(L)}{(1 - q_{n+1}) \cdot P^{T_2}(L)} \right]$$

using the rules for log and factoring out the constants

$$= p_{n+1} \left[\sum_{L \subseteq L^{T_1}} P^{T_1}(L) \left(\log \frac{p_{n+1}}{q_{n+1}} + \log \frac{P^{T_1}(L)}{P^{T_2}(L)} \right) \right] +$$

$$(1 - p_{n+1}) \left[\sum_{L \subseteq L^{T_1}} P^{T_1}(L) \left(\log \frac{1 - p_{n+1}}{1 - q_{n+1}} + \log \frac{P^{T_1}(L)}{P^{T_2}(L)} \right) \right]$$

expanding the inner sums and factoring out constants

$$
\begin{aligned}
=& p_{n+1}\left(\log\frac{p_{n+1}}{q_{n+1}}\right)\left[\sum_{L\subseteq L^{T_1}} P^{T_1}(L)\right] + \\
& p_{n+1}\left[\sum_{L\subseteq L^{T_1}} P^{T_1}(L)\left(\log\frac{P^{T_1}(L)}{P^{T_2}(L)}\right)\right] + \\
& (1-p_{n+1})\left(\log\frac{1-p_{n+1}}{1-q_{n+1}}\right)\left[\sum_{L\subseteq L^{T_1}} P^{T_1}(L)\right] + \\
& (1-p_{n+1})\left[\sum_{L\subseteq L^{T_1}} P^{T_1}(L)\left(\log\frac{P^{T_1}(L)}{P^{T_2}(L)}\right)\right]
\end{aligned}
$$

both $\sum_{L\subseteq L^{T_1}} P^{T_1}(L)$ and $\sum_{L\subseteq L^{T_1}} P^{T_2}(L)$ are 1, rearranging yields

$$
\begin{aligned}
=& p_{n+1}\left(\log\frac{p_{n+1}}{q_{n+1}}\right) + (1-p_{n+1})\left(\log\frac{1-p_{n+1}}{1-q_{n+1}}\right) + \\
& \sum_{L\subseteq L^{T_1}} P^{T_1}(L)\log\frac{P^{T_1}(L)}{P^{T_2}(L)}
\end{aligned}
$$

using the inductive assumption

$$
\begin{aligned}
=& p_{n+1}\left(\log\frac{p_{n+1}}{q_{n+1}}\right) + (1-p_{n+1})\left(\log\frac{1-p_{n+1}}{1-q_{n+1}}\right) + \\
& \sum_{i=1}^{n}\left(p_i\log\frac{p_i}{q_i} + (1-p_i)\log\frac{1-p_i}{1-q_i}\right)
\end{aligned}
$$

rearranging the terms

$$
= \sum_{i=1}^{n+1}\left(p_i\log\frac{p_i}{q_i} + (1-p_i)\log\frac{1-p_i}{1-q_i}\right)
$$

□

Bibliography

Nicos Angelopoulos and James Cussens. Prolog issues and experimental results of an MCMC algorithm. In Oskar Bartenstein, Ulrich Geske, Markus Hannebauer, and Osamu Yoshie, editors, *Web Knowledge Management and Decision Support*, volume 2543 of *Lecture Notes in Computer Science*, pages 186–196. Springer, Berlin / Heidelberg, 2003. DOI: 10.1007/3-540-36524-9_15.

Laura-Andreea Antanas, Ingo Thon, Martijn van Otterlo, Niels Landwehr, and Luc De Raedt. Probabilistic logical sequence models for video. In *19th International Conference on Inductive Logic Programming (ILP 2009)*, 2009.

Krzysztof R. Apt and Marc Bezem. Acyclic programs. *New Generation Computing*, 9(3/4):335–364, 1991. DOI: 10.1007/BF03037168.

James K. Baker. Trainable grammars for speech recognition. *The Journal of the Acoustical Society of America*, 65(S1):S132–S132, 1979. DOI: 10.1121/1.2017061.

Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. Magic sets and other strange ways to implement logic programs (extended abstract). In *Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems (PODS 1986)*, pages 1–15, Cambridge, Massachusetts, United States, 1986. ACM. ISBN 0-89791-179-2. DOI: 10.1145/6012.15399.

Chitta Baral, Michael Gelfond, and Nelson Rushton. Probabilistic reasoning with answer sets. *Theory and Practice of Logic Programming (TPLP)*, 9(1):57–144, 2009. DOI: 10.1017/S1471068408003645.

Robert G. Bartle. *The elements of integration and Lebesgue measure*. Wiley Classics Library. John Wiley & Sons Inc., 1995. ISBN 0-471-04222-6.

Elena Bellodi and Fabrizio Riguzzi. EM over binary decision diagrams for probabilistic logic programs. Technical Report CS-2011-01, Dipartimento di Ingegneria, Università di Ferrara, Italy, 2011.

Christoper M. Bishop. *Pattern Recognition and Machine Learning*. Springer-Verlag New York, 2006. ISBN 978-0-387-31073-2.

Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986. DOI: 10.1109/TC.1986.1676819.

Randal E. Bryant. On the complexity of VLSI implementations and graph representations of boolean functions with application to integer multiplication. *IEEE Transactions on Computers*, 40:205–213, 1991. DOI: 10.1109/12.73590.

Eugene Charniak. Tree-bank grammars. In *Proceedings of the thirteenth national conference on Artificial intelligence - Volume 2*, pages 1031–1036. AAAI Press, 1996. ISBN 0-262-51091-X.

Jianzhong Chen, Stephen Muggleton, and José Santos. Learning probabilistic logic models from probabilistic examples. *Machine learning*, 73(1):55–85, 2008. DOI: 10.1007/s10994-008-5076-4.

Mark Craven and Seán Slattery. Relational learning with statistical predicate invention: Better models for hypertext. *Machine Learning*, 43(1):97–119, 2001. DOI: 10.1023/A:1007676901476.

James Cussens. Parameter estimation in stochastic logic programs. *Machine Learning*, 44:245–271, 2001. DOI: 10.1023/A:1010924021315.

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

Adnan Darwiche. A logical approach to factoring belief networks. In *Proceedings of the Eighths International Conference on Principles and Knowledge Representation and Reasoning (KR-02)*, pages 409–420, 2002.

Adnan Darwiche. New advances in compiling CNF into decomposable negation normal form. In *ECAI*, pages 328–332, 2004.

Tinne De Laet. *Rigorously Bayesian Multitarget Tracking and Localization*. PhD thesis, Katholieke Universiteit Leuven, May 2010.

Luc De Raedt. *Logical and Relational Learning*. Cognitive Technologies. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-20040-6.

Luc De Raedt and Kristian Kersting. Probabilistic Logic Learning. *ACM-SIGKDD Explorations: Special issue on Multi-Relational Data Mining*, 5(1):31–48, 2003. DOI: 10.1145/959242.959247.

Luc De Raedt and Kristian Kersting. Probabilistic inductive logic programming. In Shoham Ben-David, John Case, and Akira Maruoka, editors, *Algorithmic Learning Theory*, volume 3244 of *LNCS (Lecture Notes in Computer Science)*, pages 19–36. Springer Berlin / Heidelberg, 2004. DOI: 10.1007/978-3-540-30215-5_3.

Luc De Raedt and Ingo Thon. Probabilistic rule learning. In Paolo Frasconi and Francesca Alessandra Lisi, editors, *Proceedings of the 20th International Conference on Inductive Logic Programming (ILP–10)*, volume 6489 of *LNCS (Lecture Notes in Computer Science)*, pages 47–58. Springer Berlin / Heidelberg, 2011. DOI: 10.1007/978-3-642-21295-6_9.

Luc De Raedt, Kristian Kersting, and Sunna Torge. Towards learning stochastic logic programs from proof-banks. In *AAAI*, pages 752–757, 2005.

Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. ProbLog: A probabilistic Prolog and its application in link discovery. In Manuela M. Veloso, editor, *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India*, pages 2462–2467, 2007.

Luc De Raedt, Paolo Frasconi, Kristian Kersting, and Stephen Muggleton, editors. *Probabilistic Inductive Logic Programming — Theory and Applications*, volume 4911 of *Lecture Notes in Artificial Intelligence*. Springer, 2008a.

Luc De Raedt, Kristian Kersting, Angelika Kimmig, Kate Revoredo, and Hannu Toivonen. Compressing probabilistic Prolog programs. *Machine Learning*, 70: 151–168, 2008b. DOI: 10.1007/s10994-007-5030-x.

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

Thomas G. Dietterich, Adam Ashenfelter, and Yaroslav Bulatov. Training conditional random fields via gradient tree boosting. In *Proceedings of the twenty-first international conference on Machine learning (ICML 2004)*, New York, NY, USA, 2004. ACM. DOI: 10.1145/1015330.1015428.

Pedro Domingos and Daniel Lowd. *Markov Logic: An Interface Layer for Artificial Intelligence*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2009. DOI: 10.2200/S00206ED1V01Y200907AIM007.

J. Eisner, E. Goldlust, and N. Smith. Compiling Comp Ling: Weighted dynamic programming and the Dyna language. In *Proceedings of the Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing (HLT/EMNLP-05)*, 2005.

David Ferrucci, Eric Brown, Jennifer Chu-Carroll, James Fan, David Gondek, Aditya A. Kalyanpur, Adam Lally, J. William Murdock, Eric Nyberg, John Prager, Nico Schlaefer, and Chris Welty. Building Watson: An overview of the DeepQA project. *AI Magazine*, 31(3), 2011.

Daan Fierens, Guy Van den Broeck, Ingo Thon, Bernd Gutmann, and Luc De Raedt. Inference in probabilistic logic programs using weighted CNF's. In *Proceedings of the Proceedings of the Twenty-Seventh Conference Annual Conference on Uncertainty in Artificial Intelligence (UAI-11)*, pages 211–220, Corvallis, Oregon, 2011. AUAI Press.

Dann Fierens. On the relationship between logical Bayesian networks and probabilistic logic programming based on the distribution semantics. In Luc De Raedt, editor, *Inductive Logic Programming*, volume 5989 of *LNCS (Lectures Notes in Computer Science)*, pages 17–24. Springer Berlin / Heidelberg, 2010. DOI: 10.1007/978-3-642-13840-9_3.

Peter A. Flach. *Simply Logical: Intelligent Reasoning by Example*. John Wiley, 1994.

Jerome H. Friedman. Greedy function approximation: A gradient boosting machine. *Annals of Statistics*, 29(5):1189–1232, 2001. DOI: 10.1214/aos/1013203451.

Nir Friedman. 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, 1997. Morgan Kaufmann Publishers Inc. ISBN 1-55860-486-3.

Nir Friedman, Lise Getoor, Daphne Koller, and Avi Pfeffer. Learning probabilistic relational models. In Thomas Dean, editor, *IJCAI*, pages 1300–1309. Morgan Kaufmann, 1999a.

Nir Friedman, Lise Getoor, Daphne Koller, and Avi Pfeffer. Learning probabilistic relational models. In Thomas Dean, editor, *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI 99)*, pages 1300–1309. Morgan Kaufmann, 1999b. ISBN 1-55860-613-0.

Norbert Fuhr. Probabilistic Datalog: Implementing logical information retrieval for advanced applications. *Journal of the American Society for Information Science (JASIS)*, 51(2):95–110, 2000.

Lise Getoor and Ben Taskar, editors. *Introduction to Statistical Relational Learning*. The MIT Press, November 2007.

Lise Getoor, Nir Friedman, Daphne Koller, and Avi Pfeffer. Learning probabilistic relational models. In Sašo Džeroski and Nada Lavrač, editors, *Relational Data Mining*, pages 307–335. Springer Verlag, 2001.

Vibhav Gogate and Rina Dechter. SampleSearch: Importance sampling in presence of determinism. *Artificial Intelligence*, 175:694–729, February 2011. DOI: 10.1016/j.artint.2010.10.009.

Noah Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: a language for generative models. In David A. McAllester and Petri Myllymäki, editors, *UAI 2008, Proceedings of the 24th Conference in Uncertainty in Artificial Intelligence, July 9-12, 2008, Helsinki, Finland*, pages 220–229. AUAI Press, 2008. ISBN 0-9749039-4-9.

Rahul Gupta and Sunita Sarawagi. Creating probabilistic databases from information extraction models. In *VLDB*, pages 965–976, 2006.

Bernd Gutmann. Relational conditional random fields. Diplomarbeit, Albert-Ludwigs-Universität Freiburg, Germany, October 2005. In German.

Bernd Gutmann and Kristian Kersting. TildeCRF: Conditional random fields for logical sequences. In Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou, editors, *Proceedings of the 15th European Conference on Machine Learning (ECML-2006)*, volume 4212 of *LNCS (Lecture Notes in Computer Science)*, pages 174–185. Springer, September 2006. DOI: 10.1007/11871842_20.

Bernd Gutmann and Kristian Kersting. Stratified gradient boosting for fast training of conditional random fields. In Donato Malerba, Annalisa Appice, and Michelangelo Ceci, editors, *Proceedings of the 6th International Workshop on Multi-relational Data Mining (MRDM07)*, pages 58–68, Warsaw, Poland, September 2007.

Bernd Gutmann, Angelika Kimmig, Luc De Raedt, and Kristian Kersting. Parameter learning in probabilistic databases: A least squares approach. In Walter Daelemans, Bart Goethals, and Katharina Morik, editors, *Proceedings of the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD 2008)*, volume 5211 of *LNCS (Lecture Notes In Computer Science)*, pages 473–488. Springer Berlin / Heidelberg, September 2008. DOI: 10.1007/978-3-540-87479-9_49.

Bernd Gutmann, Manfred Jaeger, and Luc De Raedt. Extending problog with continuous distributions. In Paolo Frasconi and Francesca Alessandra Lisi, editors, *Proceedings of the 20th International Conference on Inductive Logic Programming (ILP–10)*, volume 6489 of *LNCS (Lecture Notes in Computer Science)*, pages 76–91. Springer Berlin / Heidelberg, 2010a. DOI: 10.1007/978-3-642-21295-6_12.

Bernd Gutmann, Angelika Kimmig, Kristian Kersting, and Luc De Raedt. Parameter estimation in ProbLog from annotated queries. Technical Report CW 583, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, April 2010b.

Bernd Gutmann, Ingo Thon, and Luc De Raedt. Learning the parameters of probabilistic logic programs from interpretations. Technical Report CW 584, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, April 2010c.

Bernd Gutmann, Ingo Thon, and Luc De Raedt. Learning the parameters of probabilistic logic programs from interpretations. In Dimitrios Gunopulos, Thomas Hofmann, Donato Malerba, and Michalis Vazirgiannis, editors, *Machine Learning and Knowledge Discovery in Databases (ECML PKDD 2011)*, volume 6911 of *LNCS (Lecture Notes in Computer Science)*, pages 581–596. Springer Berlin / Heidelberg, 2011a. DOI: 10.1007/978-3-642-23780-5_47.

Bernd Gutmann, Ingo Thon, Angelika Kimmig, Maurice Bruynooghe, and Luc De Raedt. The magic of logical inference in probabilistic programming. *Theory and Practice of Logic Programming*, 11:663–680, 2011b. DOI: 10.1017/S1471068411000238.

John Hopcroft and Robert Tarjan. Efficient algorithms for graph manipulation. *Communications of the ACM*, 16:372–378, June 1973. DOI: 10.1145/362248.362272.

Masakazu Ishihata, Yoshitaka Kameya, Taisuke Sato, and Shin ichi Minato. Propositionalizing the EM algorithm by BDDs. In Filip Železný and Nada Lavrač, editors, *Proceedings of Inductive Logic Programming (ILP 2008), Late Breaking Papers*, pages 44–49, Prague, Czech Republic, September 2008.

Manfred Jaeger. Relational Bayesian networks. In Dan Geiger and Prakash P. Shenoy, editors, *UAI*, pages 266–273. Morgan Kaufmann, 1997.

Manfred Jaeger. Parameter learning for relational Bayesian networks. In Zoubin Ghahramani, editor, *ICML*, volume 227 of *ACM International Conference Proceeding Series*, pages 369–376. ACM, 2007. ISBN 978-1-59593-793-3. DOI: 10.1145/1273496.1273543.

Tomi Janhunen. Representing normal programs with clauses. In Ramon López de Mántaras and Lorenza Saitta, editors, *Proceedings of the 16th European Conference on Artificial Intelligence*, pages 358–362. IOS Press, August 2004.

Finn Verner Jensen. *Bayesian Networks and Decision Graphs*. Springer, 2001.

David B. Kemp, Divesh Srivastava, and Peter J. Stuckey. Bottom-up evaluation and query optimization of well-founded models. *Theoretical Computer Science*, 146:145–184, July 1995. DOI: 10.1016/0304-3975(94)00153-A.

Kristian Kersting. *An Inductive Logic Programming Approach to Statistical Relational Learning*. PhD thesis, Albert-Ludwigs-Universiät Freiburg, April 2006.

Kristian Kersting and Luc De Raedt. Bayesian logic programming: theory and tool. In Lise Getoor and Ben Taskar, editors, *Introduction to Statistical Relational Learning*. The MIT Press, November 2007.

Kristian Kersting and Luc De Raedt. Basic principles of learning Bayesian logic programs. In Luc De Raedt, Paolo Frasconi, Kristian Kersting, and Stephen Muggleton, editors, *Probabilistic Inductive Logic Programming - Theory and Applications*, volume 4911 of *Lecture Notes in Computer Science*, pages 189–221. Springer Berlin / Heidelberg, 2008. ISBN 978-3-540-78651-1. DOI: 10.1007/978-3-540-78652-8_7.

Kristian Kersting and Bernd Gutmann. Unbiased conjugate direction boosting for conditional random fields. In Thomas Gärtner, Gemma Casas Garriga, and Thorsten Meinl, editors, *Working Notes of the ECML-2006 Workshop on Mining and Learning with Graphs (MLG 2006)*, pages 157–164, Berlin, Germany, September 2006. Short paper.

Angelika Kimmig. *A Probabilistic Prolog and its Applications*. PhD thesis, Katholieke Universiteit Leuven, November 2010.

Angelika Kimmig, Vítor Santos Costa, Ricardo Rocha, Bart Demoen, and Luc De Raedt. On the efficient execution of ProbLog programs. In Maria Garcia de la Banda and Enrico Pontelli, editors, *Logic Programming*, volume 5366 of *Lecture Notes in Computer Science*, pages 175–189. Springer Berlin / Heidelberg, 2008. DOI: 10.1007/978-3-540-89982-2_22.

Angelika Kimmig, Bart Demoen, Luc De Raedt, Vítor Santos Costa, and Ricardo Rocha. On the implementation of the probabilistic logic programming language ProbLog. *Theory and Practice of Logic Programming (TPLP)*, 11:235–262, 2011. DOI: 10.1017/S1471068410000566.

Daphne Koller and Nir Friedman. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009.

Chee-Keong Kwoh and Duncan Fyfe Gillies. Using hidden nodes in Bayesian networks. *Artificial Intelligence*, 88(1-2):1–38, 1996. DOI: 10.1016/0004-3702(95)00119-0.

Niels Landwehr, Bernd Gutmann, Ingo Thon, Luc De Raedt, and Matthai Philipose. Relational transformation-based tagging for activity recognition. *Fundamenta Informaticae*, 89(1):111–129, 2008.

John W. Lloyd. *Foundations of logic programming*. Springer-Verlag New York, Inc., New York, NY, USA, 2nd edition, 1984. ISBN 0-387-13299-6.

Daniel Lowd and Pedro Domingos. Efficient weight learning for Markov logic networks. In Joost N. Kok, Jacek Koronacki, Ramon López de Mántaras, Stan Matwin, Dunja Mladenic, and Andrzej Skowron, editors, *Knowledge Discovery in Databases: PKDD 2007*, volume 4702 of *Lecture Notes in Computer Science*, pages 200–211. Springer Berlin / Heidelberg, 2007. DOI: 10.1007/978-3-540-74976-9_21.

Theofrastos Mantadelis and Gerda Janssens. Dedicated tabling for a probabilistic setting. In Manuel V. Hermenegildo and Torsten Schaub, editors, *Technical Communications of the 26th International Conference on Logic Programming (ICLP-10)*, volume 7 of *LIPIcs*, pages 124–133. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2010. ISBN 978-3-939897-17-0. DOI: 10.4230/LIPIcs.ICLP.2010.124.

Theofrastos Mantadelis, Bart Demoen, and Gerda Janssens. A simplified fast interface for the use of CUDD for binary decision diagrams, 2008. URL https://lirias.kuleuven.be/handle/123456789/253405.

George Marsaglia. Evaluating the normal distribution. *Journal of Statistical Software*, 11(5):1–11, 7 2004.

Wannes Meert, Jan Struyf, and Hendrik Blockeel. Learning ground CP-Logic theories by leveraging Bayesian network learning techniques. *Fundamenta Informaticae*, 89(1):131–160, 2008.

Brian Milch, Bhaskara Marthi, Stuart J. Russell, David Sontag, Daniel L. Ong, and Andrey Kolobov. BLOG: Probabilistic models with unknown objects. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30-August 5, 2005*, pages 1352–1359. Professional Book Center, 2005a. ISBN 0938075934.

Brian Milch, Bhaskara Marthi, David Sontag, Stuart Russell, Daniel L. Ong, and Andrey Kolobov. Approximate inference for infinite contingent Bayesian networks. In Robert G. Cowell and Zoubin Ghahramani, editors, *Proceedings of the Tenth International Workshop on Artificial Intelligence and Statistics, Jan 6-8, 2005, Savannah Hotel, Barbados*, pages 238–245. Society for Artificial Intelligence and Statistics, 2005b.

Brian Milch, Luke S. Zettlemoyer, Kristian Kersting, Michael Haimes, and Leslie Pack Kaelbling. Lifted probabilistic inference with counting formulas. In Dieter Fox and Carla P. Gomes, editors, *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI 2008)*, pages 1062–1068. AAAI Press, 2008. ISBN 978-1-57735-368-3.

Tom M. Mitchell. *Machine Learnning*. McGraw-Hill, 1997.

Stephen Muggleton. Stochastic logic programs. In Luc De Raedt, editor, *Advances in Inductive Logic Programming*, volume 32 of *Frontiers in Artificial Intelligence and Applications*, pages 254–264. IOS Press, 1996.

Kevin Murphy. Inference and learning in hybrid Bayesian networks. Technical Report UCB/CSD-98-990, University of Berkeley, 1998.

Sriraam Natarajan, Tushar Khot, Kristian Kersting, Bernd Gutmann, and Jude Shavlik. Gradient-based boosting for statistical relational learning: The relational dependency network case. *Machine Learning*, 2011. DOI: 10.1007/s10994-011-5244-9. To appear.

Jennifer Neville and David Jensen. Relational dependency networks. *Journal of Machine Learning Research*, 8:653–692, May 2007.

Ulf Nilsson and Jan Małuszyński. *Logic, Programming and Prolog*. John Wiley & Sons Ltd., 2nd edition, 1995.

Henrik Nottelmann and Norbert Fuhr. Learning probabilistic datalog rules for information classification and transformation. In *Proceedings of the tenth international conference on Information and knowledge management*, CIKM '01, pages 387–394, New York, NY, USA, 2001. ACM. ISBN 1-58113-436-3. DOI: 10.1145/502585.502651.

Naoaki Okazaki. libLBFGS: a library of limited-memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS), 2007. URL http://www.chokkan.org/software/liblbfgs/.

Judea Pearl. *Probabilistic Reasoning in Intelligent Systems*. Morgan Kaufmann, 1988.

Avi Pfeffer. IBAL: A probabilistic rational programming language. In Bernhard Nebel, editor, *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (ICJAI 2001)*, pages 733–740. Morgan Kaufmann Publishers, 2001. ISBN 1-55860-777-3.

Avi Pfeffer. Figaro: An object-oriented probabilistic programming language. Technical report, Charles River Analytics, Cambridge, MA 02138 USA, 2009.

David Poole. Logic programming, abduction and probability - a top-down anytime algorithm for estimating prior and posterior probabilities. *New Generation Computing*, 11(3-4):377–400, 1993. DOI: 10.1007/BF03037184.

David Poole. First-order probabilistic inference. In *Proceedings of the 18th international joint conference on Artificial intelligence (IJCAI 2003)*, pages 985–991. Morgan Kaufmann Publishers Inc., 2003.

David Poole. The independent choice logic and beyond. In Luc De Raedt, Paolo Frasconi, Kristian Kersting, and Stephen Muggleton, editors, *Probabilistic Inductive Logic Programming - Theory and Applications*, volume 4911 of *Lecture Notes in Computer Science (LNCS)*, pages 222–243. Springer, 2008. ISBN 978-3-540-78651-1. DOI: 10.1007/978-3-540-78652-8_8.

Lawrence R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77:257–286, February 1989. DOI: 10.1109/5.18626.

Matthew Richardson and Pedro Domingos. Markov logic networks. *Machine Learning*, 62:107–136, 2006. DOI: 10.1007/s10994-006-5833-1.

Fabrizio Riguzzi. Learning ground ProbLog programs from interpretations. In Donato Malerba, Annalisa Appice, and Michelangelo Ceci, editors, *Proceedings of the 6th International Workshop on Multi-relational Data Mining (MRDM07)*, pages 105–116, Warsaw, Poland, September 2007.

Fabrizio Riguzzi. ALLPAD: Approximate learning of logic programs with annotated disjunctions. *Machine Learning*, 70:207–223, March 2008. DOI: 10.1007/s10994-007-5032-8.

J. B. Rosen. The gradient projection method for nonlinear programming. part i. linear constraints. *Journal of the Society for Industrial and Applied Mathematics*, 8(1):181–217, 1960. DOI: 10.1137/0108011.

J. B. Rosen. The gradient projection method for nonlinear programming. part ii. nonlinear constraints. *Journal of the Society for Industrial and Applied Mathematics*, 9(4):514–532, 1961. DOI: 10.1137/0109044.

Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition, 2003.

Vítor Santos Costa, David Page, and James Cussens. CLP($\mathcal{BN}$): Constraint logic programming for probabilistic knowledge. In Luc De Raedt, Paolo Frasconi, Kristian Kersting, and Stephen Muggleton, editors, *Probabilistic Inductive Logic Programming - Theory and Applications*, volume 4911 of *Lecture Notes in Computer Science*, pages 156–188. Springer Berlin / Heidelberg, 2008. ISBN 978-3-540-78651-1. DOI: 10.1007/978-3-540-78652-8.

Vítor Santos Costa, Ricardo Rocha, and Luís Damas. The YAP Prolog System. *Journal of Theory and Practice of Logic Programming*, 2011. To appear, arXiv:1102.3896v1 [cs.PL].

Taisuke Sato. A statistical learning method for logic programs with distribution semantics. In Leon Sterling, editor, *Proceedings of the Twelfth International Conference on Logic Programming (ICLP 1995)*, pages 715–729. MIT Press, 1995. ISBN 0-262-69177-9.

Taisuke Sato and Yoshitaka Kameya. Parameter learning of logic programs for symbolic-statistical modeling. *Journal of Artificial Intelligence Research*, 15: 391–454, 2001. DOI: 10.1613/jair.912.

Petteri Sevon, Lauri Eronen, Petteri Hintsanen, Kimmo Kulovesi, and Hannu Toivonen. Link discovery in graphs derived from biological databases. In Ulf Leser, Felix Naumann, and Barbara A. Eckman, editors, *Data Integration in the Life Sciences*, volume 4075 of *Lecture Notes in Computer Science (LNCS)*, pages 35–49. Springer Berlin / Heidelberg, 2006. DOI: 10.1007/11799511_5.

Claude Elwood Shannon. The synthesis of two-terminal switching circuits. *Bell Systems Technical Journal*, 28:59–98, 1948.

Dimitar Sht. Shterionov, Angelika Kimmig, Theofrastos Mantadelis, and Gerda Janssens. DNF sampling for ProbLog inference. *CoRR*, abs/1009.3798, 2010.

Parag Singla and Pedro Domingos. Entity resolution with markov logic. In *Proceedings of the Sixth International Conference on Data Mining (ICDM'06)*, pages 572–582, 2006. DOI: 10.1109/ICDM.2006.65.

Alex J. Smola and Bernhard Schölkopf. A tutorial on support vector regression. *Statistics and Computing*, 14:199–222, 2004. DOI: 10.1023/B:STCO.0000035301.49549.88.

Fabio Somenzi. CUDD: CU Decision Diagram package release 2.4.2, 2009. URL http://vlsi.colorado.edu/~fabio/CUDD/.

Leon Sterling and Ehud Shapiro. *The art of Prolog*. MIT Press, Cambridge, MA, USA, 2nd edition, 1994. ISBN 0-262-19338-8.

Ingo Thon, Niels Landwehr, and Luc De Raedt. A simple model for sequences of relational state descriptions. In Walter Daelemans, Bart Goethals, and Katharina Morik, editors, *Machine Learning and Knowledge Discovery in Databases*, volume 5212 of *Lecture Notes in Computer Science (LNCS)*, pages 506–521. Springer Berlin / Heidelberg, 2008. DOI: 10.1007/978-3-540-87481-2_33.

Ingo Thon, Niels Landwehr, and Luc De Raedt. Stochastic relational processes: Efficient inference and applications. *Machine Learning*, 82:239–272, 2011. DOI: 10.1007/s10994-010-5213-8.

Sebastian Thrun, Mike Montemerlo, Hendrik Dahlkamp, David Stavens, Andrei Aron, James Diebel, Philip Fong, John Gale, Morgan Halpenny, Gabriel Hoffmann, Kenny Lau, Celia Oakley, Mark Palatucci, Vaughan Pratt, Pascal Stang, Sven Strohband, Cedric Dupont, Lars-Erik Jendrossek, Christian Koelen, Charles Markey, Carlo Rummel, Joe van Niekerk, Eric Jensen, Philippe Alessandrini, Gary Bradski, Bob Davies, Scott Ettinger, Adrian Kaehler, Ara Nefian, and Pamela Mahoney. Stanley: The robot that won the DARPA Grand Challenge. *Journal of Field Robotics*, 23(9):661–692, 2006. DOI: 10.1002/rob.20147.

Sebastion Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics*. The MIT Press, September 2005.

Leslie G. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8(3):410–421, 1979. DOI: 10.1137/0208032.

Joost Vennekens, Marc Denecker, and Maurice Bruynooghe. Representing causal information about a probabilistic process. In Michael Fisher, Wiebe van der Hoek, Boris Konev, and Alexei Lisitsa, editors, *Logics in Artificial Intelligence, 10th European Conference, (JELIA 2006), Liverpool, UK*, volume 4160 of *Lecture Notes in Computer Science*, pages 452–464. Springer, 2006. ISBN 3-540-39625-X. DOI: 10.1007/11853886_37.

Joost Vennekens, Marc Denecker, and Maurice Bruynooghe. CP-logic: A language of causal probabilistic events and its relation to logic programming. *Theory and Practice of Logic Programming*, 9(3):245–308, 2009. DOI: 10.1017/S1471068409003767.

Jue Wang and Pedro Domingos. Hybrid Markov logic networks. In Dieter Fox and Carla P. Gomes, editors, *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, pages 1106–1111. AAAI Press, 2008. ISBN 978-1-57735-368-3.

Larry Wasserman. *All of Statistics: A Concise Course in Statistical Inference (Springer Texts in Statistics)*. Springer, December 2003. ISBN 0387402721.

Stefan Wrobel, Dietrich Wettschereck, Edgar Sommer, and Werner Emde. Extensibility in data mining systems. In *KDD*, pages 214–219, 1996.

Publication List

Journal Articles

- Sriraam Natarajan, Tushar Khot, Kristian Kersting, Bernd Gutmann and Jude Shavlik. *Gradient-based Boosting for Statistical Relational Learning: The Relational Dependency Network Case*, Invited contribution to special issue of Machine Learning Journal (MLJ), to appear.
  (Springer Online First, DOI: 10.1007/s10994-011-5244-9)

- Bernd Gutmann, Ingo Thon, Angelika Kimmig, Maurice Bruynooghe, and Luc De Raedt. *The magic of logical inference in probabilistic programming.* Theory and Practice of Logic Programming, 11:663–680, 2011. DOI: 10.1017/S1471068411000238

- Niels Landwehr, Bernd Gutmann, Ingo Thon, Luc De Raedt, and Matthai Philipose. *Relational transformation-based tagging for activity recognition.* Fundamenta Informaticae, 89(1):111–129, 2008.

Conference Papers

- Bernd Gutmann, Ingo Thon, and Luc De Raedt. *Learning the Parameters of Probabilistic Logic Programs from Interpretations.* In Dimitrios Gunopulos, Thomas Hofmann, Donato Malerba, and Michalis Vazirgiannis, *European Conference on Machine Learning and Principles and Practices of Knowledge Discovery in Databases (ECML PKDD 2011)*, volume 6911 of LNCS (Lecture Notes in Computer Science), pages 581–596. Springer Berlin/Heidelberg, 2011. **Winner of the Best Paper Runner up Award in Machine Learning** (599 submissions). DOI: 10.1007/978-3-642-23780-5_47

- Daan Fierens, Guy Van den Broeck, Ingo Thon, Bernd Gutmann, and Luc De Raedt. *Inference in Probabilistic Logic Programs using Weighted CNF's.*

In *Proceedings of the Proceedings of the Twenty-Seventh Conference Annual Conference on Uncertainty in Artificial Intelligence (UAI-11)*, pages 211–220. AUAI Press, Corvallis, Oregon, 2011.

- Bernd Gutmann, Manfred Jaeger, and Luc De Raedt. *Extending ProbLog with continuous distributions.* In Paolo Frasconi and Francesca Alessandra Lisi, editors, *Proceedings of the 20th International Conference on Inductive Logic Programming (ILP-10)*, volume 6489 of LNCS (Lecture Notes in Computer Science), pages 76–91. Springer Berlin / Heidelberg, 2011. DOI: 10.1007/978-3-642-21295-6_12

- Maurice Bruynooghe, Theofrastos Mantadelis, Angelika Kimmig, Bernd Gutmann, Joost Vennekens, Gerda Janssens, and Luc De Raedt. *ProbLog technology for inference in a probabilistic first order logic.* In Helder Coelho, Rudi Studer, and Michael Wooldridge, *ECAI 2010 - 19th European Conference on Artificial Intelligence*, volume 215 of Frontiers in Artificial Intelligence and Applications, pages 719–724. IOS Press, 2010. DOI: 10.3233/978-1-60750-606-5-719

- Bernd Gutmann, Angelika Kimmig, Kristian Kersting, and Luc De Raedt. *Parameter learning in probabilistic databases: A least squares approach.* In Walter Daelemans, Bart Goethals, and Katharina Morik, *Proceedings of the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD 2008)*, volume 5211 of LNCS (Lecture Notes In Computer Science), pages 473–488, September 2008. Springer Berlin/Heidelberg. DOI: 10.1007/978-3-540-87479-9_49

- Bernd Gutmann and Kristian Kersting. *TildeCRF: Conditional random fields for logical sequences.* In Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou, *Proceedings of the 15th European Conference on Machine Learning (ECML-2006)*, volume 4212 of LNCS (Lecture Notes in Computer Science), pages 174–185. Springer Berlin / Heidelberg, 2006. **Winner of the ECML Best Student Paper Award** (564 submissions). DOI: 10.1007/11871842_20

Workshop Papers

- Ingo Thon, Bernd Gutmann, Guy Van den Broeck. *Probabilistic programming for planning problems.* In Kristian Kersting, Stuart Russell, Leslie Pack Kaelbling, Alon Halevy, Sriraam Natarajan, Lilyana Mihalkova, editors, *Statistical Relational AI workshop*, Atlanta, USA, 12 July 2010.

- Laura-Andreea Antanas, Bernd Gutmann, Ingo Thon, Kristian Kersting, Luc De Raedt. *Combining video and sequential statistical relational techniques to*

monitor card games. In Christian Thurau, Kurt Diressens, and Olana Missura, editors, *Proceedings of the ICML 2010 Workshop on Machine Learning and Games*, Haifa, Israel, 25 June 2010.

- Laura-Andreea Antanas, Bernd Gutmann, Ingo Thon, Kristian Kersting, and Luc De Raedt. *Combining video and sequential statistical relational techniques to monitor card games*. In Jan Ramon, Celine Vens, Kurt Driessens, Martijn Van Otterlo, and Joaquin Vanschoren, editors, *The annual machine learning conference of Belgium and The Netherlands (BeneLearn 2010)*, Leuven, Belgium, 27-28 May 2010.

- Ingo Thon, Bernd Gutmann, Martijn van Otterlo, Niels Landwehr, Luc De Raedt. *From non-deterministic to probabilistic planning with the help of statistical relational learning*, *ICAPS 2009 - Workshop on Planning and Learning*, pages 23–30, Thessaloniki, 20 September 2009.

- Joost Vennekens, Angelika Kimmig, Theofrastos Mantadelis, Bernd Gutmann, Maurice Bruynooghe, and Luc De Raedt. *From ProbLog to first order logic: A first exploration*. In Pedro Domingos and Kristian Kersting, editors, *International Workshop on Statistical Relational Learning (SRL-2009)*, Leuven, Belgium, 2-4 July 2009.

- Angelika Kimmig, Bernd Gutmann, and Vìtor Santos Costa. *Trading memory for answers: Towards tabling ProbLog*. In Pedro Domingos and Kristian Kersting, editors, *International Workshop on Statistical Relational Learning (SRL-2009)*, Leuven, Belgium, 2-4 July 2009.

- Maurice Bruynooghe, Broes De Cat, Jochen Drijkoningen, Daan Fierens, Jan Goos, Bernd Gutmann, Angelika Kimmig, Wouter Labeeuw, Steven Langenaken, Niels Landwehr, Wannes Meert, Ewoud Nuyts, Robin Pellegrims, Roel Rymenants, Stefan Segers, Ingo Thon, Jelle Van Eyck, Guy Van den Broeck, Tine Vangansewinkel, Lucie Van Hove, Joost Vennekens, Timmy Weytjens, and Luc De Raedt. *An exercise with statistical relational learning systems*. In Pedro Domingos and Kristian Kersting, editors, *International Workshop on Statistical Relational Learning (SRL-2009)*, Leuven, Belgium, 2-4 July 2009.

- Luc De Raedt, Bart Demoen, Daan Fierens, Bernd Gutmann, Gerda Janssens, Angelika Kimmig, Niels Landwehr, Theofrastos Mantadelis, Wannes Meert, Ricardo Rocha, Vìtor Santos Costa, Ingo Thon, and Joost Vennekens. *Towards digesting the alphabet-soup of statistical relational learning*. In Daniel Roy, John Winn, David McAllester, Vikash Mansinghka, and Joshua Tenenbaum, editors, *Proceedings of the 1st Workshop on Probabilistic Programming: Universal Languages, Systems and Applications*, Whistler, Canada, December 2008.

- Bernd Gutmann, Angelika Kimmig, Luc De Raedt, and Kristian Kersting. *Estimating the parameters of probabilistic databases from probabilistically weighted queries and proofs [extended abstract].* In Filip Železný and Nada Lavrač, editors, *Proceedings of the 18th International Conference on Inductive Logic Programming (ILP-2008), Late Breaking Papers*, pages 38–43, Prague, Czech Republic, September 2008.

- Bernd Gutmann, Angelika Kimmig, Luc De Raedt, and Kristian Kersting. *Parameter learning in probabilistic databases: A least squares approach.* In Samuel Kaski, S V N Vishwanathan, and Stefan Wrobel, editors, *Proceedings of the 6th International Workshop on Mining and Learning with Graphs (MLG 2008)*, Helsinki, Finland, 4-5 July 2008.

- Bernd Gutmann, Angelika Kimmig, Luc De Raedt, and Kristian Kersting. *Parameter learning in probabilistic databases: A least squares approach.* In Louis Wehenkel, Pierre Geurts, and Raphaël Marée, editors, *The annual machine learning conference of Belgium and The Netherlands (BeneLearn 2008)*, pages 25–26, Spa, Belgium, May 2008.

- Bernd Gutmann and Kristian Kersting. *Stratified gradient boosting for fast training of conditional random fields.* In Donato Malerba, Annalisa Appice, and Michelangelo Ceci, editors, *Proceedings of the 6th International Workshop on Multi-relational Data Mining (MRDM07)*, pages 58–68, Warsaw, Poland, September 2007.

- Bernd Gutmann and Kristian Kersting. *Stratified gradient boosting for fast training of CRFs.* In Paolo Frasconi, Kristian Kersting, and Koji Tsuda, editors,*Proceedings of the 5th International Workshop on Mining and Learning with Graphs (MLG 2007)*, pages 131–134, Florence, Italy, August 2007. Extended Abstract.

- Niels Landwehr, Bernd Gutmann, Ingo Thon, Matthai Philipose, and Luc De Raedt. *Relational transformation-based tagging for human activity recognition.* In João Gama, Mohamed Medhat Gaber, and Jesús Aguilar-Ruiz, editors, *Proceedings of the International Workshop on Knowledge Discovery from Ubiquitous Data Streams (IWKDUS07)*, pages 83–94, Warsaw, Poland, September 2007.

- Andreas Karwath, Bernd Gutmann, and Christian Borst. *On the Use of Types for TildeCRF.* In *The 31st Annual Conference of the German Classification Society (GfKl) on Data Analysis, Machine Learning and Applications*, Freiburg, Germany, March 7-9, 2007.

- Kristian Kersting and Bernd Gutmann. *Unbiased conjugate direction boosting for conditional random fields.* In Thomas Gärtner, Gemma Casas Garriga, and Thorsten Meinl, editors, *Working Notes of the ECML-2006 Workshop*

on Mining and Learning with Graphs (MLG 2006), pages 157–164, Berlin, Germany, September 2006.

Book Chapters

- Luc De Raedt, Angelika Kimmig, Bernd Gutmann, Kristian Kersting, Vìtor Santos Costa, and Hannu Toivonen. *Probabilistic inductive querying using ProbLog.* In Sašo Džeroski, Bart Goethals, and Panče Panov, editors, *Inductive Databases and Constraint-Based Data Mining*, pages 229–262. Springer New York, 2010. DOI: 10.1007/978-1-4419-7738-0_10

- Kristian Kersting, Luc De Raedt, Bernd Gutmann, Andreas Karwath, and Niels Landwehr. *Relational sequence learning.* In Luc De Raedt, Paolo Frasconi, Kristian Kersting, and Stephen Muggleton, editors, *Probabilistic Inductive Logic Programming*, volume 4911 of LNCS (Lecture Notes in Computer Science), pages 28–55. Springer Berlin / Heidelberg, 2008. DOI: 10.1007/978-3-540-78652-8_2

Technical Reports

- Daan Fierens, Guy Van den Broeck, Ingo Thon, Bernd Gutmann, and Luc De Raedt. *Inference in Probabilistic Logic Programs using Weighted CNF's.* Technical Report CW 607, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, June 2011.

- Bernd Gutmann, Ingo Thon, and Luc De Raedt. *Learning the parameters of probabilistic logic programs from interpretations.* Technical Report CW 584, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, April 2010.

- Bernd Gutmann, Angelika Kimmig, Kristian Kersting, and Luc De Raedt. *Parameter estimation in ProbLog from annotated queries.* Technical Report CW 583, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, April 2010.

- Luc De Raedt, Angelika Kimmig, Kristian Kersting, Vìtor Santos Costa, and Hannu Toivonen. *Probabilistic inductive querying using ProbLog.* Technical Report CW 552, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, June 2009.

Curriculum vitae

Bernd Gutmann was born on November 25, 1979 in Freiburg im Breisgau, Germany. He went to school at the Kastelbergschule Waldkirch, Realschule Kollnau and Wirtschaftsgymnasium Emmendingen, from where he graduated with "Allgemeine Hochschulreife" in June 1999. After a year of military service in the German Army, he started studying computer science with mathematics as minor at the Albert-Ludwigs-Universität Freiburg, Germany. He obtained the "Vordiplom" in October 2002 and graduated as "Diplom-Informatiker" in November 2005. He then joined the machine learning group at the same university, headed by Luc De Raedt, where he worked in the area of probabilistic logic learning. From June to September 2006 he worked as an intern in Matthai Philipose's human activity recognition group at Intel Research Seattle. In November 2006 he moved to Belgium – together with Luc De Raedt's machine learning group – to work in the DTAI group (Declarative Talen en Artificiële Intelligentie) at the Katholieke Universiteit Leuven. From October 2007 his research has been funded by a personal grant from the Research Foundation Flanders (FWO Vlaanderen). In October 2011 he will defend his Ph.D. thesis "On Continuous Distributions and Parameter Estimation in Probabilistic Logic Programs".

Arenberg Doctoral School of Science, Engineering & Technology
Faculty of Engineering
Department of Computer Science
Research Group Declarative Languages and Artificial Intelligence
Celestijnenlaan 200A
3001 Heverlee, Belgium