UNIVERSITÀ DEGLI STUDI DI FERRARA

FACOLTÀ DI INGEGNERIA

DOTTORATO DI RICERCA IN SCIENZE DELL'INGEGNERIA

Ciclo XXXII

COORDINATORE Prof. Stefano Trillo

SETTORE SCIENTIFICO DISCIPLINARE ING-INF/05

Scalable Probabilistic Inductive Logic Programming for Big Data

Dottorando

Dott. Arnaud Nguembang Fadja

Tutore

Prof. Fabrizio Riguzzi

Correlatore

Prof.ssa Evelina Lamma

Anni 2016/2019

Abstract

The size of the data available on the Internet and in various fields is constantly increasing. This lead to the phenomenon called *big data*. Since these data come from different sources and are heterogeneous, they are intrinsically characterized by incompleteness and/or uncertainty. In order to manage such data, systems have to be not only able to represent uncertainty but also scalable enough to deal with ever-increasing data. Systems based on logic provide powerful tools for representing, reasoning and learning from data characterized by uncertainty.

The Distributions Semantics (DS) is a well known formalism for representing data characterized by uncertainty. It provides a powerful tool for combining logic and probability. Programs following this formalism are called Probabilistic Logic Programs (PLPs). Many languages follow the DS, such as Logic Program with Annotated Disjunctions (LPADs) and ProbLog. Moreover, state of the art systems for learning either the parameters, EMBLEM and LFI-ProbLog, or the structure, SLIPCOVER and ProbFOIL+, have been implemented. These systems provide good performance in terms of accuracy but still suffer from scalability problems. In this thesis, we address these problems by proposing two languages under the DS in which reasoning and learning are cheaper.

We first present a language, Liftable Probabilistic Logic Programs (LPLPs), in which inference is performed at a lifted level, i.e groups of individuals are considered as a whole instead of one by one. Then we propose the algorithm, LIFTCOVER for LIFTed slipCOVER, that learns the structure of LPLPs from data. Two versions of LIFTCOVER are proposed: the first, LIFTCOVER-EM, uses the Expectation Maximization (EM) algorithm as a sub-routine for parameter learning and the second, LIFTCOVER-LBFGS, uses an optimization method called Limited memory BFGS.

In the second part we present an extension of the language of LPLPs, called Hierarchical Probabilistic Logic Programs (HPLPs), in which clauses and predicates are hierarchically organized. A program in this language can be converted to a set of Arithmetic Circuits (ACs) or deep neural networks and inference is done by evaluating the ACs. We describe how to perform inference and learning in HPLPs. To learn the parameters of HPLPs from data, we propose the algorithm, Parameter learning for HIerarchical probabilistic Logic programs (PHIL). Two variants of PHIL, Deep PHIL (DPHIL) and EM PHIL (EMPHIL), and their regularized versions are presented. We also propose an algorithm, SLEAHP for Structure LEArning of Hierarchical Probabilistic logic programming, for learning both the structure and the parameters of HPLPs from data.

All these algorithms were tested on real world problems and their performances, in terms of time, were better than the state of the art while obtaining solutions of comparable quality.

Sommario

La dimensione dei dati disponibili su Internet e in vari campi è in costante aumento. Questo porta al fenomeno cosiddetto *big data*. Poiché questi dati provengono da fonti diverse e sono eterogenei, sono intrinsecamente caratterizzati da incompletezza e/o incertezza. Per gestire tali dati, i sistemi devono essere non solo in grado di rappresentare l'incertezza, ma anche abbastanza scalabili per gestire dati in costante aumento. I sistemi basati sulla logica forniscono potenti strumenti per rappresentare, ragionare e apprendere dai dati caratterizzati da incertezza. La Semantica distribuzionale (SD) è un formalismo ben noto per rappresentare dati caratterizzati da incertezza. Fornisce un potente strumento per combinare logica e probabilità. Programmi che seguono questo formalismo sono chiamati Programmi Logici Probabilistici (PLP). Molti linguaggi seguono la SD, come i Programmi Logici con Disgiunzioni Annotate (PLDA) e ProbLog. Inoltre, sono stati implementati sistemi all'avanguardia per l'apprendimento dei parametri, EMBLEM e LFI-ProbLog, e della struttura, SLIPCOVER e ProbFOIL+ di programmi in tali linguaggi. Questi sistemi offrono buone prestazioni in termini di accuratezza ma soffrono ancora di problemi di scalabilità. In questa tesi, affrontiamo questi problemi proponendo due linguaggi sotto la SD in cui il ragionamento e l'apprendimento sono veloci.

Innanzitutto proponiamo un linguaggio, chiamato Liftable Probabilistic Logic Programs (LPLP), in cui l'inferenza è eseguita ad alto livello, cioè gruppi di individui sono considerati complessivamente anziché uno per uno. Poi proponiamo l'algoritmo, LIFTCOVER per LIFTed slipCOVER, che apprende la struttura di LPLP dai dati. Due versioni di LIFTCOVER sono state proposte: il primo, LIFTCOVER-EM, utilizza l'algoritmo Expectation Maximization (EM) come sub-routine per l'apprendimento dei parametri e il secondo, LIFTCOVER-LBFGS, utilizza un metodo di ottimizzazione chiamato BFGS a memoria limitata.

Nella seconda parte presentiamo un'estensione del linguaggio LPLP, chiamato Hierarchical Probabilistic Logic Programs (HPLP), in cui le clausole e i predicati sono organizzati gerarchicamente. Un programma in questa linguaggio può essere convertito in una serie di Circuiti Aritmetici (CA) o reti neurali profonde e l'inferenza viene fatta valutando gli CA. La tesi descrive come eseguire l'inferenza e l'apprendimento negli HPLP. Per apprendere i parametri degli HPLP dai dati, abbiamo proposto l'algoritmo, Parameter learning for HIerarchical probabilistic Logic programs (PHIL). Due varianti di PHIL, Deep PHIL (DPHIL) ed EM PHIL (EMPHIL) e le loro versioni regolarizzate sono state presentate. Inoltre, abbiamo proposto l'algoritmo SLEAHP, per Structure LEArning of Hierarchical Probabilistic logic programming, per apprendere sia la struttura che i parametri di HPLP dai dati.

Tutti questi algoritmi sono stati sperimentati su problemi del mondo reale e le loro prestazioni, in termini di tempo, sono migliori dello stato dell'arte ottenendo soluzioni di qualità paragonabile.

Acknowledgements

First of all, i would like to sincerely thank my supervisors Prof. Evelina Lamma and Prof. Fabrizio Riguzzi for their advice and their personal attention for me during this work. Their support and guidance have been of considerable help in carrying out this research.

I thank my lab mates for the discussions and the hours spent working together and for all the fun during coffee breaks.

I thank my family for their ever-increase attention. They helped me believe in me since my childhood. I particularly thank my father, Fadja Armand, and my mother, Mbiatat Fride, who raised me and made me become an adult. Many thanks to my brothers Fabo Fadja Achille Fredy, Gnose Fadja Aristide, Fadja Fadja Armand and my sisters Nzouakeu Fadja Armelle and Heumi Fadja Arlette for their unconditional support.

A special thank to Prudencia and Kesiah for the support and encouragements. They gave me strength and helped me overcome difficult moments during my PhD years.

To my family

Contents

List of Figures

List of Tables

List of Algorithms

Part I

Introduction

Chapter 1

Motivation

Because of the increase of the number of connected devices and the huge amounts of data flowing among them, it is important nowadays to investigate how to implement systems in order to manage large quantities of data. The Gartner institute claimed that, by 2020, almost 30 billions devices will be inter-connected through the Internet of Thing (IoT). These devices should be able to exponentially generate large amounts of structured data (database, on-line transactions, log files) and unstructured data (images, videos, social network data, GPS data, sensors data). The larger will these data be, the more efficient should be the systems for analyzing, reasoning and learning from them. The phenomenon of ever-increasing data is known in the literature as *big data*, see Figure 1.1.

When dealing with big data, various challenges have to be considered. We have to:

1. define techniques for collecting large amounts of data.
2. improve techniques for storing these data.
3. define robust, *scalable* and high-performance systems for representing, analyzing and extracting knowledge from ever-increasing amounts of data. These systems should also be able to manage uncertainty, typical of many real world domains.

To represent data and relationhips among them, First-Order Logic (FOL) and Logic programming (LP) are some of the most used formalism. They define

Figure 1.1: Big data.

not only tools for representing individuals from several domains of interest, but also for modeling relation among them. Recently, declarative approaches, based on logic, have been combined with *probability theory*, to represent and reason in domains characterized by uncertainty. This combination, called Probabilistic Logic Programming, allows the representation of data in uncertain domains by integrating logic and probability. This combination is particularly useful because elements are (almost) never absolutely true or false, but are affected by uncertainty. One of the most powerful semantics for PLP was proposed by Taisuke Sato in 1995 [121]: the *distribution semantics* (DS). Since then, many languages under the DS have been proposed such as PRISM [122], Independent Choice Logic (ICL) [132], Logic Programs with Annotated Disjunctions (LPADs) [142], ProbLog [60], just to name a few. Each of these languages allows not only to represent data as a set of facts and rules, the *structure* of the program, but also to assign probabilities, the *parameters* of the program, to these facts and rules in order to represent uncertainty. Reasoning means computing the probability of certain facts, called *queries*, given a PLP and a background knowledge, typically a set of true facts in the domains. Learning means inducing the parameters and the structure of PLPs from data. Different learning tasks can be considered:

1. Inductive Logic Programming (ILP)
   This problem deals with domains without uncertainty. Given a dataset (set of facts), ILP induces a logic program consistent with the data.

2. Parameters learning
   In this problem we are given the structure of a logic program and the data. The task is to induce the parameters from data.

3. Probabilistic Inductive Logic Programming (PILP)
   PILP combines the previous tasks and tries to learn both the structure and the parameters of PLPs from data.

Many state-of-the-at systems have been implemented for solving the previous tasks. For ILP, sytems such as FOIL [104], Aleph [129] and Progol [79] have been implemented. State-of-art parameter learning systems such as PRISM [124], EMBLEM [10], LFI-ProbLog [36] have also been proposed. In PILP, systems such as SLIPCOVER [11], ProbFOIL [31] and ProbFOIL+ [105] have been proposed for learning both the structure and parameters from data.

These systems have been successfully applied in many real domains such as link discovery in social networks [29], natural language processing [150, 115, 87], bioinformatics [78, 29, 123], entity resolution [110], just to name a few. They perform well on these domains in terms of solution quality but all suffer from one problem: *scalability*. The time taken by learning in almost all of these systems becomes prohibitive as the size of the data increases.

Many attempts to overcome the scalability problem have been investigated and approaches based on distributed algorithms, such as *Map-Reduce* [112, 15], have been implemented and applied to many real domains. However, these systems still suffer from the exponential growth of data and it is necessary to investigate new languages based on FOL in which reasoning and learning are less expensive. Systems based on these languages should be combined with systems based on distributed algorithms to achieve scalability, fundamental for big data.

Chapter 2

Thesis Aims

Because of the ever-increasing amount of data available on the web and the limits of systems based on distributed algorithms, as described in the previous chapter, one of the most important challenge to face in Statistical Relational Learning (SRL) is to define languages for managing large amounts of data. These languages should be expressive enough to represent many domains, even those affected by uncertainty, and should be computationally less expensive in terms of reasoning and learning.

This thesis aims at defining new languages following the DS formalism that overcome the problem described previously. We first present a set of applications and examples, see Chapter 8, formalized by FOL in PLP which illustrate its expressiveness and maturity. Then, we present our threefold contribution to SRL.

First, we propose two restrictions of the language of LPADs called Liftable Probabilistic Logic Program (LPLP), and Hierarchical Probabilistic Logic Program (HPLP). These languages allow a hierarchy among clauses. While LPLPs allow only one layer of clauses, HPLPs extend LPLPs by allowing many layers of clauses. Inference in these languages is less expensive than for general PLPs. Therefore learning both the parameters and the structure of LPLPs and HPLPs becomes computationally less expensive and consequently more scalable.

Second, different parameter learning algorithms are proposed for each language. For LPLPs, two algorithms based on Expectation Maximization (EM) and on Gradient method (Limited-memory BFGS (LBFGS) [92]) have been implemented. For HPLPs, we propose an algorithm, called Parameter learning for HIerarchical probabilistic Logic programs (PHIL)[1], that estimates the parameters of HPLPs from data. Two versions of PHIL and their regularizations are implemented: the first is based on EM algorithm called Expectation Maximization PHIL (EMPHIL) and the second is based on Gradient descent and particular on the ADAM optimizer called Deep PHIL (DPHIL). We performed experiments on real world data comparing PHIL with state of the art parameter learning algorithms such as EMBLEM and LFI-ProbLog. PHIL shows comparable and often better performance in less time.

Third, we propose two algorithms, LIFTCOVER[2] and SLEAPH, for learning both the structure and the parameters of LPLPs and HPLPs from data respectively. Different versions of these algorithms based on EM and GD were also implemented. Experiments comparing LIFTCOVER and SLEAPH with the state of the art structure learning algorithms SLIPCOVER and ProbFOIL+ also show comparable and often better performance in less time.

[1]The code are available at `https://github.com/ArnaudFadja/phil`.

[2]The code of the systems and the datasets are available at `https://bitbucket.org/machinelearningunife/liftcover`.

Chapter 3

Structure of the thesis

This thesis is divided into 7 parts: Introduction, Logic and Probability, Probabilistic Logic Programming (PLP), Probabilistic Inductive Logic Programming (PILP), Lifted Probabilistic Logic Programming, Hierarchical Probabilistic Logic Programming, Summary and Future Work.

In the introduction, Part I, the motivation and the goals of the thesis are presented in chapters 1 and 2 respectively. Then, the structure of the thesis and the publications related to the themes treated in the thesis are respectively presented in chapters 3 and 4.

In the second part, Part II, the foundations of logic, i.e propositional logic, first order logic and logic programming, are presented in Chapter 5. Chapter 6 illustrates some basic concepts of probability necessary for understanding the algorithms presented in the following parts.

Part III describes how to integrate logic programming and probability theory in order to model domains characterized by uncertainty. Different languages under the distribution semantics including LPADs and ProbLog are described in Chapter 7. How to perform inference in such languages is presented in the same chapter. In Chapter 8, Probabilistic Logic programs in Action, we present a set of examples in various domains in which PLPs can be applied.

Part 10 presents algorithms for learning general PLPs. Chapter 9 discusses ILP including how to structure the search and different approaches for inducing a logic program form data. Chapter 10 describes how to induce both the structure and the parameters of PLPs from data. State of the art parameter (EMBLEM and LFI-ProbLog) and structure (SLIPCOVER and ProbFOIL+)

learning are also presented.

In Part V, we propose a restriction of the language of LPADs called liftable PLP (LPLP). Chapter 11 describes how to perform inference and presents an algorithm, LIFTCOVER, similar to SLIPCOVER, for learning both the parameters and the structure of LPLPs. Two implementations of LIFTCOVER based on Expectation Maximization (IFTCOVER-EM) and on gradient method (LIFTCOVER-LBFGS) are presented.

Another important contribution of this thesis is presented in Part VI. In this part we propose an extension of LPLP, called Hierarchical PLP (HPLP), which is still a restriction of LPADs, and for which inference and learning are less expensive than for general LPADs. In Chapter 12 we present in detail a description of Hierarchical PLPs, how to perform inference in such programs and how to convert them into arithmetic circuits or deep neural networks. Chapter 13 presents the parameter learning algorithm PHIL (DPHIL and EMPHIL) and their regularization versions. The structure learning algorithm, SLEAHP, is presented in Chapter 14.

Part VII summarizes the work conducted in this thesis in Chapter 15 and presents directions for future work in Chapter 16.

3.1 How to Read This Thesis

The thesis is written such that it can be read by both experts and non-experts in logic programming. Figure 3.1 shows the dependencies among the main chapters. In gray we highlight the chapters useful for understanding the basic concepts of logic programming and probability theory as well as PLP languages and state of the art parameter and structure learning systems. Our contribution including PLP in action, LIFTCOVER, PHIL and SLEAHP are in blue.

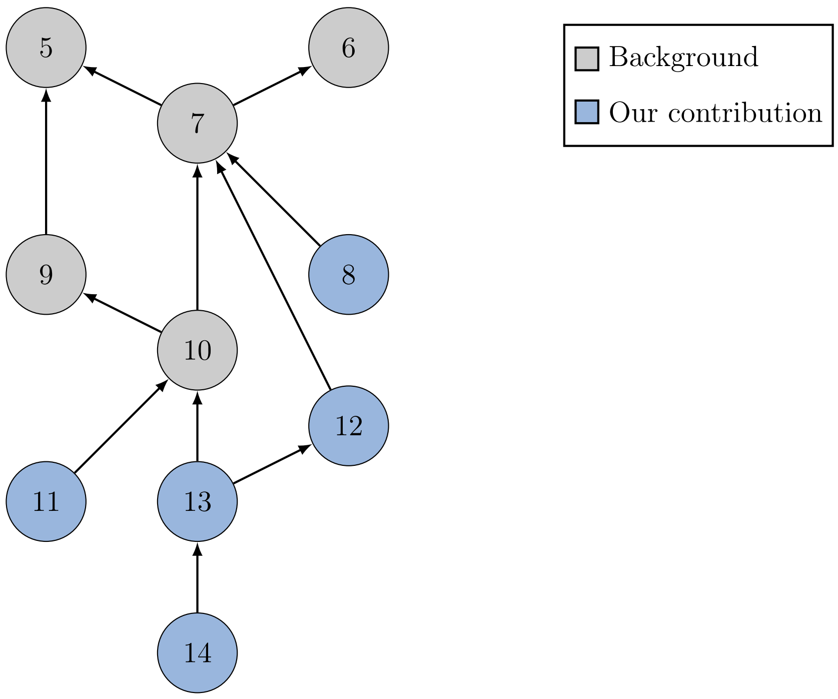


Figure 3.1: Dependency graph of the main chapters. For instance, to understand Chapter 13 you have to read chapters 10 and 12 first.

Chapter 4

Publications

The work described in this thesis was published in:

- International Journals

    1. Arnaud Nguembang Fadja and Fabrizio Riguzzi. Lifted discriminative learning of probabilistic logic programs. Machine Learning, 108(7):1111–1135, © Springer, 2019.
    2. Arnaud Nguembang Fadja, Fabrizio Riguzzi and Evelina Lamma. Learning Hierarchical Probabilistic Logic Programs. International Joint Conference on Learning and Reasoning (submitted).

- Book Chapters

    1. Arnaud Nguembang Fadja and Fabrizio Riguzzi. Probabilistic logic programming in action. In Andreas Holzinger, Randy Goebel, Massimo Ferri, and Vasile Palade, editors, Towards Integrative Machine Learning and Knowledge Extraction: BIRS Workshop, Banff, AB, Canada, July 24-26, 2015, Revised Selected Papers, volume 10344 of Lecture Notes in Computer Science, pages 89–116. Springer, Heidelberg, Germany, © Springer, 2017. The final publication is available at Springer via http://dx.doi.org/10.1007/978-3-319-69775-8_5.

- International Conferences

    1. Arnaud Nguembang Fadja, Fabrizio Riguzzi, and Evelina Lamma. Expectation maximization in deep probabilistic logic programming.

In Chiara Ghidini, Bernardo Magnini, and Andrea Passerini, editors, Proceedings of the 17th Conference of the Italian Association for Artificial Intelligence (AI*IA2018), Trento, Italy, 20-23 November, 2018, volume 11298 of Lecture Notes in Computer Science, pages 293–306, Heidelberg, Germany, 2018. © Springer, Springer.

2. Arnaud Nguembang Fadja and Fabrizio Riguzzi. Lifted discriminative learning of probabilistic logic programs. In Nicolas Lachiche and Christel Vrain, editors, 27th International Conference on Inductive Logic Programming, ILP 2017, 2017.

- International Workshops

1. Arnaud Nguembang Fadja, Evelina Lamma, and Fabrizio Riguzzi. Vision inspection with neural networks. In Marco Maratea and Mauro Vallati, editors, R.i.C.e.R.c.A: RCRA Incontri E Confronti, Proceedings of the RiCeRcA Workshop co-located with the 17th International Conference of the Italian Association for Artificial Intelligence (Ai*iA 2018), volume 2272 of CEUR Workshop Proceedings, pages 1–10, Aachen, Germany, 2018. © by the authors, Sun SITE Central Europe.

2. Arnaud Nguembang Fadja, Fabrizio Riguzzi, and Evelina Lamma. Deep learning for probabilistic logic programming. In Marco Rospocher, Luciano Serafini, and Sara Tonelli, editors, AI*IA 2018 Doctoral Consortium, Proceedings of the AI*IA Doctoral Consortium (DC), volume 2249 of CEUR Workshop Proceedings, pages 43–47, Aachen, Germany, 2018. © by the authors, Sun SITE Central Europe.

3. Arnaud Nguembang Fadja, Fabrizio Riguzzi, and Evelina Lamma. Learning the parameters of deep probabilistic logic programs. In Elena Bellodi and Tom Schrijvers, editors, Probabilistic Logic Programming (PLP 2018), volume 2219 of CEUR Workshop Proceedings, pages 9–14, Aachen, Germany, 2018. © by the authors, Sun SITE Central Europe.

4. Arnaud Nguembang Fadja, Evelina Lamma, and Fabrizio Riguzzi. Deep probabilistic logic programming. In Christian Theil Have

and Riccardo Zese, editors, Proceedings of the 4th International Workshop on Probabilistic logic programming, (PLP 2017), volume 1916 of CEUR Workshop Proceedings, pages 3–14, Aachen, Germany, 2017. Sun SITE Central Europe.

Part II

Logic and Probability

Chapter 5

Logic

Logic can be defined as the study of different forms of reasoning with the main objective of recognizing and identifying the correct forms of reasoning and distinguishing them from incorrect ones. It is often called *formal* or *symbolic* logic because the forms of logic are expressed by means of *symbols*. The aim of this chapter is to discuss and present the different forms of logic used in machine learning and statistical relational learning. After presenting a brief introduction in Section 5.1, Propositional logic, First Order Logic and Logic Programming are described in Sections 5.2, 5.3 and 5.4 respectively.

5.1 Introduction

Logic and Propositional Logic date back to Aristode whose purpose was to model reasoning. Modeling reasoning is useful in many fields of research such as planning, automated reasoning, machine learning and artificial intelligence in general. Propositional logic aims at defining a syntax for representing propositions, for defining their semantic and truth. Since Propositional logic is not enough expressive to represent relational domains, other logic formalisms have been proposed: First Order Logic, FOL, and Logic Programming, LP. Both formalisms allow *predicate* and *function symbols* that are often useful for representing complex relations among entities. FOL and LP share the same syntax but differ in the way they define their semantics. The following sections describe each approach in turns.

5.2 Propositional Logic

Propositional logic defines a way for representing sentences called propositions. It does not investigate the meaning of the individual propositions, but only the schemes in which the propositions can be composed, specially by operators called *logical connectives*. The purpose is to assign a truth to statements (or logical expressions) obtained by composing simple propositions. In order to define such a language, we need to formally define an *alphabet*, which consists of symbolic elements used for representing propositions (and statements), a *syntax*, which defines how to obtain the statements by combining elements in the alphabet and the logical connectives. Finally, we need a *semantic* for defining the truth of each statement. Let us present these components for propositional logic. They will also be presented in the case of FOL and LP.

Alphabet

The alphabet is a set of symbols and constructs necessary for building sentences in a language. Statements in propositional logic are built by combining a set of:

- Constants (truth values): *{True, False}* also indicated {T, F} or {0,1} or $\{\top,\bot\}$
- Propositional symbols {a,b, ..., z}
- Logical connectives: $\neg$ (not), $\wedge$ (and), $\vee$ (or), $\rightarrow$ or $\supset$ (implication) , $\leftrightarrow$ or $\equiv$ (equivalence)
- Auxiliary symbols: "(" (left parenthesis), ")" (right parenthesis)

5.2.1 Syntax

The syntax defines how to combine alphabet's constructs for representing propositions. A proposition can be *simple* or *compound*. A simple (or atomic) proposition is a statement which:

- can be *{True, False}*,
- can be formalized by a propositional symbol.

Example 1. *Simple propositions*

1. *Each triangle can be inscribed in a circle*

2. *Rome is in France*

Note in the example that, we are not interesting in the meaning of each proposition. We only need to give their truth. The first proposition is *true* and the second is *false*.

Atomic propositions can be compounded by means of logical connectives. If a an b are atomic propositions, then $\neg(a)$, $(a \wedge b)$, $(a \vee b)$, $(a \rightarrow b)$, $(a \leftrightarrow b)$ are compound propositions. Formally, let L be the set of constants and propositional symbols: the set, P, of propositions (or formulas) can be inductively defined as follows:

- $\forall a \in L$, $(a) \in P$
- $\forall p \in P$, $\neg(p) \in P$
- $\forall p, q \in P$, $(p \vee q)$, $(p \wedge q)$, $(p \rightarrow q) \equiv (\neg p \vee q)$, $(p \leftrightarrow q) \equiv ((p \wedge q) \vee (\neg p \wedge \neg q)) \in P$

The proposition $(p \vee q)$ is called *disjunction* and a and b are *disjuncts.* $(p \wedge q)$ is called *conjunction* and a, b *conjuncts.* The proposition $(p \rightarrow q)$ is called *implication*: p is called the *antecedent* and q the *consequent.* Note that precedence among logical connectives allows to reduce the number of parentheses necessary to correctly interpret a proposition. From the highest to the lowest precedence we have:

$\neg$,
$\wedge$,
$\vee$,
$\rightarrow, \leftrightarrow$.

Example 2. *Examples of propositions:*

$p \qquad q \qquad p \wedge q$

$p \wedge (q \vee z) \qquad \neg p \vee (q \wedge \neg z)$

5.2.2 Semantic

The semantics is the set of rules that allow you to associate a truth value (*true* or *false*) with each proposition, starting from the value of its propositional symbols. The semantics of the connectives is illustrated by Table 5.1, called truth table.

Table 5.1: Truth table

a	b	$a \wedge b$	$a \vee b$	$\neg a$	$a \rightarrow b$	$a \leftrightarrow b$
T	T	T	T	F	T	T
T	F	F	T	F	F	F
F	T	F	T	T	T	F
F	F	F	F	T	T	T

An *interpretation* of a proposition is a function (called *boolean function*) that assigns one of the two truth values (T or F) to each atomic proposition and which assigns a truth value to the compound proposition based on the truth table. An assignment that makes a proposition true is called a *satisfying assignment.* An interpretation is then a boolean function $b : P \rightarrow \{T, F\}$. If p is composed of n atomic propositions, its truth table will have 2^n entries each corresponding to an interpretation.

Example 3. *The expression $E = p \wedge (p \vee q)$, for the assignment $p = T$ and $q = F$ is evaluated to T. So this assignment is a satisfying assignment for E. E can be evaluated for other three assignments and thus build its entire boolean function.*

If an interpretation (or an assignment) , $b(.)$, makes a proposition p true (b(p)=T), *b satisfies p.* A proposition p is said to be *satisfiable* if there exists at least one interpretation (assignment), b(·), which satisfies it.

5.3 First Order Logic

Propositional logic allows to represent statements and their truth. It is not expressive enough to represent various types of statements that are used in computer science and especially in machine learning. First Order Logic (FOL), also called *predicate logic*, extends the expressiveness of propositional logic by allowing formulas which contains *quantifiers* and *variables*. This extension allows FOL to represent different types of relationship among objects in many domains of interest. FOL has a *syntax*, which defines how to form formulas, and a *semantics* which gives an interpretation to these formulas.

5.3.1 Syntax

FOL's alphabet (Σ), an extension of proposition logic alphabet, defines the following set of symbols:

- *Constants*, individual entities of the domain.
    - e.g. a, b, maria, 3, jeff.
- *Logical Variables*, alphanumeric strings (here starting with an uppercase letter adopting the logic programming convention) which refer to objects in the domain.
    - X,Y,Z
- *function symbols (or functors)*, denoted by alphanumeric strings (here starting with a lower-case character). It univocally identifies an object of the domain through a relationship between other n (called arity) objects of the domain. We will use the notation f/n to denote a function symbol f with arity n.
    - e.g. $f(a, b)$, $mother(maria)$, $s(X)$
- *Predicate symbols*, alphanumeric strings (here starting with a lowercase character) which is a generic relation (which may be true or false) among n objects of the domain of discourse. Even in this case we will use p/n to indicate a predicate p with arity n.
    - e.g. mother(maria,jeff), brother(jeff,paul), p(X)
- *Logical connectives*, used for constructing formulas
    - e.g. $\neg$, $\wedge$, $\vee$, $\rightarrow$ or $\supset$, $\leftrightarrow$ or $\equiv$
- *Quantifiers*, for expressing generality

  - e.g. $\forall$ universal quantifier ("for all"), $\exists$ existential quantifier ("there exists")

- *Auxiliary symbols*, for defining precedences among connectives
  - e.g. "(", ")"

From this alphabet, we can define the following expressions:

- A *term* is either a constant, a variable or, if f is a function symbol with arity n and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term.
  - e.g. maria, $f(X)$,
- An *atom* (or an *atomic formula*) is obtained by applying a predicate symbol p to n terms $t_1, \ldots, t_n$: $p(t_1, \ldots, t_n)$
  - e.g. mother(maria,jeff)
- A *literal* is either an atom a (positive literal) or its negation $\neg a$ (negative literal)
  - e.g. mother(maria,jeff), ¬mother(maria,jeff)

Definition 1. *FOL well-formed formula (wff)*
Formulas in FOL are recursively defined as follows:

1. *atomic formulas or atoms are formulas;*
2. *true and false are formulas;*
3. *if F and G are formulas, then so are $(\neg F)$, $(F \wedge G)$, $(F \vee G)$, $(F \rightarrow G)$, $(F \leftrightarrow G)$;*
4. *if F is a formula and X is a variable, then so are $(\exists X F)$ and $(\forall X F)$*

In order to correctly interpret a formula, the following precedence is adopted from the highest to the lowest
$\neg$,$\exists$, $\forall$
$\wedge$,
$\vee$,

$\rightarrow, \leftarrow, \leftrightarrow$.

The *scope* of a quantifier is the wff which immediately follows. In the case of ambiguity the auxiliary symbols, "(" and ")", are used. For example, in the following formula

$$F_1 = \forall X(p(X,Y) \wedge q(X)) \vee q(X)$$

the scope of $\forall$ is $p(X,Y) \wedge q(X)$

A variable is *free* if and only if it does not appear in the scope of any quantifier. Otherwise it is *bound.* In the formula

$$F_2 = \forall X(p(X,Y) \wedge q(X))$$

Y is free and X is bound.

A formula is *open* if and only if it has free variables. Otherwise it is *closed.* For example, the formula $\forall X \forall Y p(X,Y)$ is closed.

A clause, C, in FOL is a disjunction where all the variables are universally quantified:

$$\forall X h_1 \vee \ldots \vee h_n \vee \neg b_1 \vee \ldots \vee \neg b_m$$

where X is the set of variables appearing in C. The universal quantifier, $\forall$, is often implicit and can be omitted. C can equally be seen as a set of literals

$$\{h_1, \ldots, h_n, \neg b_1, \ldots, \neg b_m\}$$

where the disjunction among the literals is implicit. A clause can also be written as

$$\forall X h_1 \vee \ldots \vee h_n \leftarrow b_1 \wedge \ldots \wedge b_m$$

The form of a clause to be used in the following will be clear from the context. $h_1 \vee \ldots \vee h_n$ is the *head* of the clause (written $head(C)$) and $b_1 \wedge \ldots \wedge b_m$ its *body*, written $body(C)$. The clause states that: "if the conjunction of all the $b_j s$ is true then so is the disjunction of all $h_i s$". If the body is empty, the clause is called *fact.* The clause with exactly one head (or one positive literal) is called a *definite clause.* An *expression* is a literal, a term or a clause. An expression is ground if it does not contain variables. *Unification* is a formal procedure used to establish when two expressions can coincide by making appropriate substitutions. An application of a *substitution* $\theta = \{V_1/t_1, \ldots, V_n/t_n\}$ to an expression E, written $E\theta$, produces a new expression obtained by simultaneously replacing each variable V_i of E with the corresponding term t_i. $E\theta$ is called an instance of E. A substitution grounds E if all variables in E are replaced by a term. A *theory* T is a set of clauses. A definite theory is a finite set of definite clauses.

The *Herbrand universe* of a theory T, $HU(T)$, is the set of all ground terms constructed by using the function symbols and constants in T. The *Herbrand base* of a theory T, denoted $HB(T)$, is the set of all ground atoms obtained from predicates appearing in T and terms of its Herbrand universe.

Example 4. *Given the following theory T*

$$parent(jeff, paul)$$
$$parent(paul, ann)$$
$$grandparent(X, Y) \longleftarrow parent(X, Z), parent(Z, Y)$$

its Herbrand universe is

$$HU(T) = jeff, paul, ann$$

and the corresponding Herbrand base is

$$HB(T) = \{parent(jeff, jeff), parent(paul, paul), parent(ann, ann), parent(jeff, paul), parent(jeff, ann), parent(paul, jeff), \ldots, grandparent(jeff, jeff), grandparent(paul, paul), \ldots\}$$

5.3.2 Semantics

In the previous section, we defined the FOL'S alphabet and presented a syntax for building well formed formulas. This section describes how to associate a meaning to formulas in FOL by means of interpretations, which is a process that associates an object to each term and a truth-valued to each formula.

An *Herbrand interpretation* (or *interpretation* for simplicity) of a theory T is a set of ground atoms subset of the Herbrand base of the theory $HB(T)$. If a ground atom is in the interpretation, it is said to be true (with respect to the interpretation) otherwise it is false. A Herbrand interpretation is also called a two-valued interpretation because it assigns truth values (true or false) to formulas. Formulas which are not true are false. An interpretation is a (Herbrand) *model* of a closed formula Φ if Φ is evaluated to true with respect to I. Formally, if I is an interpretation and Φ is a formula, Φ is true in I (or is satisfied in I), written $I \vDash \Phi$ if:

1. if Φ is a ground atom a, $a \in I$
2. if Φ is a ground negated-atom $\neg a$, $a \notin I$
3. if Φ is a conjunction $\Phi_1 \wedge \Phi_2$, $I \vDash \Phi_1$ and $I \vDash \Phi_2$
4. if Φ is a disjunction $\Phi_1 \vee \Phi_2$, $I \vDash \Phi_1$ or $I \vDash \Phi_2$
5. if $\Phi = \forall X \Phi_1$ for all substitution θ that assign a value to all variables of X, $I \vDash \Phi_1\theta$
6. if $\Phi = \exists X \Phi_1$ there exists a substitution θ that assigns a value to all variables of X such that $I \vDash \Phi_1\theta$

Let S be a set of closed formulas: an interpretation I satisfies (or is a model of) S if I is a model for every formula in S.

A closed formula Φ is a *logical entailment* or a *logical consequence* of the set of closed formula S, written $S \vDash \Phi$, if every model of S is also a model of Φ. Let C be a clause of the form

$$h_1; \ldots; h_n \leftarrow b_1, \ldots, b_m$$

where the semicolon ";" is equivalent to disjunction and the comma "," to conjunction. C is satisfied (or true) in an interpretation I iff for any substitution θ grounding C, if $I \vDash \mathit{body}(C)\theta$ then $I \vDash \mathit{head}(C)\theta$ (or $\mathit{head}(C)\theta \cap I \neq \emptyset$). Otherwise it is false. Particularly, if C is a define clause, C has one atom h in its head, C is satisfied (or true) in an interpretation I iff for any substitution θ grounding C, if $I \vDash \mathit{body}(C)\theta \rightarrow h \in I$.

A theory P (set of clauses) is true in an interpretation I iff all its clauses are true in I and we write

$$I \vDash P$$

When P is true in an interpretation I, I is said to be a model for P. To show that an interpretation is not a model for P, it is sufficient to show that one clause in P is false in I. In this work and in StarAI in general, given a theory P and a query Q (set of atoms), we will be interested in deciding if Q is a logical consequence of P i.e $P \vDash Q$. This means that every model of P must be a model for Q.

Herbrand interpretations and models provide a complete way for defining semantics of a set of clauses in the sense that a set of clauses is unsatisfiable if and only if it does not have a Herbrand model, see [45].

5.4 Logic Programming

Logic Programming (LP) is based on FOL with a different semantic. It was proposed in the early 1970s by Kowalski who elaborated its theoretical foundations, [69]. The idea was to make the computer representing knowledge and performing reasoning. Reasoning means inferring true assertions from known assertions.

For definite clauses, of the form $\{h_1, \neg b_1, \ldots, \neg b_m\}$, Herbrand models have an important property: if a program P is a set of definite clauses, the intersection of a set of Herbrand models of P is still a Herbrand model of P. This model is called the *minimal Herbrand model* of P, written lhm(P). The lhm(P) of P always exists and is the set of atoms which are logical consequence of P. So $P \models a$ iff $a \in lhm(P)$. The following example

$human(X) \leftarrow female(X)$

$female(mary)$

has the $lhm(P) = \{female(mary), human(mary)\}$.

The procedure that proves if a formula Φ is a logical consequence of a program P is called *resolution.* If the program is a define logic program, i.e a set of definite clauses, the resolution is called *linear resolution with selection function for definite logic programs* (SLD-resolution). Before defining how does a SLD-resolution works, let us define some useful concepts.

Let $\theta = \{V_1/t_1, \ldots, V_n/t_n\}$ be a substitution and a, b two atoms: a and b can be unified if there exist a substitution θ such that $a\theta = b\theta$. θ is the most general general unifier (mgu) if it is the minimal substitution such that $a\theta = b\theta$ i.e there is no substitution δ such that $\theta = \delta\sigma$, where σ is a substitution. Given a definite logic program P and a query (or goal)

$$G_0 = q_1, \ldots, q_m$$

the fundamental reasoning method at the basis of SLD-resolution can be summarized to the following inference rule:

$$\frac{\leftarrow q_1, \dots q_{i-1}, q_i, q_{i+1}, \dots, q_m \qquad h \leftarrow b_1, \dots, b_n}{\leftarrow (q_1, \dots q_{i-1}, b_1, \dots, b_n, q_{i+1}, \dots, q_m)\theta}$$

where

- $q_1, \dots, q_m$ are atoms;
- $h \leftarrow b_1, \dots, b_n$ is a (possibly renamed) definite clause in P $(n \geqslant 0)$
- $\theta = mgu(q_i, h)$

Let $Q = \neg G_0$ be a query. This inference rule states that in order to resolve a goal G_0, the SLD-resolution initially selects a subgoal, q_i, and replaces G_0 with the body of a clause in P whose head unified (mgu) with q_i. This creates a new goal

$$G_1 = (q_{i-1}, b_1, \dots, b_n, q_{i+1}, \dots, q_m)\theta$$

then the procedure iteratively repeats creating a sequence of goals

$$G_0 \sim G_1 \sim \dots \sim G_k$$

to be resolved. The resolution stops for two reasons: 1) the current goal is empty ($G_i = \square$) and there is a *refutation* of G_0 i.e G_0 fails and so Q succeeds 2) the head of every clause in P cannot be unified with the current selected subgoal, the query Q fails.

Note that, at each iteration of resolution, two strategies have to be considered: the subgoal's selection strategy and clause selection strategy. SLD-resolution is *sound*, i.e if a goal G succeeds in a program P then G is a logical consequence of P. SLD-resolution is also *complete*, i.e if a goal G can be resolved by a program P, then there is a refutation of $P \sqcup \{\neg G\}$. soundness and completeness are two fundamental properties of SLD-resolution and are strongly depend on the resolution strategies. For example Prolog, a logic programming language, adopts the following strategies during resolution:

- Subgoals in the current goal are selected from left to right

- Clauses whose head unifies with the selected subgoal are chosen top-down

These strategies could make Prolog **incomplete**. Let us consider the following examples:

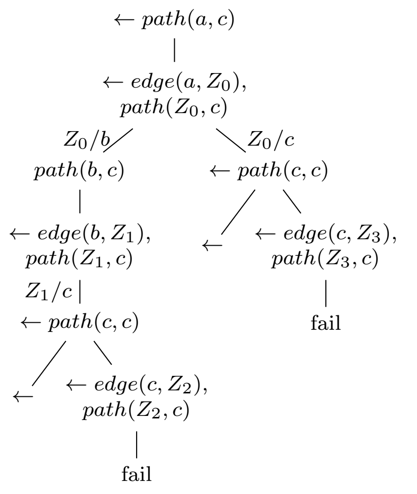
Example 5 (Paths in a graph- Prolog)**.** *The following program computes paths in a graph:*

$path(X, X).$
$path(X, Y) \leftarrow edge(X, Z), path(Z, Y).$
$edge(a, b).$
$edge(b, c).$
$edge(a, c).$

$path(X, Y)$ is true if there is a path from X to Y in the graph where the edges are represented by facts for the predicate $edge/2$.

The first clause states that there is a path from a node to itself. The second states that there is a path from a node X to a node Y if there exists a node Z such that there is an edge from X to Z and there is a path from Z to Y.

Figure 5.1 shows the SLD tree for the query $path(a, c)$. The labels of the edges indicate the most general unifiers used. The query has two successful derivations, corresponding to the paths from the root to the $\leftarrow$ leaves.



$\leftarrow path(a,c)$
$\leftarrow edge(a,Z_0), path(Z_0,c)$
Z_0/b
Z_0/c
$path(b,c)$
$\leftarrow path(c,c)$
$\leftarrow edge(b,Z_1), path(Z_1,c)$
$\leftarrow$
$\leftarrow edge(c,Z_3), path(Z_3,c)$
Z_1/c
$\leftarrow path(c,c)$
fail
$\leftarrow$
$\leftarrow edge(c,Z_2), path(Z_2,c)$
fail

Figure 5.1: SLD tree for the query $path(a, c)$ from the program of Example 5.

Example 6. *Consider the following example:*

$sibling(X, Y) \leftarrow sibling(Y, X).$
$sibling(b, a).$

Suppose we want to resolve the goal $G_0 = sibling(a, X)$: the head of two clauses in the program unify with G_0. Prolog chooses the first clause and according to the inference rule the new goal $G_1 = sibling(Y, a)$ is generated. To resolve G_1 the first clause is again chosen and then the subgoal $G_2 = sibling(a, X) = G_0$ is generated coming back to the first goal. The resolution proceeds and runs into an infinite loop. This example shows the incompleteness of Prolog. Note that to avoid going into an infinite loop, it is sufficient to change, textually, the order of the first and the second clause.

Chapter 6

Probability Theory

First order logic is very effective for modeling entities and relation in domains in which answers of queries are certain. However, there are domains affected by uncertainty in which answering a query with certainty becomes impossible. Therefore, it is appropriate to extend FOLs in order to model uncertainty. Modeling and representing uncertainty can be done with probability theory. This concept, used since the 17th century, has become over time the basis of various scientific disciplines such as artificial intelligence and specially machine learning. Before presenting, in the following part of this work, how to combine FOLs with probability, let us give a brief background of the concepts underlining probability theory.

6.1 Event Spaces

In probability theory we consider a phenomenon observable exclusively from the point of view of the possibility or not of its occurrence, regardless of its nature. If the experiment under examination is deterministic, the result of the observation can be predicted exactly. If the experiment is random, for example throwing a die, the result of the observation is not known a priori and it is possible to define a set Ω, called *sample space* or *universe*, that contains all the possible outcomes ($\{1, 2, 3, 4, 5, 6\}$ when throwing a die). Each element $\omega \in \Omega$ is a *sample point.* An event, E, is a set of sample points subset of Ω, e.g $E = \{2, 4, 6\}$. An *elementary event* $\{\omega\}$ is an event that contains only one sample point. The *certain event* contains all the sample points and is equivalent to Ω. An impossible event $\emptyset$ does not contains sample points.

6.2 Probability Distributions

There are three mains definition of probability: the *classical*, the *empirical* and the *axiomatic* definition.

Definition 2. *Classical definition of probability*

The probability $P(E)$ of a random event E is the ratio between the number of favorable cases, n_E, and the number of possible cases, n, where all elementary events are considered to be equally likely.

$$P(E) = \frac{n_E}{n}$$

Example 7. *Throwing a die*

$\Omega = \{1, 2, 3, 4, 5, 6\}$. *Let* $E = \{1, 2\}$ *then* $P(E) = \frac{2}{6} = \frac{1}{3}$

This classical definition allows to effectively compute the probability of events in many situations. It is an **operational definition** in the sense that the definition also provides a method for computing the probability. However, it presents different drawbacks:

1. It can be applied only to experiments in which elementary events are **equally likely**.
2. It hypotheses a finite set of possible outcomes.
3. The definition is circular because the notion of probability (equally likely of elementary events) is used to define the probability itself.

In order to overcome these downsides, Richard von Mises [146] proposed an empirical definition of probability: the probability of an event is the limit to which the relative frequency of the event tends as the number of experiments increases. Suppose an experiment can be run an infinite number of times and let an event E occurs n_E times out of n experiments then

$$P(E) = \lim_{n \to \infty} \frac{n_E}{n}$$

Even if this definition tends to overcome some limitations of the classical definition of probability, for example no hypothesis on the probability of elementary events, it still suffers from some limitations:

- Not all experiments are repeatable.
- It is not always possible to imagine infinitely repeatable experiments.

One can subjectively define a probability as the assessment that the individual can coherently formulate, based on his knowledge, the degree of verifiability of an event. e.g: let E="it will rain tonight. One can state that $P(E) = 0.6$ which represents his own degree of belief based on evidence such as "the sky is cloudy". This leads to the following definition proposed by Andrey Nikolaevich Kolmogorov [68], called axiomatic/ Kolmogorov definition of probability which holds for any interpretation of probability.

Definition 3. *Kolmogorov definition of probability*

Consider a random experiment and $E \subseteq \Omega$ an event. The probability of E, $P(E)$, is defined as a function that associates to E a non-negative real number such that the sum of the probabilities of all elementary events is equal to 1. This number meets the following three conditions:

1. *Probability is a non-negative number: $P(E) \geqslant 0$.*
2. *The probability of the certain event is unitary: $P(\Omega) = 1$.*
3. *Given two events E_1 and E_2 defined as mutually exclusive ($E_1 \cap E_2 = \emptyset$), then $P(E_1 \cup E_2) = P(E_1) + P(E_2)$.*

From this definition we can derive the following assertions:

1. $P(\emptyset) = 0$ and $P(E) \leqslant 1$.
2. If $E^c = \Omega \setminus E$ then $P(E^c) = 1 - P(E)$.
3. If $E_1, E_2 \subseteq \Omega$ are two events, then
   $P(E_1 \cup E_2) = P(E_1) + P(E_2) - P(E_1 \cap E_2)$.
4. If $E_1, E_2 \subseteq \Omega$ are two events, then
   $P(E_1) = P(E_1 \cap E_2) + P(E_1 \cap E_2^c)$.
5. If $E, E_1, \dots E_n \subseteq \Omega$ are n+1 events where E_i are mutually exclusive ($\cap_i^n E_i = \emptyset$), then $P(E) = P(E \cap E_1) \dots P(E \cap E_n)$.

6.3 Conditional Probability

Definition 4. *Conditional probability*

Let E_1, E_2 be two events: we define the conditional probability of E_1 given E_2, written $P(E_1|E_2)$, the probability that the event E_1 occurs when we know that E_2 occurred, as:

$$P(E_2|E_1) = \frac{P(E_2 \cap E_1)}{P(E_1)} \qquad\qquad P(E_1) > 0 \tag{6.1}$$

Consider the following example:

Example 8. *Consider a distribution over a population of students taking a certain course. The space of outcomes, Ω, is the set of all students. Let us consider the events S_{GA} ="students with grade A" and S_{HI} ="students with high intelligence". Consider the probability of these events: $P(S_{GA})$, $P(S_{HI})$ and their intersection $P(S_{HI} \cap S_{GA})$ (the set of intelligent students who got grade A). If we observe that a student has grade A, what is the probability that the student is intelligent? the answer is given by the conditional probability*

$$P(S_{HI}|S_{GA}) = \frac{P(S_{HI} \cap S_{GA})}{P(S_{GA})}$$

which is the probability that the student is intelligent conditioned by the fact that he/she got grade A.

From Equation 6.1, we can derive the equation, called the *chain rule*

$$P(E_1 \cap E_2) = P(E_1) \cdot P(E_2|E_1) \tag{6.2}$$

which can be generalized as

$$\begin{aligned} P(E_1 \cap \cdots \cap E_n) &= P(E_1) \cdot P(E_2|E_1) \ldots P(E_n|E_1 \cap \ldots \cap E_{n-1}) \\ &= \prod_{i=1}^{n} P(E_i|E_{i_1} \ldots E_1) \end{aligned} \tag{6.3}$$

Equation 6.3 states that the probability of the intersection of several events can be expressed in terms of the probability of the first, the probability of the second given the first and so on. Before applying the equation, we have to define an order among events. Note that the equation holds for any order.

Another direct consequence of equation 6.2 is the Bayes theorem

$$P(E_1|E_2) = \frac{P(E_2|E_1) \cdot P(E_1)}{P(E_2)} \tag{6.4}$$

Example 9. *From Example 8 let us define the event* $Intelligent$ = *"students who are intelligent" which gives the probability that a student is intelligent. Suppose we have these probability* $P(Intelligent) = 0.3$ *and* $P(S_{GA} = 0.2)$*. We believe, based on some past statistics, that* $P(S_{GA}|Intelligent) = 0.6$*. If we observe that a new student has grade A, we want to compute the probability that he/she is intelligent. From equation 6.4 we have*

$$P(Intelligent|S_{GA}) = \frac{P(S_{GA}|Intelligent) \cdot P(Intelligent)}{P(S_{GA})} = 0.6 \cdot \frac{0.3}{0.2} = 0.9$$

which obviously states that if a student has a grade A, he/she is likely to be intelligent. Note that $P(S_{GA} = 0.2)$*. This means that obtaining grade A is rare. So only students who are intelligent tend to have grade A. However, if* $P(S_{GA})$ *were higher, say* $P(S_{GA} = 0.7)$*, maybe because tests are often easier, we would have*

$$P(Intelligent|S_{GA}) = 0.6 \cdot \frac{0.3}{0.7} = 0.26$$

which states that obtaining grade A does not effectively means that the student is intelligent.

6.4 Random Variables and Distributions

In several experiments, it is useful to associate a value to each sample point in the sample space in order to avoid creating many variables for each event. In Example 8, we created the event S_{GA} which denotes the set of students whose Grade is A. Suppose there are different levels of grade, let us say $\{A, B, C\}$: in order to represent an event whose outcomes are the set of student who obtained a certain grade, we can create one variable for each event, for example S_{GA}, S_{GB} and S_{GC} which denote students whose grade are A, B, C respectively. This procedure could be annoying and cumbersome if there are several different grades. To overcome this issue, we can create a variable, S_G, whose values are the different grades, $\{A, B, C\}$. We write $S_G = x$ where $x \in \{A, B, C\}$. The variable S_G is called *random variable* since the experiment is random and its values are not known deterministically. So the random variables S_G associates a value to each sample point. Now let us give a formal definition of random variable.

Definition 5. *Random variable*
A random variable X is a function which associates a value val $\in \mathbb{D}$ to each element in Ω.

$$f_X \colon \Omega \to \mathbb{D}$$

Example 10. *Consider an experiment where a fair coin is tossed twice. The sample space is $\Omega = \{HH, HT, TH, TT\}$. If the variable X denotes the number of heads, X is a random variable whose values are $\{0, 1, 2\}$ as shown in Table 6.1. $X = 1$ is a shorthand for the event $\{\omega \in \Omega : f_X(\omega) = 1\} = \{HT, TH\}$ which are events with exactly one head landed.*

Table 6.1: Random variable (tossing a coin twice)

	HH	HT	TH	TT
X	2	1	1	0

Random variable whose values belong to a finite or countable set (e.g $\mathbb{N}$) is said to be a *discrete random variable.* The ones whose values belong to uncountable (e.g $\mathbb{R}$) are *continuous.* In this thesis, we will use uppercase letters (e.g X, Y, Z) to denote random variables and lowercase letters to denote their values. So x refers to a generic value of X. Set of variables are represented in boldface type, e.g **X**, **Y**, **Z**.

The *probability distribution* is a mathematical function which describes the possible values, x, of a discrete random variable, X, and their associated probabilities, $P(X = x)$. It has the following properties:

1. $P(X = x) \geq 0$.
2. $\sum_x P(X = x) = 1$.

where the sum in 2 is taken over all possible values of x. $P(x)$ is sometimes used as a shorthand for $P(X = x)$.

Example 11. *Let us compute the probability distribution corresponding to the random variable X of Example 10. Considering that the coin is fair we have $P(\omega) = \frac{1}{4}$, $\omega \in \{HH, HT, TH, TT\}$. If X denotes the number of landed heads,*

$P(X = 0) = P(TT) = \frac{1}{4}$
$P(X = 1) = P(HT \cup TH) = P(HT) + P(TH) = \frac{1}{4} + \frac{1}{4} = \frac{1}{2}$
$P(X = 2) = P(HH) = \frac{1}{4}$

The probability distribution function is shown in Table 6.2. Note that

$$\sum_{x \in \{0,1,2\}} P(X = x) = 1$$

Table 6.2: Probability distribution (tossing a coin twice).

x	0	1	2
P(X=x)	$\frac{1}{4}$	$\frac{1}{2}$	$\frac{1}{4}$

If X is a continuous random variable, its probability distribution function (pdf) is defined as

$$P(X \leqslant a) = \int_{-\infty}^{a} p(x)dx$$

where $p(x)$ is the probability density function such that

$$\int_{-\infty}^{+\infty} p(x)dx = \int_{Values(X)} p(x)dx = 1$$

$Values(X)$ is the set of values of X.

6.5 Expectation of a Random Variable

Let X be a discrete random variable having a finite set of values, $x_1, \ldots, x_n \in \mathbb{R}$, and P a distribution over X such that $P(x_i) = p_i$ with $\sum_i^n p_i = 1$. The *expectation* of X is defined as

$$E[X] = \sum_i^n x_i \cdot p_i$$

If X is a continuous random variable with probability density function $p(x)$, the expectation is defined as

$$E[X] = \int_{-\infty}^{+\infty} x \cdot p(x) dx$$

Since all probabilities sum up to 1 ($\sum_i^n p_i = 1$ or $\int_{-\infty}^{+\infty} p(x)dx = 1$), the expectation of X is the weighted average of x_i, with p_i's being the weights.

Example 12. *When throwing a fair coin twice, see Example 10, if X denotes the number of heads, the expected value of X is given by:*
$E[X] = 0 \cdot P(X = 0) + 1 \cdot P(X = 1) + 2 \cdot P(X = 2) = 0 \cdot \frac{1}{4} + 1 \cdot \frac{1}{2} + 2 \cdot \frac{1}{4} = 1$ *which is the average of $Values(X) = \{0, 1, 2\}$*

When throwing a fair die, see Example 7, if X denotes the outcomes $\in \{1, 2, 3, 4, 5, 6\}$, the expectation of X is given by

$$\begin{aligned} E[X] &= 1 \cdot P(X = 1) + 2 \cdot P(X = 2) + 3 \cdot P(X = 3) + 4 \cdot P(X = 4) \\ &\quad + 5 \cdot P(X = 5) + 6 \cdot P(X = 6) \\ &= 1 \cdot \frac{1}{6} + 2 \cdot \frac{1}{6} + 3 \cdot \frac{1}{6} + 4 \cdot \frac{1}{6} + 5 \cdot \frac{1}{6} + 6 \cdot \frac{1}{6} \\ &= 3.5 \end{aligned}$$

which is also the average value of $Values(X) = \{1, 2, 3, 4, 5, 6\}$

These examples show that, when running a random experiment, if the sample points are equally likely, the weighted average becomes the simple average. Now suppose in the second experiment that the outcomes of the die are not equally likely, say $P(X = 6) = 0.5$ and $P(X = x) = 0.1$ for $x < 6$. We have

$$\begin{aligned} E[X] &= 1 \cdot 0.1 + 2 \cdot 0.1 + 3 \cdot 0.1 + 4 \cdot 0.1 + 5 \cdot 0.1 + 6 \cdot 0.5 \\ &= 4.5 \end{aligned}$$

Let X and Y be two random variables. Among the properties of the expectation, some of the most important are:

1. **Monotonicity**: if $X < Y$, then $E[X] < E[Y]$;
   where $X < Y$ means $f_X(\omega) < f_Y(\omega)\ \forall \omega \in \Omega$
2. **Linearity**:
   - $E[X + Y] = E[X] + E[Y]$. Which is true even if X and Y are not independent;
   - $E[a \cdot X + b] = a \cdot E[X] + b$ where a and b are constants;
3. **Non-multiplicativity**: $E[X \cdot Y] = E[X] \cdot E[Y]$ only if X and Y are independent;

Given some evidence e, the conditional expectation of a random variable X given e is given by the following equation:

$$E[X|e] = \sum_x x \cdot P(x|e) \tag{6.5}$$

where x are the possible values of X.

Part III

Probabilistic Logic Programming

Probabilistic Programming (PP) has recently emerged as an effective approach for building complex probabilistic models. Until recently, PP was mostly focused on functional programming while Probabilistic Logic Programming (PLP) now forms a significant subfield. In this part of the thesis, we aim at presenting a quick overview of the features of current languages and systems for PLP. We first describe PLP and the underlining languages in Chapter 7 and present some applications in real domains in Chapter 8.

Chapter 7

Languages under the Distribution Semantics

Probabilistic Logic Programming (PLP) [28] models domains characterized by complex and uncertain relationships among entities by combining logic programming, Chapter 5, with probability theory, Chapter 6. The field started in the early nineties with the seminal work of Dantsin [21], Poole [99] and Sato [121] and is now well established, with a dedicated annual workshop since 2014. PLP has been applied successfully to many problems such as concept relatedness in biological networks [29], Mendel's genetic inheritance [123], natural language processing [125], link prediction in social networks [75], entity resolution [110] and model checking [40]. Various approaches have been proposed for representing probabilistic information in Logic Programming. The distribution semantics [121] is one of the most used and it underlies many languages.

We start this chapter by presenting the distribution semantics in Section 7.1. Then Sections 7.2 and 7.3 present two common Languages under the distribution semantic, Logic Programs with Annotated Disjunctions (LPADs) and ProbLog respectively. For LPADs, we present a semantics for programs without function symbols and then discuss the extensions of this semantics for programs including function symbols, that may have infinite computation branches. Programs including continuous random variables, a recent proposal that brought PLP closer to functional PP approaches are also presented. Section 7.4 discusses approaches for inference starting from exact inference by knowledge compilation and going to techniques for dealing with infinite computation branches and continuous random variables. We also illustrate approximate inference approaches based on Monte Carlo methods that can overcome some of the limit of exact inference both in terms of computation time and allowed language features, permitting inference on a less restricted class of programs.

7.1 Distribution Semantics

Combining Logic Programming and probability is an interesting challenge in the field of Statistical Relational Learning. This combination allows not only to represent a wide range of real domains characterized by uncertainty but also to easily perform reasoning and learning in these domains. Two approaches for combining logic and probabilities emerged: those based on *Distribution Semantics* (DS) and those based on *Knowledge Base Model Construction* (KBMC). In KBMC, a probabilistic logic program is converted into graphical models such as Bayesian or Markov Network and inference/learning are performed on them. These languages include Bayesian Logic Programs [55], Probabilistic Knowledge Bases [86], Prolog Factor Language [37], and many others.

The Distribution Semantics, the one used in this work defines a probability distribution over normal logic programs, called *worlds* or *instances*. The distribution is extended to a join distribution over worlds and queries and the probability of a query is computed by summing out the worlds, i.e by marginalization. DS was introduced in 1995 by Taisuke Sato [121] in which a semantic combining *definite logic programs* and probabilistic facts were presented. For a deep understanding of DS see [121].

Many languages under the DS have been presented among which: Independent Choice Logic [98], PRISM [123], Logic Programs with Annotated Disjunctions (LPADs) [143] and ProbLog [29], to name a few. While these languages differ syntactically, they have the same expressive power, as there are linear transformations among them [142]. In the following sections we respectively present LPADs, for its generality, and ProbLog, for its simplicity.

7.2 Logic Programs with Annotated Disjunctions

All the languages following the distribution semantics allow the specification of alternatives either on facts and/or on clauses. We present here the syntax of LPADs [143] which allows alternatives on clauses.

An LPAD is a finite set of annotated disjunctive clauses of the form

$$h_{i1} : \Pi_{i1}; \ldots ; h_{in_i} : \Pi_{in_i} \text{ :- } b_{i1}, \ldots , b_{im_i}.$$

where $b_{i1}, \ldots , b_{im_i}$ are literals, $h_{i1}, \ldots h_{in_i}$ are atoms and $\Pi_{i1}, \ldots , \Pi_{in_i}$ are real numbers in the interval $[0, 1]$ such that $\sum_{k=1}^{n_i} \Pi_{ik} \leq 1$. This clause can be interpreted as "if $b_{i1}, \ldots , b_{im_i}$ is true, then h_{i1} is true with probability Π_{i1} or ...or h_{in_i} is true with probability Π_{in_i}." $b_{i1}, \ldots , b_{im_i}$ is the body of the clause and we indicate it with *body*(C) if the clause is C. If $n_i = 1$ and $\Pi_{i1} = 1$ the clause is non-disjunctive and so not probabilistic. If $\sum_{k=1}^{n_i} \Pi_{ik} < 1$, there is an implicit annotated atom $\mathit{null} : (1 - \sum_{k=1}^{n_i} \Pi_{ik})$ that does not appear in the body of any clauses of the program.

Given an LPAD P, the grounding *ground*(P) is obtained by replacing variables with all possible logic terms. If P does not contain function symbols, the set of possible terms is equal to the set of all constants appearing in P and is finite so *ground*(P) is finite as well.

ground(P) is still an LPAD from which, by selecting a head atom for each ground clause, we can obtain a normal logic program, called "world". In the distribution semantics, the choices of head atoms for different clauses are independent, so we can assign a probability to a world by multiplying the probabilities of all the head atoms chosen to form the world. In this way we get a probability distribution over worlds from which we can define a probability distribution over the truth values of a ground atom: the probability of an atom q being true is the sum of the probabilities of the worlds where q is true in the well-founded model [140] of the world.

The well-founded model [140] in general is three valued, so a query that is not true in the model is either undefined or false. However, we consider atoms as Boolean random variables so we do not want to deal with the undefined truth value. How to manage uncertainty by combining nonmonotonic reasoning and probability theory is still an open problem, see [18] for a discussion of the issues. So we require each world to have a two-valued well-founded model and therefore q can only be true or false in a world.

Formally, each grounding of a clause $C_i\theta_j$ corresponds to a random variable X_{ij} with as many values as the number of head atoms of C_i. The random variables X_{ij} are independent of each other.

An *atomic choice* [97] is a triple (C_i, θ_j, k) where $C_i \in P$, θ_j is a substitution that grounds C_i and $k \in \{1, \dots, n_i\}$ identifies one of the head atoms. In practice $C_i\theta_j$ corresponds to an assignment $X_{ij} = k$. A set of atomic choices κ is *consistent* if only one head is selected for the same ground clause. A consistent set κ of atomic choices is called a *composite choice*. We can assign a probability to κ as the random variables are independent:

$$P(\kappa) = \prod_{(C_i,\theta_j,k)\in\kappa} \Pi_{ik}$$

A *selection* σ is a composite choice that, for each clause $C_i\theta_j$ in $ground(P)$, contains an atomic choice (C_i, θ_j, k). A selection σ identifies a normal logic program l_σ defined as $l_\sigma = \{(h_{ik} \leftarrow body(C_i))\theta_j | (C_i, \theta_j, k) \in \sigma\}$. l_σ is called an *instance*, *possible world* or simply *world* of P. Since selections are composite choices, we can assign a probability to instances: $P(l_\sigma) = P(\sigma) = \prod_{(C_i,\theta_j,k)\in\sigma} \Pi_{ik}$.

We write $l_\sigma \models q$ to mean that the query q (a ground atom) is true in the well-founded model of the program l_σ. The probability of a query q given a world l_σ can be now defined as $P(q|l_\sigma) = 1$ if $l_\sigma \models q$ and 0 otherwise. Let $P(L_P)$ be the distribution over worlds. The probability of a query q is given by

$$P(q) = \sum_{l_\sigma \in L_P} P(q, l_\sigma) = \sum_{l_\sigma \in L_P} P(q|l_\sigma)P(l_\sigma) = \sum_{l_\sigma \in L_P : l_\sigma \models q} P(l_\sigma) \tag{7.1}$$

Example 13 (From [11]). *The following LPAD P encodes geological knowledge on the Stromboli Italian island:*

$$\begin{aligned}
C_1 &= eruption : 0.6 \;;\; earthquake : 0.3 :- \; sudden\_energy\_release, \\
&\quad\; fault\_rupture(X). \\
C_2 &= sudden\_energy\_release : 0.7. \\
C_3 &= fault\_rupture(southwest\_northeast). \\
C_4 &= fault\_rupture(east\_west).
\end{aligned}$$

The Stromboli island is located at the intersection of two geological faults, one in the southwest-northeast direction, the other in the east-west direction, and contains a active volcano. This program models the possibility that an eruption or an earthquake occurs at Stromboli. If there is a sudden energy release under the island and there is a fault rupture, then there can be an eruption of the volcano on the island with probability 0.6 or an earthquake in the area with probability 0.3 or no event with probability 0.1. The energy release occurs with probability 0.7 while we are sure that ruptures occur in both faults.

Clause C_1 has two groundings, $C_1\theta_1$ with $\theta_1 = \{X/southwest\_northeast\}$ and $C_1\theta_2$ with $\theta_2 = \{X/east\_west\}$, so there are two random variables X_{11} and X_{12}. Clause C_2 has only one grounding $C_2\emptyset$ instead, so there is one random variable X_{21}. X_{11} and X_{12} can take three values since C_1 has three head atoms; similarly X_{21} can take two values since C_2 has two head atoms. P has 18 instances, the query eruption is true in 5 of them and its probability is
$P(eruption) = 0.6{\cdot}0.6{\cdot}0.7 + 0.6{\cdot}0.3{\cdot}0.7 + 0.6{\cdot}0.1{\cdot}0.7 + 0.3{\cdot}0.6{\cdot}0.7 + 0.1{\cdot}0.6{\cdot}0.7 = 0.588$.

To compute the conditional probability $P(q|e)$ of a query q given evidence e, you can use the definition of conditional probability, $P(q|e) = P(q, e)/P(e)$, and compute first the probability of q, e (the sum of probabilities of worlds where both q and e are true) and the probability of e and then divide the two.

7.2.1 Sampling interpretation of the semantics

The semantics previously described can also be given a sampling interpretation: the probability of a query q is the fraction of worlds, sampled from the distribution over worlds, where q is true. To sample from the distribution over worlds, you simply randomly select a head atom for each ground clause according to the probabilistic annotations. Note that you do not even need to sample a complete world: if the samples you have taken ensure the truth value of q is determined, you don't need to sample more clauses, as they don't influence q. If q is true in n_{True} worlds out of n sampled, then its probability can be approximated by

$$P(q) = \hat{p} = \frac{n_{True}}{n}$$

7.2.2 Programs with function symbols

If the program P contains function symbols, a more complex definition of the semantics is necessary. In fact *ground*(P) is infinite and a world would be obtained by making an infinite number of choices so its probability would be 0, as it is a product of infinite numbers all bounded away from 1 from below. In this case we have to work with sets of worlds and use Kolmogorov's definition of probability space. It turns out that the probability of the query is the sum of a convergent series [111].

7.2.3 Hybrid programs

Up to now we have considered only discrete random variables and discrete probability distributions. How can we consider continuous random variables and probability density functions, for example real variables following a Gaussian distribution? `cplint` [2] and Distributional Clauses (DC) [91] allow the description of continuous random variables in so called *hybrid programs*.

`cplint` allows the specification of density functions over arguments of atoms in the head of rules. For example, in

```
g(X,Y): gaussian(Y,0,1):- object(X).
```

`X` takes terms while `Y` takes real numbers as values. The clause states that, for each `X` such that `object(X)` is true, the values of `Y` such that `g(X,Y)` is true, follow a Gaussian distribution with mean 0 and variance 1. You can think of an atom such as `g(a,Y)` as an encoding of a continuous random variable associated to term `g(a)`. In DC you can express the same density as

```
g(X)~gaussian(0,1):= object(X).
```

where `:=` indicates implication and continuous random variables are represented as terms that denote a value from a continuous domain. It is possible to translate DC into programs for `cplint` and in fact `cplint` allows also the DC syntax, automatically translating DC into its own syntax.

A semantics for hybrid programs was given independently in [42, 91] and [50]. In [91] the semantics of Hybrid Probabilistic Logic Programs is defined by means of a stochastic generalization STp of the Tp operator that applies the sampling interpretation of the distribution semantics to continuous variables: STp is applied to interpretations that contain ground atoms (as in standard logic programming) and terms of the form $t = v$ where t is a term indicating a continuous random variable and v is a real number. If the body of a clause is true in an interpretation I, $STp(I)$ will contain a sample from the head.

The authors of [50] define a probability space for N continuous random variables by considering the Borel σ-algebra over $\mathbb{R}^N$ and fixing a Lebesgue measure on this set as the probability measure. The probability space is lifted to cover the entire program using the least model semantics of constraint logic programs.

If an atom encodes a continuous random variable (such as `g(X,Y)` above), asking for the probability that a ground instantiation, such as `g(a,0.3)`, is true is not meaningful, as the probability that a continuous random variables takes a specific value is always 0. In this case you want to compute the probability that the random variable falls in an interval or you want to know its density, possibly after having observed some evidence. If the evidence is on an atom defining another continuous random variable, the definition of conditional probability cannot be applied, as the probability of the evidence would be 0 and so the fraction would be undefined. This problem is tackled in [91] by providing a definition using limits.

7.3 ProbLog

In ProbLog [60], a program is a set of *normal rules* and *probabilistic facts*. Each probabilistic fact, f_i, is annotated with a probability π_i and is of the following form

$$\pi_i :: f_i$$

where $\pi_i \in [0, 1]$ and f_i is an atom. This probabilistic fact states that each ground instantiation $f_i\theta$ of f_i is true with probability π_i and false with probability $1 - \pi_i$. In order to obtain a world, some instantiations of probabilistic facts are selected. An *atomic choice* is a triple (f, θ, k) where $k \in \{0, 1\}$. If $k = 1$, the fact is selected. Otherwise ($k = 0$) the fact is not selected. A set κ of atomic choices is *consistent* if only one alternative is selected for a ground probabilistic fact, i.e. does not contain two atomic choices (f, θ, k) and (f, θ, j) with $k \neq j$. A *composite choice* is a consistent set of atomic choices. As in LPADs, the probability of composite choice κ is equal to the probability of the conjunction of many random variables and is given by the product

$$P(\kappa) = \prod_{(f_i,\theta,1)\in\kappa} \pi_i \prod_{(f_i,\theta,0)\in\kappa} 1 - \pi_i.$$

since the selection of an atomic choice does not depend on the selection of the others.

A *selection* σ contains one atomic choice for every grounding of every probabilistic fact. It is a total composite choice. A *world* or *instance* is a logic program containing the facts identified by a selection σ plus the normal rules. The world w is formed by including the atom corresponding to each atomic choice $(f, \theta, 1)$ of σ. The probability of a world w is $P(w) = P(\sigma)$. For programs without function symbols, the set of groundings of each probabilistic fact is finite, and so is the set of worlds. P(w) is a probability distribution over worlds, i.e., $\sum_w P(w) = 1$

The probability of a query q is computed by marginalization as done for LPADs in Section 7.2.

$$P(q) = \sum_w P(q, w) = \sum_w P(q|w)P(w) = \sum_{w\models q} P(w). \qquad (7.2)$$

This semantics can also be given a sampling interpretation as described in Section 7.2.1.

Translating LPAD to ProbLog

There is a linear translation among languages under the distribution semantic. In order to illustrate a translation and also the computation time necessary for translating, let us present how to convert a program from LPAD to ProbLog as described by De Raedt et al., in [26]. Consider an LPAD clause C_i having a set of variables $\mathbf{X}$ with the following form

$$H_1 : p_1; H_2 : p_2 \ldots H_n : p_n : -B.$$

C_i can be converted into the following set of rules and probabilistic facts

$H_1 : -B, f_{i,1}(\mathbf{X}).$
$H_2 : -B, not(f_{i,1}(\mathbf{X})), f_{i,2}(\mathbf{X}).$
$\vdots$
$H_2 : -B, not(f_{i,1}(\mathbf{X})), \ldots, not(f_{i,n-1}(\mathbf{X})).$

$\pi_1 : -f_{i,1}(\mathbf{X}).$
$\vdots$
$\pi_{n-1} : -f_{i,n-1}(\mathbf{X}).$
where $\pi_1 = p_1$, $\pi_2 = \frac{p_2}{1-\pi_1}$, $\pi_3 = \frac{p_3}{(1-\pi_1)\cdot(1-\pi_2)}$, ..., $\pi_i = \frac{p_i}{\prod_j^{i-1}(1-\pi_j)}$.

This conversion illustrates the linear translation among languages under the distribution semantics.

7.4 Inference in Probabilistic Logic Programming

Computing all the worlds is impractical because their number is exponential in the number of ground probabilistic clauses when there are no function symbols and impossible otherwise, because with function symbols the number of worlds is uncontably infinite. Alternative approaches for inference have been considered that can be grouped in exact and approximate ones [2].

For exact inference from discrete program without function symbols a successful approach finds explanations for the query q [29], where an explanation is a set of composite choices that are sufficient for entailing the query. Once all explanations for the query are found, they are encoded as a Boolean formula in DNF and the problem is reduced to that of computing the probability that the formula is true given the probabilities of being true of all the (mutually independent) random variables. This problem is called *disjoint-sum* as it can be solved by finding a DNF where all the disjuncts are mutually exclusive. Its complexity is #P [138] so the problem is highly difficult and intractable in general. In practice, problems of significant size can be tackled using *knowledge compilation* [23], i.e. converting the DNF into a language from which the computation of the probability is polynomial [29, 116], such as Binary Decision Diagrams.

Formally, a composite choice κ is an *explanation* for a query q if q is entailed by every instance consistent with κ, where an instance l_σ is consistent with κ iff

$\kappa \subseteq \sigma$. Let λ_κ be the set of worlds consistent with κ. In particular, algorithms find a covering set of explanations for the query, where a set of composite choices K is *covering* with respect to q if every program l_σ in which q is entailed is in λ_K, where $\lambda_K = \sum_{\kappa \in K} \lambda_\kappa$. The problem of computing the probability of a query q can thus be reduced to computing the probability of the Boolean function

$$f_q(\mathbf{X}) = \bigvee_{\kappa \in E(q)} \bigwedge_{(C_i, \theta_j, k) \in \kappa} X_{ij} = k \tag{7.3}$$

where $E(q)$ is a covering set of explanations for q.

Example 14 (Example 13 cont.). *The query eruption has the covering set of explanations $E(eruption) = \{\kappa_1, \kappa_2\}$ where:*

$$\begin{array}{rcl}\kappa_1 & = & \{(C_1, \{X/southwest\_northeast\}, 1), (C_2, \{\}, 1)\} \\ \kappa_2 & = & \{(C_1, \{X/east\_west\}, 1), (C_2, \{\}, 1)\}\end{array}$$

Each atomic choice (C_i, θ_j, k) is represented by the propositional equation $X_{ij} = k$:

$$\begin{array}{lcl}(C_1, \{X/southwest\_northeast\}, 1) & \rightarrow & X_{11} = 1 \\ (C_1, \{X/east\_west\}, 1) & \rightarrow & X_{12} = 1 \\ (C_2, \{\}, 1) & \rightarrow & X_{21} = 1\end{array}$$

The resulting Boolean function $f_{eruption}(\mathbf{X})$ returns 1 if the values of the variables correspond to an explanation for the goal. Equations for a single explanation are conjoined and the conjunctions for the different explanations are disjoined. The set of explanations $E(eruption)$ can thus be encoded with the function:

$$f_{eruption}(\mathbf{X}) = (X_{11} = 1 \wedge X_{21} = 1) \vee (X_{12} = 1 \wedge X_{21} = 1) \tag{7.4}$$

Examples of systems that perform inference using this approach are ProbLog [61] and PITA [116, 117].

When a discrete program contains function symbols, the number of explanations may be infinite and the probability of the query may be the sum of a convergent series. In this case the inference algorithm has to recognize the presence of an infinite number of explanations and identify the terms of the series.

In [40] the authors present the algorithm PIP (for Probabilistic Inference Plus), that is able to perform inference even when explanations are not necessarily mutually exclusive and the number of explanations is infinite. They require the programs to be *temporally well-formed*, i.e., that one of the arguments of predicates can be interpreted as a time that grows from head to body. In this case the explanations for an atom can be represented succinctly by Definite Clause Grammars (DCGs). Such DCGs are called *explanation generators* and are used to build Factored Explanation Diagrams (FED) that have a structure that closely follows that of Binary Decision Diagrams. FEDs can be used to obtain a system of polynomial equations that is monotonic and thus convergent as in [126, 120]. So, even when the system is non linear, a least solution can be computed to within an arbitrary approximation bound by an iterative procedure.

For approximate inference one of the most used approach consists in Monte Carlo sampling, following the sampling interpretation of the semantics given in Section 7.2.1. Monte Carlo approach has been implemented in ProbLog [61] and MCINTYRE [109] and found to give good performance in terms of quality of the solutions and of running time. Monte Carlo sampling is attractive for the simplicity of its implementation and because you can improve the estimate as more time is available. Moreover, Monte Carlo can be used also for programs with function symbols, in which goals may have infinite explanations and exact inference may loop. In fact, taking a sample of a query corresponds naturally to an explanation and the probability of a derivation is the same as the probability of the corresponding explanation. The risk is that of incurring in an infinite explanation. But infinite explanations have probability 0 so the probability that the computation goes down such a path and does not terminate is 0 as well.

Monte Carlo inference provides also smart algorithms for computing conditional probabilities: rejection sampling or Metropolis-Hastings Markov Chain Monte Carlo (MCMC). In rejection sampling [147], you first query the evidence and, if the query is successful, query the goal in the same sample, otherwise the sample is discarded. In Metropolis-Hastings MCMC [82], a Markov chain is built by taking an initial sample and by generating successor samples. The initial sample is built by randomly sampling choices so that the evidence is true. A successor sample is obtained by deleting a fixed number of sampled probabilistic choices. Then the evidence is queried again by sampling starting with the undeleted choices. If the query succeeds, the goal is then also queried by sampling. The goal sample is accepted with a probability of $\min\{1, \frac{N_0}{N_1}\}$ where N_0 is the number of choices sampled in the previous sample and N_1 is the number of choices sampled in the current sample. The number of successes of the query is increased by 1 if the query succeeded in the last accepted sample. The final probability is given by the number of successes over the total number of samples.

When you have evidence on ground atoms that have continuous values as arguments, you can still use Monte Carlo sampling. You cannot use rejection sampling or Metropolis-Hastings, as the probability of the evidence is 0, but you can use likelihood weighting [91] to obtain weighted samples of continuous arguments of a goal. For each sample to be taken, likelihood weighting samples the query and then assigns a weight to the sample on the basis of evidence. The weight is computed by deriving the evidence backward in the same sample of the query starting with a weight of one: each time a choice should be taken or a continuous variable sampled, if the choice/variable has already been taken, the current weight is multiplied by probability of the choice/by the density value of the continuous value.

If likelihood weighting is used to find the posterior density of a continuous random variable, we obtain a set of weighted samples for the variables whose weight that can be interpreted as a relative frequency. The set of samples without the weight, instead, can be interpreted as the prior density of the variable. These two set of samples can be used to plot the density before and after observing the evidence.

You can sample arguments of queries also for discrete goals: in this case you get a discrete distribution over the values of one or more arguments of a goal. If the query predicate is determinate in each world, i.e., given values for input arguments there is a single value for output arguments that make the query true, for each sample you get a single value. Moreover, if clauses sharing an atom in the head are mutually exclusive, i.e., in each world the body of at most one clause is true, then the query defines a probability distribution over output arguments. In this way we can simulate those languages such as PRISM and Stochastic Logic Programs [80] that define probability distributions over arguments rather than probability distributions over truth values of ground atoms.

Chapter 8

Probabilistic Logic Programming in action

We have presented in the previous chapter how to combine logic and probabilities and stated that this combination can model various useful domains especially those characterized by uncertainty. In order to illustrate this combination, we present in this chapter several examples and domains modeled by PLP. The chapter is organized as follows: In the first two examples, tile map generation and Markov Logic Networks encoding presented in sections 8.1 and 8.4 respectively, all the variables are discrete and no infinite computation path exists. The next three problems have infinite computation paths: the truel game, the coupon collector problem and the one-dimensional random walk described in sections from 8.3 to 8.5 respectively. The last two examples, latent Dirichlet allocation and the Indian GPA problem presented respectively in sections 8.6 and 8.7 include continuous variables. These examples show the maturity of PLP.

8.1 Tile map generation

PP and PLP can be used to generate random complex structures. For example, we can write programs for randomly generating maps of video games. We are given a fixed set of tiles that we want to combine to obtain a 2D map that is random but satisfies some soft constraints on the placement of tiles.

Suppose we want to draw a 10x10 map with a tendency to have a lake in the center. The tiles are randomly placed such that, in the central area, water is more probable. The problem can be modeled with the following program [1], where `map(H,W,M)` instantiates `M` to a map of height `H` and width `W`:

```
map(H,W,M):-
  tiles(Tiles),
  length(Rows,H),
  M=..[map,Tiles|Rows],
  foldl(select(H,W),Rows,1,_).

select(H,W,Row,N0,N):-
  length(RowL,W),
  N is N0+1,
  Row=..[row|RowL],
  foldl(pick_row(H,W,N0),RowL,1,_).

pick_row(H,W,N,T,M0,M):-
  M is M0+1,
  pick_tile(N,M0,H,W,T).
```

where `foldl/4` is a SWI-Prolog [151] library predicate that implements the `foldl` meta primitive from functional programming. `pick_tile(Y,X,H,W,T)` returns in `T` a tile for position `(X,Y)` of a map of size `W*H`. The center tile is water:

```
pick_tile(HC,WC,H,W,water):-
  HC is H//2,
  WC is W//2,!.
```

In the central area water is more probable:

```
pick_tile(Y,X,H,W,T):
  discrete(T,[grass:0.05,water:0.9,tree:0.025,rock:0.025]):-
  central_area(Y,X,H,W),!
```

`central_area(Y,X,H,W)` is true if `(X,Y)` is adjacent to the center of the `W*H` map (definition omitted for brevity). In the other places, tiles are chosen at random with distribution `[grass:0.5,water:0.3,tree:0.1,rock:0.1]`:

```
pick_tile(_,_,_,_,T):discrete(T,[grass:0.5,water:0.3,tree:0.1,rock:0.1]).
```

We can generate a map by taking a sample of the query `map(10,10,M)` and collecting the value of `M`. For example, the map of Figure 8.1 can be obtained[2].

[1] `http://cplint.lamping.unife.it/example/inference/tile_map.swinb`

[2] Tiles from `https://github.com/silveira/openpixels`

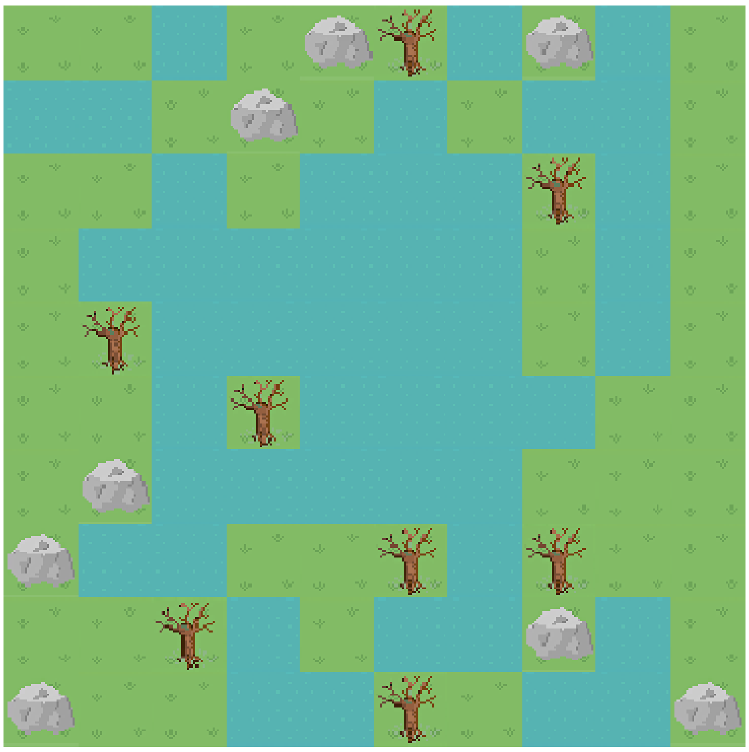


Figure 8.1: A random tile map.

8.2 Markov Logic Networks

Markov Networks (MN) and Markov Logic Networks (MLN) [107] can be encoded with PLP. The encoding is based on the observation that a MN factor can be represented with a Bayesian Network (BN) with an extra node that is always observed. Since PLP programs under the distribution semantics can encode BN [143], we can encode MLN. For example, the MLN clause

$$1.5\ Intelligent(x) \Rightarrow GoodMarks(x)$$

where 1.5 is the weight of the clause.

for a single constant *anna* originates an edges between the Boolean nodes for $Intelligent(anna)$ and $GoodMarks(anna)$. This means that the two variables cannot be d-separated in any way. This dependence can be modeled with BN by adding and extra Boolean node, $Clause(anna)$, that is a child of $Intelligent$ $(anna)$ and $GoodMarks(anna)$ and is observed. In this way, $Intelligent(anna)$ and $GoodMarks(anna)$ are not d-separated in the BN no matter what other nodes the BN contains.

In general, for a domain with Herbrand base X and an MLN ground clause C mentioning atom variables X', the equivalent BN should contain a Boolean node C with X' as parents. All the query of the form $P(a|b)$ should then be posed to the BN as $P(a|b, C = true)$. The problem is now how to assign values to the conditional probability (CPT) of C given X' so that the joint distribution of X in the BN is the same as that of the MLN.

An MLN formulae of the form $\alpha\ C$ contributes to the probabilities of the worlds with a factor e^{α} for the worlds where the clauses is true and 1 for the worlds where the clause is false. If we use c to indicate $C = true$, the joint probability of a state of the world x can then be computed as

$$P(x|c) = \frac{P(x, c)}{P(c)} \propto P(x, c)$$

i.e $P(x|c)$ is proportional to $P(x, c)$, because the denominator does not depend on x and is thus a normalizing constant.

$P(x, c)$ can be written as

$$P(x, c) = P(c|x)P(x) = P(c|x')P(x)$$

where x' is the state of the parents of C, so

$$P(x|c) \propto P(c|x')P(x)$$

To model the MLN formula we just have to ensure that $P(c|x')$ is proportional to e^{α} when x' makes C true and to 1 when x' makes C false. We cannot use e^{α} directly in the CPT for C because it can be larger than 1 but we can use the values $e^{\alpha}/(1 + e^{\alpha})$ and $1/(1 + e^{\alpha})$ that are proportional to e^{α} and 1 and are surely less than 1.

For an MLN containing the example formula above, the probability of a world would be represented by $P(i, g|c)$ where i and g are values for $Intelligent(anna)$ and $GoodMarks(anna)$ and c is $Clause(anna) = true$. The CPT will have the values $e^{1.5}/(1 + e^{1.5})$ for $Clause(anna)$ being true given that the parents' values make the clause true and $1/(1 + e^{1.5})$ for $Clause(anna)$ being true given that the parents' values make the clause false.

In order to model MLN formulas with LPADs, we can add an extra atom $clause_i(X)$ for each formula $F_i = \alpha_i\ C_i$ where X is the vector of variables appearing in C_i. Then, when we query for the probability of query q given evidence e, we have to ask for the probability of q given $e \wedge ce$ where ce is the conjunction of all the groundings of $clause_i(X)$ for all values of i. Then, clause C_i should be transformed into a Disjunctive Normal Form (DNF) formula $C_{i1} \vee \ldots \vee C_{in_i}$ where the disjuncts are mutually exclusive and the LPAD should contain the clauses

$$clause_i(X) : e^{\alpha}/(1 + e^{\alpha}) \leftarrow C_{ij}$$

for all j. Similalry, $\neg C_i$ should be transformed into a disjoint sum $D_{i1} \vee \ldots \vee D_{im_i}$ and the LPAD should contain the clauses

$$clause_i(X) : 1/(1 + e^{\alpha}) \leftarrow D_{il}$$

for all l.

Alternatively, if α is negative, e^{α} will be smaller than 1 and we can use the two probability values e^{α} and 1 with the clauses

$$\begin{array}{l} clause_i(X) : e^{\alpha} \leftarrow C_{ij} \\ \ldots \\ clause_i(X) \leftarrow D_{il} \end{array}$$

This solution has the advantage that some clauses are certain, reducing the number of random variables. If α is positive in formula $\alpha\ C$, we can consider $-\alpha\ \neg C$.

MLN formulas can also be added to a regular probabilistic logic program. In this case their effect is equivalent to a soft form of evidence, where certain worlds are weighted more than others. This is the same as soft evidence in Figaro [95]. MLN hard constraints, i.e., formulas with an infinite weight, can instead be used to rule out completely certain worlds, those violating the constraint. For example, given hard constraint C equivalent to the disjunction $C_{i1} \vee \ldots \vee C_{in_i}$, the LPAD should contain the clauses

$$clause_i(X) \leftarrow C_{ij}$$

for all j, and the evidence should contain $clause_i(x)$ for all groundings x of X. In this way, the worlds that violate C are ruled out. Let see an example[3] where we translate the MLN

```
1.5 Intelligent(x) => GoodMarks(x)
1.1 Friends(x, y) => (Intelligent(x) <=> Intelligent(y))
```

The first MLN formula is translated into

```
clause1(X): 0.8175744762:- \+intelligent(X).
clause1(X): 0.1824255238:- intelligent(X), \+good_marks(X).
clause1(X): 0.8175744762:- intelligent(X), good_marks(X).
```

where $0.8175744762 = e^{1.5}/(1 + e^{1.5})$ and $0.1824255238 = 1/(1 + e^{1.5})$.

The MLN formula

```
1.1 Friends(x, y) => (Intelligent(x) <=> Intelligent(y))
```

is translated into the clauses

```
clause2(X,Y): 0.7502601056:-
  \+friends(X,Y).
clause2(X,Y): 0.7502601056:-
  friends(X,Y), intelligent(X),intelligent(Y).
clause2(X,Y): 0.7502601056:-
  friends(X,Y), \+intelligent(X),\+intelligent(Y).
clause2(X,Y): 0.2497398944:-
  friends(X,Y), intelligent(X),\+intelligent(Y).
clause2(X,Y): 0.2497398944:-
  friends(X,Y), \+intelligent(X),intelligent(Y).
```

where $0.7502601056 = e^{1.1}/(1 + e^{1.1})$ and $0.2497398944 = 1/(1 + e^{1.1})$. A priori we have a uniform distribution over student intelligence, good marks and friendship:

```
intelligent(_):0.5.
good_marks(_):0.5.
friends(_,_):0.5.
```

and there are two students:

```
student(anna).
student(bob).
```

[3]http://cplint.lamping.unife.it/example/inference/mln.swinb

The evidence must include the truth of all groundings of the $clause_i$ predicates:

```
evidence_mln:- clause1(anna),clause1(bob),clause2(anna,anna),
    clause2(anna,bob),clause2(bob,anna),clause2(bob,bob).
```

We want to query the probability that Anna gets good marks given that she is fried with Bob and Bob is intelligent, so we define

```
ev_intelligent_bob_friends_anna_bob:-
    intelligent(bob),friends(anna,bob),evidence_mln.
```

and query for $P($`good_marks(anna)`$|$`ev_intelligent_bob_friends_anna_bob`$)$ obtaining 0.7330 which is higher than the prior probability 0.6069 of Anna getting good marks, obtained with the query $P($`good_marks(anna)`$|$`evidence_mln`$)$.

8.3 Truel

A truel [59] is a duel among three opponents. There are three truelists, a, b and c, that take turns in shooting with a gun. The firing order is a, b and c. Each truelist can shoot at another truelist or at the sky (deliberate miss). The truelist have these probabilities of hitting the target (if they are not aiming at the sky): 1/3, 2/3 and 1 for a, b and c respectively. The aim for each truelist is to kill all the other truelists. The question is: what should a do to maximize his probability of winning? Aim at b, c or the sky?

Let us see first the strategy for the other truelists and situations. When only two players are left, the best strategy is to shoot at the other player.

When all three players remain, the best strategy for b is to shoot at c, since if c shoots at him he his dead and if c shoots at a, b remains with c which is the best shooter. Similarly, when all three players remain, the best strategy for c is to shoot at b, since in this way he remains with a, the worst shooter.

For a it is more complex. Let us first compute the probability of a to win a duel with a single opponent. When a and c remain, a wins if it shoots c, with probability 1/3. If he misses c, c will surely kill him. When a and b remain, the probability p of a to win can be computed with

$$\begin{aligned} p &= P(\text{a hits b}) + P(\text{a misses b})P(\text{b misses a})p \\ p &= 1/3 + 2/3 \times 1/3 \times p \\ p &= 3/7 \end{aligned}$$

The probability can be also computed by building the probability tree of Figure 8.2. The probability that a survives is thus

$$\begin{aligned} p &= 1/3 + 2/3 \cdot 1/3 \cdot 1/3 + 2/3 \cdot 1/3 \cdot 2/3 \cdot 1/3 \cdot 1/3 + \ldots = \\ &= 1/3 + 2/3^3 + 2^2/3^5 + \ldots = \frac{1}{3} + \sum_{i=0}^{\infty} \frac{2}{3^3}\left(\frac{2}{9}\right)^i = \frac{1}{3} + \frac{\frac{2}{3^3}}{1-\frac{2}{9}} = \\ &= \frac{1}{3} + \frac{\frac{2}{3^3}}{\frac{7}{9}} = \frac{1}{3} + \frac{\frac{2}{3}}{7} = \frac{1}{3} + \frac{2}{21} = \frac{9}{21} = \frac{3}{7} \end{aligned}$$

When all three players remain, if a shoots at b, b is dead with probability 1/3 but then c will kill a. If b is not dead (probability 2/3), b shoots at c and kills him with probability 2/3. In this case, a is left in a duel with b, with probability of surviving of 3/7. If b doesn't kill c (probability 1/3), c will kill b surely and a is left in a duel with c, with a probability of surviving of 1/3. So overall, if a shoots at b, his probability of winning is

$$2/3 \cdot 2/3 \cdot 3/7 + 2/3 \cdot 1/3 \cdot 1/3 = 4/21 + 2/27 = \frac{36+15}{189} = \frac{50}{189} = 0.2645$$

When all three players remain, if a shoots at c, c is dead with probability 1/3. b then shoots at a and a survives with probability 1/3 and a is then in a duel with b and surviving with probability 3/7. If c survives (probability 2/3), b shoots at c and kills him with probability 2/3, so a remains in a duel with b and wins with probability 3/7. If c survives again, he kills b surely and a is left in a duel with c, with probability 1/3 of winning. So overall, if a shoots at c, his probability of winning is

$$1/3 \cdot 1/3 \cdot 3/7 + 2/3 \cdot 2/3 \cdot 3/7 + 2/3 \cdot 1/3 \cdot 1/3 = 1/21 + 4/21 + 2/27 = 59/189 = 0.3122$$

When all three players remain, if a shoots at the sky, b shoots at c and kills him with probability 2/3, with a remaining in a duel with b. If b doesn't kill c, c will surely kill b and a remains in a duel with c. So overall, if a shoots at the sky, his probability of winning is

$$2/3 \cdot 3/7 + 1/3 \cdot 1/3 = 2/7 + 1/9 = 25/63 = 0.3968$$

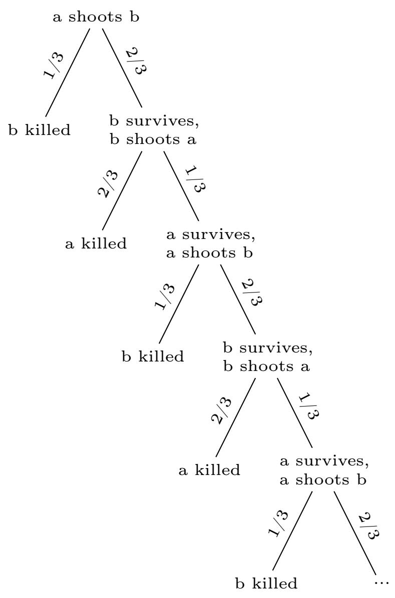


Figure 8.2: Probability tree of the truel with opponents a and b.

This problem can be modeled with an LPAD[4]. However, as can be seen from Figure 8.2, the number of explanations may be infinite so we have to use an appropriate exact inference algorithm or Monte Carlo inference. We discuss below a program that uses MCINTYRE.

`survives_action(A,L0,T,S)` is true if `A` survives truel performing action `S` with `L0` still alive in turn `T`:

```
survives_action(A,L0,T,S):-
  shoot(A,S,L0,T,L1),
  remaining(L1,A,Rest),
  survives_round(Rest,L1,A,T).
```

`shoot(H,S,L0,T,L)` is true when `H` shoots at `S` in round `T` with `L0` and `L` the list of truelists still alive before and after the shot:

```
shoot(H,S,L0,T,L):-
    (S=sky -> L=L0
    ;  (hit(T,H) ->  delete(L0,S,L)
      ;  L=L0
      )
    ).
```

[4]`http://cplint.lamping.unife.it/example/inference/truel.pl`

The probabilities of each truelist to hit the chosen target are

```
hit(_,a):1/3.
hit(_,b):2/3.
hit(_,c):1.
```

`survives(L,A,T)` is true if individual `A` survives the truel with truelists `L` at round `T`:

```
survives([A],A,_):-!.

survives(L,A,T):-
  survives_round(L,L,A,T).
```

`survives_round(Rest,L0,A,T)` is true if individual `A` survives the truel at round `T` with `Rest` still to shoot and `L0` still alive:

```
survives_round([],L,A,T):-
  survives(L,A,s(T)).

survives_round([H|_Rest],L0,A,T):-
  base_best_strategy(H,L0,S),
  shoot(H,S,L0,T,L1),
  remaining(L1,H,Rest1),
  member(A,L1),
  survives_round(Rest1,L1,A,T).
```

These strategies are easy to find:

```
base_best_strategy(b,[b,c],c).
base_best_strategy(c,[b,c],b).
base_best_strategy(a,[a,c],c).
base_best_strategy(c,[a,c],a).
base_best_strategy(a,[a,b],b).
base_best_strategy(b,[a,b],a).
base_best_strategy(b,[a,b,c],c).
base_best_strategy(c,[a,b,c],b).
```

Auxiliary predicate `remaining/3` is defined as

```
remaining([A|Rest],A,Rest):-!.

remaining([_|Rest0],A,Rest):-
  remaining(Rest0,A,Rest).
```

We can decide the best strategy for `a` by asking the queries

```
survives_action(a,[a,b,c],0,b)
survives_action(a,[a,b,c],0,c)
survives_action(a,[a,b,c],0,sky)
```

If we take 1000 samples, possible answers are 0.256, 0.316 and 0.389, showing that `a` should aim at the sky.

8.4 Coupon Collector Problem

The coupon collector problem is described in [51] as

> Suppose each box of cereal contains one of N different coupons and once a consumer has collected a coupon of each type, he can trade them for a prize. The aim of the problem is determining the average number of cereal boxes the consumer should buy to collect all coupon types, assuming that each coupon type occurs with the same probability in the cereal boxes.

If there are N different coupons, how many boxes, T, do I have to buy to get the prize? This problem can be modeled by a program[5] defining predicate `coupons/2` such that `coupons(N,T)` is true if we need `T` boxes to get `N` coupons. We represent the coupons with a term for functor `cp/N` with the number of coupons as arity. The ith argument of the term is 1 if the ith coupon has been collected and is a variable otherwise. The term thus represents an array:

```
coupons(N,T):-
  length(CP,N),
  CPTerm=..[cp|CP],
  new_coupon(N,CPTerm,0,N,T).
```

If 0 coupons remain to be collected, the collection ends:

```
new_coupon(0,_CP,T,_N,T).
```

If `N0` coupons remain to be collected, collect one and recurse:

```
new_coupon(N0,CP,T0,N,T):-
  N0>0,
  collect(CP,N,T0,T1),
  N1 is N0-1,
  new_coupon(N1,CP,T1,N,T).
```

[5]`http://cplint.lamping.unife.it/example/inference/coupon.swinb`

`collect/4` collects one new coupon and updates the number of boxes bought:

```
collect(CP,N,T0,T):-
  pick_a_box(T0,N,I),
  T1 is T0+1,
  arg(I,CP,CPI),
  (var(CPI)-> CPI=1, T=T1
  ; collect(CP,N,T1,T)
  ).
```

`pick_a_box/3` randomly picks a box, an element from the list $[1 \dots N]$:

```
pick_a_box(_,N,I):uniform(I,L):- numlist(1, N, L).
```

If there are 5 different coupons, we may ask:

- how many boxes do I have to buy to get the prize?
- what is the distribution of the number of boxes I have to buy to get the prize?
- what is the expected number of boxes I have to buy to get the prize?

To answer the first query, we can take a single sample for the query `coupons(5,T)`: in the sample, the query will succeed as `coupons/2` is a determinate predicate and the result will instantiate `T` to a specific value. For example, we may get `T=15`. Note that the maximum number of boxes to buy is unbounded but the case where we have to buy an infinite number of boxes has probability 0, so sampling will surely finish.

To compute the distribution on the number of boxes, we can take a number of samples, say 1000, and plot the number of times a value is obtained as a function of the value. We can do so by dividing the domain of the number of boxes in intervals and counting the number of sampled values that fall in each interval. By doing so we may get the graph in Figure 8.3.

To compute the expected number of boxes, we can take a number of samples, say 100, of `coupons(5,T)`. Each sample will instantiate `T`. By summing all these values and dividing the 100, the number of samples, we can get an estimate of the expectation. For example, we may get a value of 11.47.

We can also plot the dependency of the expected number of boxes from the number of coupons, obtaining Figure 8.4. As observed in [51] , the number

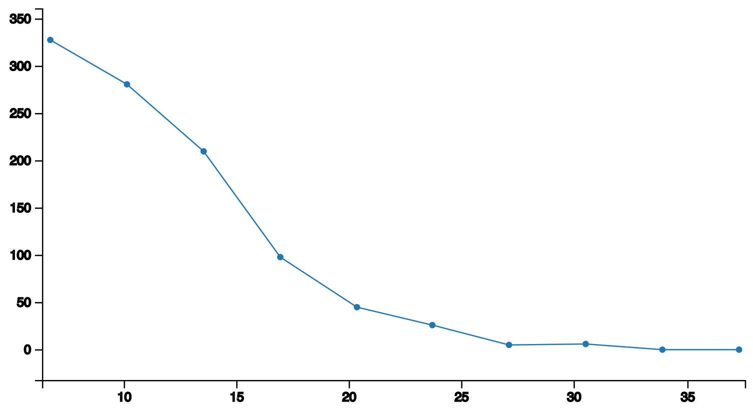


Figure 8.3: Distribution of the number of boxes.

of boxes grows as $O(N \log N)$ where N is the number of coupons. The graph shows the accordance of the two curves.

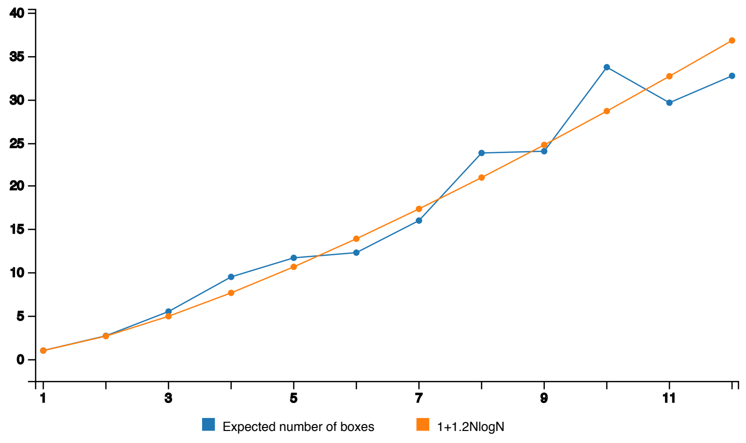


Figure 8.4: Expected number of boxes as a function the number of coupons.

The coupon collector problem is similar to the sticker collector problem, where you have an album with a space for every different sticker, you can buy stickers in packs and your objective is to complete the album. A program for the coupon collector problem can be applied to solve the sticker collector problem: if you have N different stickers and packs contain P stickers, we can solve the coupon collector problem for N coupons and get the number of boxes B. Then the number of packs you have to buy to complete the collection is $\lceil B/P \rceil$. So we can write:

```
stickers(N,P,T):- coupons(N,T0), T is ceiling(T0/P).
```

If there are 50 different stickers and packs contain 4 stickers, by sampling the query `stickers(50,4,T)` we can get `T`=47, i.e., we have to buy 47 packs to complete the entire album.

8.5 One-Dimensional Random Walk

We consider the version of the problem described in [51]: a particle starts at position $x = 10$ and moves with equal probability one unit to the left or one unit to the right in each turn. The random walk stops if the particle reaches position $x = 0$.

The walk terminates with probability one [47] but requires, on average, an infinite time, i.e., the expected number of turns is infinite [51].

We can compute the number of turns with the following program[6]. The walk starts at time 0 and $x = 10$:

```
walk(T):- walk(10,0,T).
```

If x is 0, the walk ends otherwise the particle makes a move:

```
walk(0,T,T).

walk(X,T0,T):-
  X>0,
  move(T0,Move),
  T1 is T0+1,
  X1 is X+Move,
  walk(X1,T1,T).
```

The move is either one step to the left or to the right with equal probability.

```
move(T,1):0.5; move(T,-1):0.5.
```

By sampling the query `walk(T)` we obtain a success as `walk/1` is determinate. The value for `T` represents the number of turns. For example, we may get `T = 3692`.

[6] http://cplint.lamping.unife.it/example/inference/random_walk.swinb

8.6 Latent Dirichlet Allocation

Latent Dirichlet Allocation (LDA) [13] is natural language model which assign topics from a finite set to words in documents. The model describes a generative process where documents are represented as random mixtures over latent topics and each topic defines a distribution over words. LDA assumes the following generative process for a corpus D consisting of M documents each of length N_i:

1. Choose $\theta_i \sim \mathrm{Dir}(\alpha)$, where $i \in \{1, \dots, M\}$ and $\mathrm{Dir}(\alpha)$ is the Dirichlet distribution with parameter α
2. Choose $\varphi_k \sim \mathrm{Dir}(\beta)$, where $k \in \{1, \dots, K\}$
3. For each of the word positions i, j, where $j \in \{1, \dots, N_i\}$, and $i \in \{1, \dots, M\}$
   (a) Choose a topic $z_{i,j} \sim \mathrm{Categorical}(\theta_i)$.
   (b) Choose a word $w_{i,j} \sim \mathrm{Categorical}(\varphi_{z_{i,j}})$.

This is a smoothed LDA model to be precise. The subscript is often dropped, as in the plate diagrams 8.5. The aim is to compute the word probabilities of

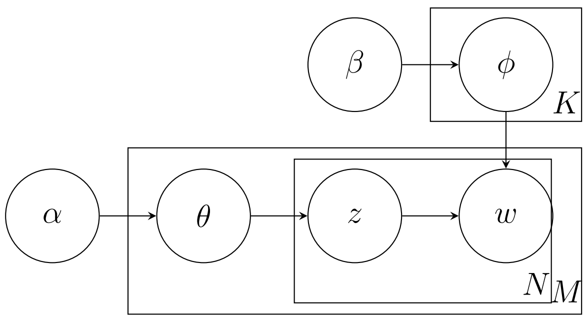


Figure 8.5: Smoothed LDA.

each topic, the topic of each word, and the particular topic mixture of each document. This can be done with Bayesian inference: the documents in the dataset represent the observations (evidence) and we want to compute the posterior distribution of the above quantities.

This problem can modeled by the MCINTYRE program[7] below, where predicate

```
word(Doc,Position,Word)
```

indicates that document `Doc` in position `Position` (from 1 to the number of words of the document) has word `Word` and predicate

```
topic(Doc,Position,Topic)
```

indicates that document `Doc` associates topic `Topic` to the word in position `Position`. We also assume that the distributions for both θ_m and φ_k are symmetric Dirichlet distributions with scalar concentration parameter η set using a fact for the predicate `eta/1`, i.e., $\alpha = \beta = [\eta, \dots, \eta]$. The program is then:

```
theta(_,Theta):dirichlet(Theta,Alpha):-
  alpha(Alpha).

topic(DocumentID,_,Topic):discrete(Topic,Dist):-
  theta(DocumentID,Theta),
  topic_list(Topics),
  maplist(pair,Topics,Theta,Dist).

word(DocumentID,WordID,Word):discrete(Word,Dist):-
  topic(DocumentID,WordID,Topic),
  beta(Topic,Beta),
  word_list(Words),
  maplist(pair,Words,Beta,Dist).

beta(_,Beta):dirichlet(Beta,Parameters):-
  n_words(N),
  eta(Eta),
  findall(Eta,between(1,N,_),Parameters).

alpha(Alpha):-
  eta(Eta),
  n_topics(N),
  findall(Eta,between(1,N,_),Alpha).

eta(2).

pair(V,P,V:P).
```

[7] `http://cplint.lamping.unife.it/example/inference/lda.swinb`

where `maplist/4` is a library of SWI-Prolog encoding the `maplist` primitive of functional programming. Suppose we have two topics, indicated with integers 1 and 2, and 10 words, indicated with integers $1, \ldots, 10$:

```
topic_list(L):-
  n_topics(N),
  numlist(1,N,L).

word_list(L):-
  n_words(N),
  numlist(1,N,L).

n_topics(2).

n_words(10).
```

We can, for example, use the model generatively and sample values for word in position 1 of document 1. The histogram of the frequency of word values when taking 100 samples is shown in Figure 8.6.

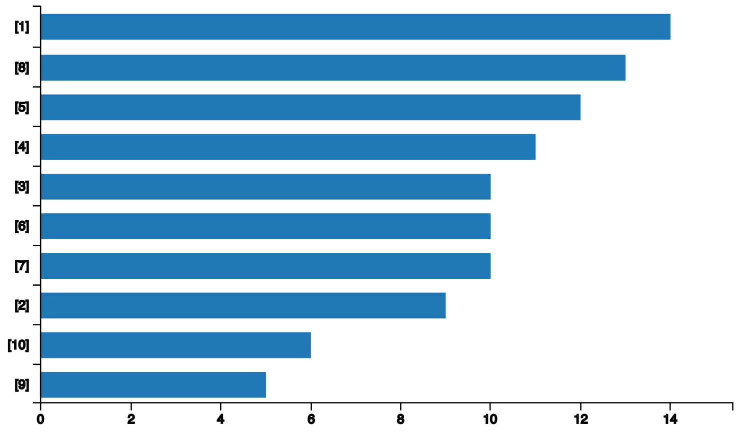


Figure 8.6: Values for word in position 1 of document 1.

We can also sample values for couples (word, topic) in position 1 of document 1. The histogram of the frequency of the couples when taking 100 samples is shown in Figure 8.7.

We can use the model to classify the words into topics. Here we use conditional inference with Metropolis-Hastings that is implemented in MCINTYRE. A priori both topics are about equally probable for word 1 of document , so if we take 100 samples of `topic(1,1,T)` we get the histogram in Figure 8.8. If we ob-

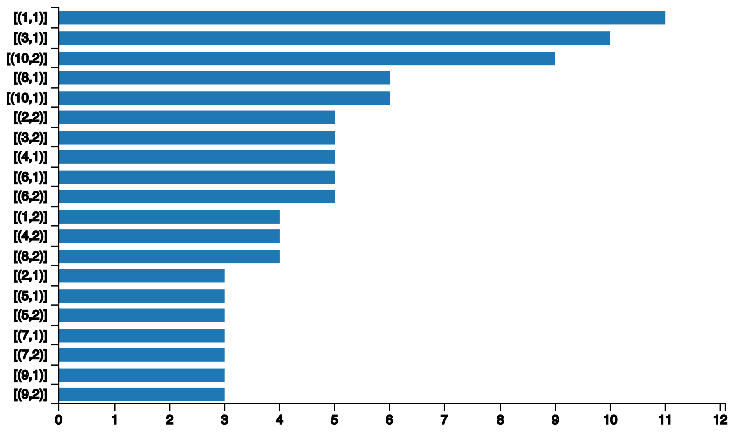


Figure 8.7: Values for couples (word,topic) in position 1 of document 1.

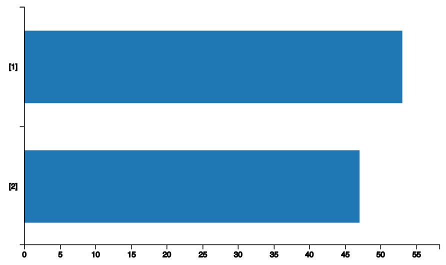


Figure 8.8: Prior distribution of topics for word in position 1 of document 1.

serve that words 1 and 2 of document 1 are equal (`word(1,1,1),word(1,2,1)` as evidence) and take again 100 samples, one of the topics gets more probable, as the histogram of Figure 8.9 shows. You can also see this if you look at the density of the probability of topic 1 before and after observing that words 1 and 2 of document 1 are equal: the observation makes the distribution less uniform, see Figure 8.10

8.7 The Indian GPA Problem

In the Indian GPA problem proposed by Stuart Russel [94, 91] the question is: if you observe that a student GPA is exactly 4.0, what is the probability that the student is from India, given that the American GPA score is from 0.0 to 4.0 and the Indian GPA score is from 0.0 to 10.0? Stuart Russel observed than most probabilistic programming system are not able to deal with this query because it requires combining continuous and discrete distributions. This

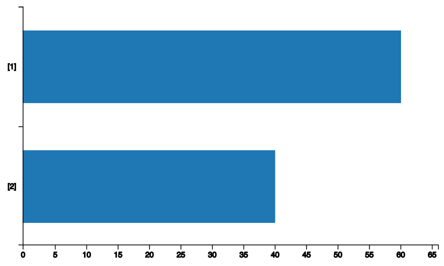


Figure 8.9: Posterior distribution of topics for word in position 1 of document 1.

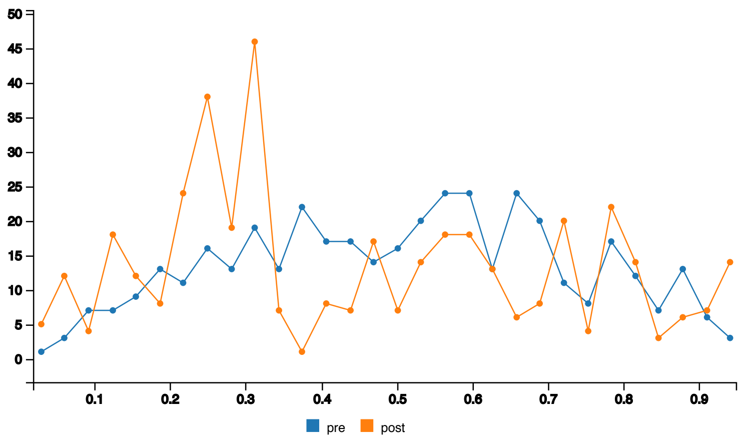


Figure 8.10: Density of the probability of topic 1 before and after observing that words 1 and 2 of document 1 are equal.

problem can be modeled by building a mixture of a continuous and a discrete distribution for each nation to account for grade inflation (extreme values have a non-zero probability). Then the probability of the student's GPA is a mixture of the nation mixtures. From statistics, given this model and the fact that the student's GPA is exactly 4.0, the probability that the student is American must be 1.0.

This problem can be modeled with Anglican, DC and MCINTYRE. In MCINTYRE we can model it with the program below[8]. The probability distribution of GPA scores for American students is continuous with probability 0.95 and discrete with probability 0.05:

```
is_density_A:0.95;is_discrete_A:0.05.
```

The GPA of an American student follows a beta distribution if the distribution is continuous:

[8]http://cplint.lamping.unife.it/example/inference/indian_gpa.pl

```
agpa(A): beta(A,8,2) :- is_density_A.
```

The GPA of an American student is 4.0 with probability 0.85 and 0.0 with probability 0.15 if the distribution is discrete:

```
american_gpa(G) : finite(G,[4.0:0.85,0.0:0.15]) :- is_discrete_A.
```

or is obtained by rescaling the value of returned by `agpa/1` to the (0.0,4.0) interval:

```
american_gpa(A):- agpa(A0), A is A0*4.0.
```

The probability distribution of GPA scores for Indian students is continuous with probability 0.99 and discrete with probability 0.01.

```
is_density_I : 0.99; is_discrete_I:0.01.
```

The GPA of an Indian student follows a beta distribution if the distribution is continuous:

```
igpa(I): beta(I,5,5) :- is_density_I.
```

The GPA of an Indian student is 10.0 with probability 0.9 and 0.0 with probability 0.1 if the distribution is discrete:

```
indian_gpa(I): finite(I,[0.0:0.1,10.0:0.9]):-  is_discrete_I.
```

or is obtained by rescaling the value returned by `igpa/1` to the (0.0,10.0) interval:

```
indian_gpa(I) :- igpa(I0), I is I0*10.0.
```

The nation is America with probability 0.25 and India with probability 0.75.

```
nation(N) : finite(N,[a:0.25,i:0.75]).
```

The GPA of the student is computed depending on the nation:

```
student_gpa(G) :- nation(a),american_gpa(G).
student_gpa(G) :- nation(i),indian_gpa(G).
```

If we query the probability that the nation is America given that the student got 4.0 in his GPA we obtain 1.0, while the prior probability that the nation is America is 0.25.

Part IV

Probabilistic Inductive Logic Programming

Chapter 9

Inductive Logic Programming

Logic is a powerful tool for representing and modeling real world domains especially those in which entities are connected by a network of relationships. With respect to other approaches of machine learning, logic has some advantages: it provides a uniform and expressive method for representing examples, theories and background knowledge. Logic in general and First Order Logic in particular, is a very well developed mathematical field. In this chapter we describe how to learn general logic programs from data. We present different approaches for representing examples and theories and for inducing theories from examples. The chapter is organized as follows: after a formal definition of Inductive Logic Programming in Section 9.1, Sections from 9.2 to 9.4 present different ILP settings which are respectively learning from entailment, from interpretations and from proofs. Section 9.5 finally describes how to structure and perform the search in the space of theories.

9.1 Definition

Inductive Logic Programming (ILP) [81, 3] is a field of Machine Learning that uses logic programming as a language for representing examples and models. It provides learning algorithms for inducing a general theory from examples. The ILP problem can be formally defined as follows:
Given

- a space of possible theories $\mathbb{T}$

- a set E^+ of positive examples
- a set E^- of negative examples
- a **background knowledge** B such that
  $B \not\models e^+$ for at least one $e^+ \in E^+$

Find a theory $T \in \mathbb{T}$ such that

1. T is **complete** with respect to E^+
   all the positive examples are covered by T
2. T is **consistent** with respect to E^-
   no negative example is covered by T

The background knowledge is knowledge a priori in the domain of interest and consists of a set clauses whose truth is known. $E = E^+ \cup E^-$ is the **training set** and $\mathbb{T}$, the **hypothesis (theory) space**, defines the search space. The description of this hypothesis space is called **language bias**.

From the general definition of ILP described previously, different learning settings can be considered based on the methods to represent the *input* (i.e the examples and the background knowledge), the *output* (the induced theory) and to define the covering relation. Three main learning settings in the literature are presented in the following sections: learning from *entailment*, from *interpretations* and from *proofs* respectively.

9.2 Learning from entailment

Learning from entailment is one of the most popular learning settings in ILP. It has been implemented in many ILP systems such as FOIL [104], Progol [79] and Aleph [129].

Definition 6. ***Learning from entailment***
In the learning from entailment, examples are definite clauses (generally ground facts), theories are normal logic programs and the coverage relation is entailment. A theory T covers an example $e \in E$ iff

$$B, T \models e$$

Example 15. *Learning the predicate father(X,Y)*
Given:

- *a space of logic programs including clauses of the form*

  $$father(X,Y) :- \alpha$$

  where α is a conjunction of literals from the set

  $$\{parent(X,Y), parent(Y,X), male(X),$$
  $$male(Y), female(X), female(Y)\}$$

- *the background knowledge*

  $$\begin{aligned}B =\{&parent(john, mary), male(john),\\ &parent(david, steve), male(david),\\ &parent(kathy, ellen), female(kathy)\}\end{aligned}$$

- *and the set of positive and negative examples*

  $$\begin{aligned}E^+ &= \{father(john, mary), father(david, steve)\}\\ E^- &= \{father(kathy, ellen), father(john, steve)\}\end{aligned}$$

Find a theory which describes the predicate father consistent with respect to E^+ and E^-. A possible solution could be

$$father(X,Y) :- parent(X,Y), male(X).$$

9.3 Learning from interpretations

Definition 7. ***Learning from interpretations***
When learning from interpretations [27], theories are set of disjunctive clauses, examples are Herbrand interpretations and the coverage relation is truth in an

interpretation. An interpretation I is covered by a theory T iff

$$I, B \models T$$

In the same way a clause C covers an interpretation if $I, B \models C$ i.e for all grounding substitutions θ of C, $I \models \mathit{body}(C)\theta \Rightarrow \mathit{head}(C) \cap I \neq \emptyset$. In order to have the truth of a clause C in an interpretation I, one can run the query $?-\mathit{body}(C), \mathit{not}(\mathit{head}(C))$ against a logic program containing I: if the query succeeds, C is false in I. Otherwise C is true in I . Note that in this setting an example is an interpretation (set of facts) formed by computing the model of the background and I.

Example 16. *Consider the background*

$$\begin{aligned} B = \{&father(henry, bill), father(alan, betsy), father(alan, benny),\\ &father(brian, bonnie), father(bill, carl), father(benny, cecily),\\ &father(carl, dennis), mother(ann, bill), mother(ann, betsy),\\ &mother(ann, bonnie), mother(alice, benny), mother(betsy, carl),\\ &mother(bonnie, cecily), mother(cecily, dennis), founder(henry).\\ &founder(alan).founder(an).founder(brian).founder(alice).\}\end{aligned}$$

and the interpretation $I = \{carrier(alan), carrier(ann), carrier(betsy)\}$
This interpretation is covered by the clause
$carrier(X) : -mother(M, X), carrier(M), father(F, X), carrier(F).$

There is a main difference between learning from entailment and from interpretation: while examples in learning from interpretations contain much information (interpretations are *large set of facts*), examples in the learning from entailment typically consist of a *single fact.*

9.4 Learning from proofs

With respect to the amount of information each example carries, learning from entailment represents an example as a simple fact and learning from interpretations represents an example as a set of severals ground facts. Since these approaches occupy extreme positions w.r.t. the amount of information, it is important to explore another (possibly) middle position. Learning from proofs [54] investigates this intermediate position.

Definition 8. ***Learning from proofs***
In this learning setting, examples are ground proof-trees, theories are set of definite clauses and the background knowledge B is a set of ground facts. An example e is covered by a theory T iff e is a proof tree for $T \cup B$.

Note that since example are expressed as ground proof-trees, various possible forms of proof-trees can be used. In [54], trees are represented as and-tree in which nodes contain ground atoms. It is worth nothing that proof-trees (as interpretations) contain a lot of information. They typically contain instances of clauses used in the proofs. It is hard to use learning from proofs because it is not always easy to provide examples of this form.

Among these learning settings, learning from entailment and from interpretations are the most popular and represent the approaches adopted in the following part for developing ILP algorithms.

9.5 Search Space

We have seen in the previous sections various approaches for inducing a theory from a set of examples and a background knowledge. In this section we are going to describe the general ILP learning procedure and how to structure the search.

The main goal of ILP is to search a theory in the space of theories that satisfies desirable properties and which is consistent w.r.t the examples and the background knowledge. Learning means finding a correct theory. Two main procedures are often performed during theory searching: *generalization* and *specialization*. During search, if the current theory together with the background knowledge cannot entail all positive examples, we have to find a more general theory such that as many as possible (or all) positive examples are entailed. Conversely, if the current theory together with the background covers negative examples, we have to find a more specific theory which is consistent w.r.t the negative examples. Therefore learning means repeating these actions until the final theory entails all positive examples and does not entail the negative ones. This leads to the following definitions.

Definition 9. ***Generalization***
Let $T_1, T_2 \in \mathbb{T}$ be two theories: T_1 is a generalization of T_2, denoted $T_1 \preceq T_2$, iff all examples cover by T_2 are covered by T_1. We note $cov(T_2) \subseteq cov(T_1)$ where $cov(T)$ denotes the set of examples covered by T.

If T_1 is a generalization of T_2 then T_2 is its specialization. These definitions hold when theories are composed of single or sets of clauses. If a theory (clause) covers an example, all of its generalizations will cover as well (covers is *anti-monotonic* with respect to specialization). If a theory (clause) does not cover an example, none of its specializations will.

Now let us describe how to organize the search in learning from entailment setting.

Definition 10. *When learning from entailment, a theory $T \in \mathbb{T}$ is more general than $H \in \mathbb{T}$, written $T \geq H$, iff T logically entails H, i.e $T \models H$.*

Instead of using the entailment relation, which is often computationally expensive, θ-subsumption is generally used.

Definition 11. ***θ-subsumption***
A clause C θ-subsumes D, written $C \geq D$ if there exists a substitution θ such that $C\theta \subseteq D$, see [96].

From the definition, if $C \geq D$ then $C \models D$ and so C is more general than D. Note that if $C \models D$ it does not follow that C θ-subsumes D. Let us see some examples.

Example 17. *Consider the following clauses*

$$\begin{aligned}
C_1 &= father(X,Y) :- parent(X,Y).\\
&= \{father(X,Y), \neg parent(X,Y)\}\\
C_2 &= father(X,Y) :- parent(X,Y), male(X).\\
&= \{father(X,Y), \neg parent(X,Y), \neg male(X)\}\\
C_3 &= father(john, steve) :- parent(john, steve), male(john).\\
&= \{father(john, steve), \neg parent(john, steve), \neg male(john)\}
\end{aligned}$$

We have that:

- C_1 *θ-subsumes* C_2 *with* $\theta = \emptyset$
- C_1 *and* C_2 *θ-subsumes* C_3 *with* $\theta = \{X/john, Y/steve\}$

Example 18. *Given the following clauses*

$$C_1 = even(X) : -even(half(X)).$$
$$C_2 = even(X) : -even(half(half(X))).$$

We can obtain C_2 *by resolving* C_1 *with itself so* $C_1 \models C_2$*. However, there is no substitution* θ *such that* $C_1\theta \subseteq C_2$ *so* $C_1 \not\succeq C_2$

9.5.1 Refinements Operator

We stated previously that learning in ILP is finding a theory that covers all positive examples and contradicts no negative ones. In order to obtain such theory, one could start from the most general (specific) theory and then explore its specializations (generalizations). This is obtained using a refinement operator.

Definition 12. ***Refinements Operator***
The refinement operator consists of generating a set generalizations or specializations of a theory during the search.

In learning from entailment and from the definition of θ-subsumption, a clause can be generalized by applying the following refinement operations:

1. removing a literal from the body.
2. turning constants into variables (either in the head or in the body).
3. adding a new atom to the head.

Example 19. *In Example 17,* C_1 *is a generalization of* C_2 *and* C_3*. In fact* C_1 *is obtained from* C_2 *by removing the literal* $male(X)$ *and from* C_3 *by removing the literal* $male(johh)$ *and turning the constants* $john$ *and* $steve$ *into variables* X *and* Y *respectively.*

In the same way, a clause can be specialized by applying the following operations:

1. replacing variables with constants.
2. removing atom from the head.
3. adding literal to the body.

Example 20. *C_2 in Example 17 is a specialization of C_1 and C_3 a specialization of C_2. In fact C_2 is obtained from C_1 by adding the literal $male(X)$ to its body and C_3 from C_2 by replacing the variables X and Y by constants $john$ and $steve$.*

In ILP, the learning algorithm consists of successively adding clauses to an initial theory (often empty). These clauses can be found by performing a search either *top-down* or *bottom-up*. Top-down search starts with a most general clause and successively applies a set of specializations while bottom-up approach starts with a most specific clause and proceeds by generalization. For a detailed description of top-down search , see Section 9.5.3.

Since the search space is in general large, specific forms of clauses are considered during the learning. The user has to define the form of the allowed theories by means of a language called **language bias**.

9.5.2 Language bias

The language bias defines the clauses contained in the space of clauses during the learning procedure. It imposes restrictions on the form of clauses to be induced. In the literature, many language bias have been defined but all share the same basic concept: a) the use of *predicate, type* and *mode declarations* which respectively define the predicates to be used, the type of their arguments and the input/output behavior of their arguments b) the notion of *determination* which provides the form of the allowed clauses. Let us present the form of these declarations used in the Aleph [129] and SLIPCOVER [11] systems.

The **mode declarations** define the mode of call for predicates that can appear in any clause. Following Muggleton [79], a mode declaration is either a head declaration

$$modeh(RecallNumber, PredicateMode).$$

or a body declaration

$$modeb(RecallNumber, PredicateMode).$$

The $RecallNumber$ can be either a number which specifies the number of answers to consider the predicate or $*$ which specifies that all answers are considered. $PredicateMode$ is a schema representing a template for literals in the head or in the body of clauses in the theories and are of the form $p(ModeType, ModeType, ...)$ where each $ModeType$ is either a simple or structured place-maker terms. Simple place-maker terms are of the form $\#Type$, $+Type$, $-Type$ which respectively stand for ground terms, input variables and output variables of type $Type$. Structured place-maker terms are of the form $f(\ldots)$ where f is a function symbol whose arguments are either a simple or structured place-maker.

Type specifications associate a $Type$ to every argument of all predicates. This is done by means of place-marker described in the previous paragraph. Note that types are just names and no control is performed to verify if a constant (ground terms) corresponds to a particular $Type$.

Determinations define predicates for constructing the clauses in the theory. It is of the form

$$determination(HeadName/Arity, BodyName/Arity).$$

where $HeadName/Arity$ is the name (together with the arity) of a predicate which can appear in the head of a clause and $BodyName/Arity$ represents the predicate which can appear in the body of such clause. Since a definite clauses has one predicate in the head and many literals in the body, many determinations with the same $HeadName$ and different $BodyName$ can occur.

With this language bias, we ensure the following properties for an hypothesized clause of the form $h : -b_1, b_2, \ldots, b_m$:

- any input variable in a literal b_i appears as an output variable in a literal

b_j (with $j < i$)

- any output variable in h appears as an output variable in some b_i,
- any argument of predicates required to be ground are ground.

Now let us present the general learning algorithms used in ILP together with the top-down algorithm for searching the clauses.

9.5.3 ILP algorithm

The general top-down ILP algorithm is shown in Algorithm 1. The algorithm

Algorithm 1 ILP algorithm.

```
1: function LearnTheory(E,B)
2:     $P := \emptyset$
3:     repeat/* covering loop */
4:         $C$ :=FindClauseTopDown(E,B)
5:         $P := P \cup C$
6:         Remove from $E$ the positive examples covered by $P$
7:     until Stop condition
8:     return $P$
9: end function
```

takes as input a set of positive and negative examples $E = E^+ \cup E^-$, a background knowledge B and returns a program (set of normal clauses) that entails as many positive examples as possible and as few negative ones as possible. The function starts with an empty program, line 2, and finds a clause C that covers some positive examples in E and covers few negative ones, line 4. C is found by performing a top-down search, see Algorithm 2. Then C is added to the current program and all the positive examples covered by the current program are removed in E, lines 5-6. This procedure is repeated until a certain condition ($E^+ = \emptyset$) is satisfied, line 7. Finally, the learned program is returned, line 8.

To find the clause C, Algorithm 2 takes as input the current set of examples E and the background knowledge B and a beam which initially contains the most general clause, line 2. Then it removes the first clause in the beam and computes its refinements, lines 5-6, as described in Section 9.5.1. The refined clauses are scored using a heuristic function and the clause with the best score is selected as the currently best clause, lines 7-8. The refined clauses are added back to the beam and the beam is ordered according to the score.

The last clauses that exceed the size d of the beam are removed, lines 9-11. The algorithm repeats until a stop condition is satisfied (for example $E^+ = \emptyset$), line 12. Finally, the best clause in the beam is returned, line 13.

Examples of ILP systems based on this algorithm are FOIL [104] and Alpeh [129] Progol [79].

Algorithm 2 Find clause

```
 1: function FindClauseTopDown(E,B)
 2:     $Beam := \{p(X) \leftarrow true\}$
 3:     $BestClause := null$
 4:     repeat/* specialization loop */
 5:         Remove the first clause C of Beam
 6:         compute $\rho(C)$
 7:         score all the refinements
 8:         update $BestClause$
 9:         add all the refinements to the beam
10:         order the beam according to the score
11:         remove the last clauses that exceed the dimension d
12:     until Stop condition
13:     return $BestClause$
14: end function
```

Chapter 10

Learning Probabilistic Logic Programming

In Chapter 7, we saw that PLPs are useful for representing various domains especially those characterized by uncertainty. We have presented various languages that integrate logic with probability and described how to perform inference in such languages. In the previous chapter, we presented Inductive Logic Programs and described many setting for inducing logic programs from data. Since PLP is an interesting representation formalism in machine learning and given the different learning settings for ILP, an interesting problem to investigate is how to induce a probabilistic program from data. This learning task is often called *Probabilistic Inductive logic Program* (PILP). Two main learning problems have been investigated in PILP: given a PLP with unknown parameters, learning the parameters from data. *Structure learning* algorithms instead aim at inducing a PLP (and their parameters) from data. This chapter presents PILP and describes different state-of-art parameter and structure learning algorithms. After a brief description of PILP in Section 10.1, Section 10.2 presents two parameter learning algorithms called EMBLEM [10] and LFI-ProbLog [36]. While EMBLEM learns the parameters of LPADs (described in Section 7.2) from data, LFI-ProbLog estimates those of Problog program (see Section 7.3) from data. Section 10.3 presents a structure learning algorithm called SLIPCOVER [11] which learns both the structure and the parameters of general LPADs from data.

10.1 PILP Settings

In order to deal with PILP we have to:

1. Define a method for integrating probability and clauses. This has been presented in Part III. Many languages allowing such integration have also been presented.

2. Define a new covers relation in ILP called probabilistic covers relation which can be defined as follows:

Definition 13. *Given an example e, a theory T and a background knowledge B, a probabilistic covers relation, written $cov(e, T \cup B) = P(e|T, B)$, returns a value between 0 and 1 which represents the probability that the example is covered (the likelihood of the example).*

Given a set of positive and negative examples $E = E^+ \cup E^-$, the purpose of PILP is to find a PLP T^* that maximizes the likelihood, $P(E|T^*, B)$, of the data. Assuming that all examples are independently and identically distributed (i.i.d), the likelihood of the data becomes

$$P(E|T^*, B) = \prod_{e \in E^+} P(e|T^*, B) \cdot \prod_{\neg e \in E^-} P(\neg e|T^*, B) \tag{10.1}$$

Note that a PLP is composed of two components: the set of clauses which denotes the *structure* of the program and the probabilities associated with each clause which represent its *parameters*. Therefore, two common algorithms have been investigating in PILP. Suppose the structure of a PLP is known, maybe given by an expert in the domain of interest, *parameter learning* algorithms estimate the optimal values of the probabilities that best describe data. If both the parameters and structure are unknown, *structure learning* algorithms induce the structure and the values of the probabilities that best describe the data. In the following sections, we present a brief description of each algorithm and discuss state-of-art algorithms.

10.2 Parameter learning

The parameter problem can be defined as follows:
Given

1. a set of examples E,
2. a background knowledge B,
3. a Probabilistic model $M = (S, \Lambda)$ where S is the structure and Λ the parameters,
4. a probabilistic coverage $P(e|M, B)$ which computes the probability of an observing example $e \in E$ given the model and the background,
5. a scoring function, $score(E, M)$, that uses the probabilistic coverage $P(e|M, B)$.

Find the optimal parameters Λ^* that maximize the scoring function, i.e

$$\Lambda^* = argmax_{\Lambda} score(E, (S, \Lambda)) \tag{10.2}$$

Generally, the (conditional) likelihood of the data given the model is considered as scoring function, see Equation 10.1. In traditional PILP, instead of maximizing the conditional likelihood, its logarithm is maximized. This is done because products of many numbers all smaller than one become very small quickly and thus pose numerical issues. Therefore, the *conditional log likelihood (CLL)* function is used as the scoring function. In order to find the values of the parameters that maximized the CLL, two main algorithms are often performed: the *Expectation Maximization* and *Gradient Descent* algorithms.

10.2.1 Expectation Maximization

Expectation-Maximization (EM), see [32], is an iterative approach for computing an estimation of parameter values that maximize the (conditional) log likelihood. The EM approach is useful in domains in which data are not completely observed, also known as incomplete data problems. If the data are fully observed, optimizing the CLL reduces to frequency counting. Otherwise, a numerical optimization algorithm has to be applied. The EM algorithm starts by randomly initializing the parameters and then iteratively performs two steps:

- **Expectation Step**
  Given the current parameters of the program and the partially observed data, the E-step computes an estimation of the conditional distribution of the unobserved data. These unobserved data are also called hidden (or latent) variables.

- **Maximization step**
  The M-step then computes the parameters of the program that maximize the CLL function under the assumption that the missing data are known (computed in the E-step).

These steps are repeated until convergence or until a stopping condition is satisfied. Note that each EM iteration increases the CLL function. Moreover, if there are multiple maxima, EM does not guarantee convergence to the global one.

EM is applied in many parameter learning systems such as EMBLEM and LFI-ProbLog described in sections 10.2.4 and 10.2.5 respectively. Other systems such as PRISM [124] and RIB [114] also implement an EM approach.

10.2.2 Gradient Descent

Gradient-based methods are iterative approaches for searching the parameters of PLPs for which the CLL is optimal. While *gradient descent* aims at finding a (local) minimum of the function, *gradient ascent* finds the (local) maximum. These algorithms are based on iterative optimization. They compute the gradient of the CLL function w.r.t to the current parameters and iteratively modify the parameters to follow the direction of the gradient, gradient ascent, or to follow the opposite direction of the gradient, gradient descent. Gradient descent has been implemented in systems such as LeProbLog [41] that uses a dynamic programming algorithm for computing the gradient exploiting Binary decisions diagrams.

Gradient descent is widely used in another field of artificial intelligence called *deep learning* [70] and *neural networks* [44]. This approach of AI, which is mostly applied in domains such as computer vision and natural language processing, uses a technique called *back-propagation* [118] for computing the gradient of a function in deep models organized into layers. Given an objective function f (also called loss function) and the parameters Π, the optimization algorithms compute the derivative of the objective function with respect to each parameter π, $d(\pi) = \frac{df}{d\pi}$. According to a (minimization) maximization problem, the parameters are updated in the (opposite) same direction of the gradient. Standard gradient descent updates the parameters in the following manner:

$$\pi = \pi - \alpha \cdot d(\pi) \tag{10.3}$$

where α, the *learning rate*, defines the speed of descent.

Different extensions of the standard gradient descent have been implemented. Some of them are described below:

- **Momentum method.** In the Momentum method [102], gradient descent is accelerated by taking into consideration the exponentially weighted average of the gradients. Since the gradients towards uncommon directions are eliminated, the algorithm converges in a faster way towards the minimum. The parameters are updated as follows:

$$\begin{aligned} A &= \beta \cdot A + (1-\beta) \cdot d(\pi) \\ \pi &= \pi - \alpha \cdot A \end{aligned} \tag{10.4}$$

  where A is the acceleration (initially 0) and $d(\pi)$ the velocity of descent. β (generally ≈ 0.9) is an hyper-parameter called *momentum*.

- **RMSprop.** Proposed by Geoffrey Hinton, RMSprop [136] applies an exponentially weighted average method to the second moment of the gradients ($d(\pi)^2$). The parameters are updated as follows:

$$\begin{aligned} S &= \beta \cdot S + (1-\beta) \cdot d(\pi)^2 \\ \pi &= \pi - \alpha \cdot \frac{d(\pi)}{\sqrt{S} + \epsilon} \end{aligned} \tag{10.5}$$

  where $\epsilon \approx 10^{-8}$ avoids dividing by zero.

- **Adam Optimization.** Adam Optimization [62] is a combination of the previous methods along with a bias correction. If t is the current iteration of the algorithm, the parameters are updated as follows:

$$\begin{aligned}
&A = \beta_1 \cdot A + (1-\beta_1) \cdot d(\pi) \qquad \hat{A} = \frac{A}{1-\beta_1^t} \\
&S = \beta_2 \cdot S + (1-\beta_2) \cdot d(\pi)^2 \qquad \hat{S} = \frac{S}{1-\beta_2^t} \\
&\pi = \pi - \alpha \cdot \frac{\hat{A}}{\sqrt{\hat{S}}+\epsilon}
\end{aligned} \tag{10.6}$$

  where A and S are respectively the estimations of the first and the second moments of the parameter and $\hat{A}$ and $\hat{S}$ their respective bias corrections.

Adam optimization will be used in this work in Section 13.2 for learning the parameters of a restriction of general LPADs called *Hierarchical PLP*, see Chapter 12.

10.2.3 Limited Memory BFGS: LBFGS

Another optimization method, called Limited-memory BFGS (LBFGS) [92], is based on the **Broyden–Fletcher–Goldfarb–Shanno (BFGS)** method that is often used for parameter learning. The BFGS algorithm tries to approximate quasi-Newton optimization by avoiding the computational effort necessary for computing the inverse of the Hessian matrix. Let $f(\Pi)$ be the objective function to minimize and $J(\Pi)$ its second-order approximation near Π_0 using the Taylor series. The Newton parameter update rule is given by

$$\Pi^* = \Pi_0 - H^{-1} \nabla_\Pi J(\Pi_0) \tag{10.7}$$

where H is the Hessian of J w.r.t Π_0 and $\nabla_\Pi J(\Pi_0)$ the gradient at Π_0. If $J(\Pi)$ is not quadratic and the Hessian matrix is positive definite, the update rule has to be iterated in order to gradually move towards the global minimum. However, this method suffers from the computational burden necessary for computing the inverse of the Hessian matrix at each iteration. BFGS proposes a method for approximating this computation. The approach adopted by BFGS (and by almost all quasi-Newton methods) is to approximate the inverse at each iteration t with a matrix M_t that is iteratively refined by low rank updates to become a better approximation of H^{-1}. Once the approximation is done, the direction of the descent $\rho_t = M_t \cdot g_t$ is computed and the parameters are updated as follows

$$\Pi_{t+1} = \Pi_t + \varepsilon^* \cdot \rho_t$$

where g_t is the gradient of the parameters at iteration t and ε^* the length of the step. Note that, to perform well, BFGS algorithm has to store the approximation of the Hessian matrix at each iteration. This requires $O(n^2)$ memory space where n is the number of parameters. For modern models with many parameters implementing BFGS is often impractical.

Limited Memory BFGS, written LBFGS, tries to avoid storing M_t at each iteration. The approximation of M_t is computed by using the same approach as BFGS but LBFGS assumes that at each iteration t $M^{(t-1)}$ is the identity matrix. For a more details on the LBFGS algorithm, see [92].

LBFGS optimization will be applied in Section 11.4.2 for learning the parameters of *Liftable PLP*, see Section 11.2.

10.2.4 EMBLEM

EMBLEM (for Expectation Maximization over Binary Decision Diagrams for Probabilistic Logic Programs) [10], is a parameter learning algorithm that learns the parameters of general LPADs from data.

In EMBLEM, examples E are ground facts called target (or output) predicates and the background knowledge B are other facts, called *input predicates* . The model is an LPAD program T, which is a set of annotated disjunctive clauses C_i with unknown probabilities $\Pi =< \pi_{i_1} \dots \pi_{in_i} >$ in the heads. The learning consists of finding the parameters Π^* that maximize the conditional probability of the examples given the program and the input predicates.

$$\Pi^* = argmax_{\Pi} P(E|T, B) = \prod_{e \in E} P(e|T, B)$$

EMBLEM uses knowledge compilation as described in Section 7.4. The explanations of each example is compiled into a Binary Decision Diagram (BDD). Parameter learning uses the EM algorithm described in Section 10.2.1.

The expectations are computed by performing two passes over the BDDs.

Algorithm 3 EMBLEM algorithm.

```
1: function EMBLEM($LPAD, MaxIter, \epsilon, \delta$)
2:     Build the BDDs for the examples (facts for target predicates)
3:     LL:=-inf
4:     N=0
5:     repeat
6:         $LL_0 = LL$
7:         $LL$ =Expectation(BDDs)
8:         Maximization()
9:         $N = N + 1$
10:    until $LL - LL_0 < \epsilon || LL - LL_0 < -LL \cdot \delta || N > MaxIter$
11:    Update parameters of $LPAD$
12:    return $LL, LPAD$
13: end function
```

EMBLEM, Algorithm 3, starts by building a BDD for each example, line 2. Then, an EM cycle is repeatedly performed until the difference between the current and the previous CLL drops below a threshold ϵ or the difference is below a fraction δ or a maximum number of iteration $MaxIter$ is reached, line 10. In the E-step, the expectations of the hidden variables are computed directly over the BDDs. The E-step also computes the likelihood used in the stopping condition. Then the expectations computed in the E-step are used in the M-step to update the current parameters by relative frequency. How to compute the expectations is described in the following.

Let us define c_{ikx} the number of times the boolean variable X_{ijk} takes value $x \in \{0,1\}$. We want to compute $E[c_{ikx}]$. For each example e, the conditional expectations $E[X_{ijk} = x|e]$ for clause C_is, $k = 1, \ldots, n_i - 1$, $j \in g(i) = \{j|\theta_j$ is a substitution grounding $C_i\}$ and $x \in \{0,1\}$ are computed. As defined in Equation 6.5, $E[X_{ijk} = x|e]$ is computed as follows:

$$\begin{aligned} E[X_{ijk} = x|e] &= P(X_{ijk} = x|e) \cdot 1 + P(X_{ijk} = 1 - x|e) \cdot 0 \\ &= P(X_{ijk} = x|e) \end{aligned}$$

The expectation $E[c_{ikx} = x|e]$ for all $j \in g(i)$ is defined as follows:

$$\begin{aligned} E[c_{ikx} = x|e] &= \sum_{j \in g(i)} E[X_{ikx} = x|e] \\ &= \sum_{j \in g(i)} P(X_{ijk} = x|e) \qquad (10.8) \end{aligned}$$

Considering all the examples, we have

$$E[c_{ikx}] = \sum_{e \in E} E[c_{ikx} = x|e] \tag{10.9}$$

Since $P(X_{ijk} = x|e) = \frac{P(X_{ijk}=x,e)}{P(e)}$, $P(X_{ijk} = x|e)$ can be computed by computing $P(X_{ijk} = x, e)$ and $p(e)$ which are done by performing two passes over the BDD associated with the example e.

Once the expectations $E[c_{ik0}]$ and $E[c_{ik1}]$ are computed in the E-step, the M-step updates each parameter π_i by applying the following formula

$$\pi_i = \frac{E[c_{ik1}]}{E[c_{ik0}] + E[c_{ik1}]}$$

EMBLEM is strongly related to EMPHIL, see Section13.3, which is an EM algorithm for learning the parameters of Hierarchical PLPs described in Chapter 12.

10.2.5 Learn From Interpretation ProbLog: LFI-ProbLog

LFI-ProbLog [36] is a parameter learning system that learns the parameters of ProbLog2 programs, an extension of the ProbLog language described in Section 7.3, from interpretations. In addition to probabilistic facts allowed in ProbLog, ProbLog2 introduces the following probabilistic clauses:

- **Intensional probabilistic facts.** The following intensional probabilistic fact

  $$\pi :: f(X_1, \ldots, X_n) : -Body$$

  represents a set of probabilistic facts with a single statement. *Body* is a conjunction of calls to non-probabilistic facts. When performing inference and learning, each intentional probabilistic fact is replaced by the corresponding set of ground probabilistic facts.

- **Annotated disjunction clauses.** ProbLog2 allows annotated disjunction clauses like LPAD clauses. Each annotated disjunction is written in the form

  $$\pi_{i1} :: h_{i1}; \ldots \pi_{in_i} :: h_{in_i} : -b_{i1}, \ldots b_{im_i}$$

which corresponds to the following LPAD clause

$$h_{i1} : \pi_{i1}; \ldots h_{in_i} : \pi_{in_i} : -b_{i1}, \ldots b_{im_i}$$

Note that during inference and learning, each annotated disjuction clause is converted to probabilistic facts as described in Section 7.3.

The parameter learning is formally defined as follows:
Given

- a set of examples $E = \{I_1, \ldots, I_T\}$ which are partial interpretations,
- a ProbLog2 program P with unknown parameters Π,

Find the values of the parameters, Π^*, that maximize the likelihood of the examples, .i.e

$$\Pi^* = argmax_{\Pi} P(E) = argmax_{\Pi} \prod_{t=1}^{T} P(I)$$

Each partial interpretation $I = I_T \cup I_F$ specifies the truth value of ground atoms in the interpretation. Atoms $\in I_T$ are true and those in $\in I_F$ are false. If each interpretation contains all the atoms, parameter learning can be done by relative frequency. Otherwise algorithms based on EM or on gradient descent can be applied.

LFI-ProbLog learns the parameters of ProbLog2 programs applying EM. The algorithm initially generates the ground probabilistic facts corresponding to intensional probabilistic facts and converts annotated disjunction clauses to probabilistic facts. Then an EM cycle is repeated until convergence.

Let X_{ij} be a Boolean random variable associated with each ground probabilistic facts $f_i\theta_j$, where $j \in g(i)$ and $g(i)$ is the set of grounding substitutions of f_i. We define, as in EMBLEM, the latent random variable c_{ix} denoting the number of times the boolean variable X_{ij} takes value $x \in \{0, 1\}$ for all interpretations. The E-step computes the expected values of c_{ix} as follows:

$$E[c_{ix}] = \sum_{t}^{T} E[c_{ix}|I_t]$$

where the expectation $E[c_{ix}|I_t]$ is computed by taking into account all the substitutions

$$E[c_{ix}|I_t] = \sum_{j \in g(i)} P(X_{ij} = x|I_t)$$

To compute $P(X_{ij} = x|I_t)$, LFI-ProbLog builds a d-DNNF (Deterministic Decomposable Negation Normal Form) circuit associated with I_t and visits the circuit twice: once bottom up to compute $P(I_t)$ and once top down to compute $P(X_{ij} = x, I_t)$. Then $P(X_{ij} = x|I_t)$ is computed by formula $\frac{P(X_{ij}=x,I_t)}{P(I_t)}$.

After calculating the expectations, each parameter π_i is updated in the M-step as follows:

$$\pi_i = \frac{E[c_{i1}]}{E[c_{i0}] + E[c_{i1}]}$$

10.3 Structure learning

In the previous section, the structure of PLPs was considered as known and the problem was to learn the parameters from data. However, in many domains of interest, it is often difficult, even for experts in the domains, to provide the structure of the program. In such domains, both the structure and the parameters have to be induced from data. This task in PILP is called *structure learning* and can be formally defined as follows:

Given

- a set of positives and negatives examples $E = E^+ \cup E^-$
- a language L_M of possible models, $M = (S, \Lambda)$, defined by a language bias as described in Section 9.5.2. S is the structure and Λ the parameters,
- a background knowledge B,
- a probabilistic coverage $P(e|M, B)$ which computes the probability of an observing example $e \in E$ given the model and the background,
- a scoring function, $score(E, M)$, that uses the probabilistic coverage $P(e|M, B)$.

Find the best model $M^* = (S^*, \Lambda^*)$ with the optimal parameters Λ^* that maximizes the scoring function, i.e

$$\Lambda^* = argmax_\Lambda score(E, (S, \Lambda))$$

Similarly to learning in ILP, see Section 9.5, learning in PILP is essentially a search in the space of models. Besides searching for the structure, PILP has to search the associated parameters. Given a language bias, see Section 9.5.2, a clause can be searched using a bottom-up or a top-down algorithms. Many state of the art algorithms for learning PLPs such as SLIPCOVER [11], ProbFOIL [31] and ProbFOIL+ [105] have been implemented. In the following section, we first present SLIPCOVER which it is strongly related to the algorithms proposed in this thesis and then ProbFOIL+.

10.3.1 SLIPCOVER

SLIPCOVER [11] (for **S**tructure **L**earn**I**ng of **P**robabilistic logic programs by sear**C**hing **OVER** the clause space) is a PILP system that learns both the structure and the parameters of LPADs from data. In order to learn, SLIPCOVER takes as input

- a set of examples called target (output) predicates,
- a background knowledge B, consists of input predicates organized as a set of interpretations called mega-examples,
- a probabilistic logical model M,
- a language bias to guide the construction of the refinements of the models,

and **Find** an LPAD P^* with parameters Π^* which maximize the conditional probability of the atoms for the output predicates given the atoms of the input predicates.

Language bias

SLIPCOVER extends the language bias defined in Section 9.5.2 by allowing place-maker of the form $\#{-Type}$ which are treated as $\#Type$ when variabilizing a clause and as $-Type$ when saturating the body of a clause. In addition, SLIPCOVER allows head declarations of the form

$$modeh(r, [s_1, \ldots, s_n], [a_1, \ldots, a_n], [P_1/Ar_1, \ldots, P_k/Ar_k]).$$

which are used for creating clauses with more than two head atoms.

Description of the algorithm

In order to learn, SLIPCOVER, see Algorithm 4, performs a beam search in the space of clauses for identifying the promising ones and a greedy search in the space of theories.

For finding the promising clauses, SLIPCOVER initially creates a beam of bottom clauses for each output predicate as described in Progol [79], line 2. Then each beam is considered in turn. For each bottom clause in the current beam, all the refinements are found by using the language bias. Each refined clause is scored with the conditional likelihood (CLL) obtained by applying EMBLEM on the single clause. Clauses whose heads are target predicates are inserted in a set of target clauses (TC). Otherwise there are inserted in a set of background clauses, BC. TC and BC are ordered by CLL. This is repeated until the beam becomes empty. The whole process is repeated at most NI steps, lines 5-27.

After identifying the promising clauses in TC and BC, the algorithm proceeds by iteratively moving in turn clauses from TC to a new theory Th (initially empty) until TC is empty. When a clause is added in Th, EMBLEM is performed on the current Th and its CLL is computed. If the CLL is larger than the previous one, the clause is kept in the theory. Otherwise it is discarded, lines 28-35.

Finally all the clauses in BC are added into Th and EMBLEM is run on the new theory. Clauses never used in any example derivation are removed and the theory is returned, lines 36-38.

10.3.2 ProbFOIL+

ProbFOIL+ [105] learn rules from probabilistic examples. The learning is defined as follows.

Definition 14. *Given*

1. *a set of training examples* $E = \{(e_1, p_1), \ldots, (e_T, p_T)\}$ *where each* e_i *is a ground fact for a target predicate*

2. *a background theory* B *containing information about the examples in the form of a ProbLog program*

Algorithm 4 Function SLIPCOVER

```
function SLIPCOVER($NInt, NS, NA, NI, NV, NB, NTC, NBC, D, NEM, \epsilon, \delta$)
    $IB$ =InitialBeams($NInt, NS, NA$)                                    ▷ Clause search
    $TC \leftarrow []$
    $BC \leftarrow []$
    for all ($PredSpec, Beam$) $\in IB$ do
        $Steps \leftarrow 1$
        $NewBeam \leftarrow []$
        repeat
            while $Beam$ is not empty do
                remove the first triple ($Cl, Literals, LL$) from $Beam$        ▷ Remove the first clause
                $Refs \leftarrow$ClauseRefinements(($Cl, Literals$), $NV$)        ▷ Find all refinements $Refs$ of
($Cl, Literals$) with at most $NV$ variables
                for all ($Cl', Literals'$) $\in Refs$ do
                    ($LL'', \{Cl''\}$) $\leftarrow$EMBLEM($\{Cl'\}, D, NEM, \epsilon, \delta$)
                    $NewBeam \leftarrow$Insert(($Cl'', Literals', LL''$), $NewBeam, NB$)
                    if $Cl''$ is range-restricted then
                        if $Cl''$ has a target predicate in the head then
                            $TC \leftarrow$Insert(($Cl'', LL''$), $TC, NTC$)
                        else
                            $BC \leftarrow$Insert(($Cl'', LL''$), $BC, NBC$)
                        end if
                    end if
                end for
            end while
            $Beam \leftarrow NewBeam$
            $Steps \leftarrow Steps + 1$
        until $Steps > NI$
    end for
    $Th \leftarrow \emptyset$, $ThLL \leftarrow -\infty$                                    ▷ Theory search
    repeat
        remove the first couple ($Cl, LL$) from $TC$
        ($LL', Th'$) $\leftarrow$EMBLEM($Th \cup \{Cl\}, D, NEM, \epsilon, \delta$)
        if $LL' > ThLL$ then
            $Th \leftarrow Th'$, $ThLL \leftarrow LL'$
        end if
    until $TC$ is empty
    $Th \leftarrow Th \bigcup_{(Cl,LL) \in BC} \{Cl\}$
    ($LL, Th$) $\leftarrow$EMBLEM($Th, D, NEM, \epsilon, \delta$)
    return $Th$
end function
```

3. a space of possible clauses L

find a hypothesis $H \subseteq$ *so that the absolute error* $AE = \sum_{i=1}^{T} |P(e_i) - p_i|$ *is minimized, i.e.,*

$$argmin_{H \in L} \sum_{i=1}^{T} |P(e_i) - p_i|$$

ProbFOIL+ generalizes the mFOIL system [34], itself a generalization of FOIL [103]. It performs a hill climbing search in the space of programs. It is consists of a covering loop in which one rule is added to the final program at each iteration. The covering loop ends when a condition based on a global scoring function (generally the accuracy over the dataset) is satisfied. The rule added at each iteration is obtained by a nested loop which iteratively adds literals to the body of the rule and performs a beam search in the space of clauses (as in mFOIL) guided by a local scoring function, generally an *m-estimate* [77] of the *precision*. Algorithm 5 shows the overall approach[1].

Algorithm 5 Function ProbFOIL+

```
function ProbFOIL+($target$)
    $H \leftarrow \emptyset$
    while true do
        $clause \leftarrow$ LearnRule($H, target$)
        if GScore($H$) < GScore($H \cup \{clause\}$) $\wedge$ Significant($H, clause$) then
            $H \leftarrow H \cup \{clause\}$
        else
            return $H$
        end if
    end while
end function
function LearnRule($H, target$)
    $candidates \leftarrow \{x :: target \leftarrow true\}$
    $best \leftarrow (x :: target \leftarrow true)$
    while $candidates \neq \emptyset$ do
        $next\_cand \leftarrow \emptyset$
        for all $x :: target \leftarrow body \in candidates$ do
            for all $(target \leftarrow bod, refinement) \in \rho(target \leftarrow body)$ do
                if not Reject($H, best, (x :: target \leftarrow body, refinement)$) then
                    $next\_cand \leftarrow next\_cand \cup \{(x :: target \leftarrow body, refinement)\}$
                    if LScore($H, (x :: target \leftarrow body, refinement)$) > LScore($H, best$) then
                        $best \leftarrow (x :: target \leftarrow body, refinement)$
                    end if
                end if
            end for
        end for
        $candidates \leftarrow next\_cand$
    end while
    return $best$
end function
```

[1]The description of ProbFOIL+ is based on [105] and the code at `https://bitbucket.org/antondries/prob2foil`

Part V

Lifted Probabilistic Logic Programming

Chapter 11

Liftable Probabilistic Logic Programming

PLP provides a powerful tool for reasoning with uncertain relational models. However, learning probabilistic logic programs is expensive due to the high cost of inference. Among the proposals to overcome this problem, one of the most promising is lifted inference. In this chapter we consider PLP models that are amenable to lifted inference and present an algorithm for performing parameter and structure learning of these models from positive and negative examples. We discuss parameter learning with Expectation Maximization (EM) and Limited-memory Broyden–Fletcher–Goldfarb–Shanno (LBFGS) and structure learning with LIFTCOVER [88], an algorithm similar to SLIPCOVER [11]. The results of the comparison of LIFTCOVER with SLIPCOVER on 12 datasets show that it can achieve solutions of similar or better quality in a fraction of the time.

The chapter is organized as follows: After a brief motivation in Section 11.1, Sections 11.2 and 11.3 describe the liftable PLP language and how to perform inference in liftable PLP respectively. Sections 11.4 and 11.5 illustrate parameter and structure learning respectively. Section 11.6 discusses related work and Section 11.7 presents the experiments.

11.1 Motivation

The problem of learning probabilistic logic program has received considerable attention. However, PLP usually require expensive learning procedures due to the high cost of inference. SLIPCOVER [11] for example performs structure learning of probabilistic logic programs using knowledge compilation for parameter learning: the expectations needed for the EM parameter learning algorithm are computed using the Binary Decision Diagrams (BDDs) that are built for inference. Compiling explanations for queries into BDDs has a #P cost in the number of random variables. Lifted inference [100] was proposed for improving the performance of reasoning in probabilistic relational models by reasoning on whole populations of individuals instead of considering each individual separately. For example, consider the following Logic Program with Annotated Disjunctions (adapted from [28]):

$$\mathit{popular}(X) : p :- \mathit{friends}(X, Y), \mathit{famous}(Y)$$

which states that a person is popular with probability $p \in [0, 1]$ if he/she has a famous friend. To compute the probability that *john* is popular, let n be the number of *john*'s famous friends. *john* is not popular if the rule does not fire for any his famous friends therefore $P(\neg \mathit{popular}(\mathit{john})) = (1 - p)^n$. So $P(\mathit{popular}(\mathit{john})) = 1 - (1 - p)^n$. To compute this probability, we do not need to know information about *john*'s individual famous friends, we just need to know their number. Computing this value has a cost logarithmic in n, as computing a^n is $\Theta(\log n)$ with the "square and multiply" algorithm [39], rather than #P in n.

Various algorithms have been proposed for performing lifted inference for PLP [139, 7], see [113] for a survey and comparison of the approaches.

In this chapter, we consider a simple PLP language (called *liftable PLP*) in which programs contain clauses with a single annotated atom in the head and the predicate of this atom is the same for all clauses. In this case, all the above approaches for lifted inference coincide and reduce to a computation similar to the one of the example above.

For this language, we discuss how to perform discriminative parameter learning by using EM or optimizing the likelihood with Limited-memory BFGS (LBFGS) [92]. A previous approach for performing lifted learning [141] targeted generative learning for Markov Logic Networks, so it cannot be applied directly to PLP.

We also present LIFTCOVER for “LIFTed slipCOVER”, an algorithm for performing discriminative structure learning of liftable PLP programs obtained from SLIPCOVER [11] by simplifying the search for structure and replacing parameter learning with one of the specialized approaches. We thus obtain LIFTCOVER-EM and LIFTCOVER-LBFGS that perform EM and LBFGS respectively.

Liftable PLP can also be seen as a language where the contributions of different groundings of a clause and of different clauses are combined using a noisy-OR combining rule and is therefore very much related to languages such as First-Order Probabilistic Logic [67], Bayesian Logic Programs [56] and the First-Order Conditional Influence Language [84]. Liftable PLP can be seen as a special case of each of these languages in which simpler inference and learning algorithms can be used. The experimental results show that the algorithm still yield good quality results notwithstanding the language restrictions.

11.2 Liftable PLP

In order to speed up the inference, we restrict the language of LPADs by allowing only clauses of the form

$$C_i = h_i : \Pi_i :- \, b_{i1}, \ldots, b_{iu_i}$$

in the program where all the clauses share the same predicate for the single atom in the head. Let us call this predicate *target/a* with a the arity. The literals in the body have predicates other than *target/a* and are defined by facts and rules that are certain, i.e., they have a single atom in the head with probability 1. The predicate *target/a* is called *target* and the others *input predicates*. Suppose there are n probabilistic clauses of the form above in the program; we call this language *liftable PLP*. Now let us explain, in the next section, how to perform inference programs belonging to such language.

11.3 Inference in Liftable PLP

The problem of performing inference, as described in Section 7.4, is to compute the probability of a ground instantiation q of $target/a$. This can be done at the lifted level. In *liftable PLP*, we should first find the number of ground instantiations of clauses for $target/a$ such that the body is true and the head is equal to q. Suppose there are m_i such instantiations $\{\theta_{i1}, \ldots, \theta_{im_i}\}$, for rule C_i for $i = 1, \ldots, n$. Each instantiation θ_{ij} corresponds to a random variable X_{ij} taking values 1 with probability Π_i and 0 with probability $1 - \Pi_i$. The query q is true if at least one of the random variables for a rule takes value 1: $q = true \Leftrightarrow \bigvee_{i=1}^{n} \bigvee_{j=1}^{m_i} (X_{ij} = 1)$. In other words q is false only if no random variable takes value 1. All the random variables are mutually independent so the probability that none takes value 1 is $\prod_{i=1}^{n} \prod_{j=1}^{m_i} (1 - \Pi_i) = \prod_{i=1}^{n} (1 - \Pi_i)^{m_i}$ and the probability of q being true is $P(q) = 1 - \prod_{i=1}^{n} (1 - \Pi_i)^{m_i}$. So once the number of clause instantiations with the body true is known, the probability of the query can be computed in logarithmic time. Note that finding an assignment of a set of logical variables that makes a conjunction true is an NP-complete problem [58], therefore computing the probability of the query may be prohibitive.

When using knowledge compilation, to the cost of finding the assignment, we must sum the cost of performing the compilation, that is #P in the number of satisfying logical variables assignments (clause instantiations with the body true). Therefore inference in liftable PLP is significantly cheaper than in the general case. Moreover, in machine learning the conjunctions are usually short and the knowledge compilation cost dominates.

The general language of PLP is necessary when the user wants to induce a knowledge base or an ontology regarding the domain. In that case, the possibility of having more than one head, possibly involving more than one predicate, and the possibility of learning probabilistic rules for subgoals is useful because the resulting program can thus represent and organize general knowledge about the domain. Moreover, the resulting program can then be used for answering different types of queries instead of being restricted to answering queries about a single predicate. This is similar to the problem of learning multiple predicates in Inductive Logic Programming. While this problem has received considerable attention, most work concentrated on learning a single predicate, for example systems such as FOIL [103], Progol [79] and Aleph [129] learn a single predicate at a time. Furthermore, most benchmark datasets are focused on predicting the truth value of atoms for a single predicate. For example, all the datasets we consider in the experimental evaluation, Section 11.7, include positive and negative example for a single predicate. Therefore we concentrate here on predicting a single predicate.

We believe that the problem of inducing general knowledge bases will become very important in the near future because of the growth of the Semantic Web: more and more data is being published on the web but ontologies are often shallow. If we want to be able to provide answers for complex queries given the available data, we need deep and complex knowledge bases and learning them appears to be a promising direction.

We can picture the dependence of the random variable q associated with the query from the random variables of clause groundings with the body true as in Figure 11.1. Here the Conditional Probability Table (CPT) of q is that of an or: $P(q) = 1$ if at least one of its parents is equal to 1 and 0 otherwise. The variables from clause groundings are

$$\{X_{11}, \ldots, X_{1m_1}, X_{21}, \ldots, X_{2m_2}, \ldots, X_{n1}, \ldots, X_{nm_n}\}.$$

These are parentless variables, with X_{ij} having the CPT $P(X_{ij} = 1) = \Pi_i$ and $P(X_{ij} = 0) = 1 - \Pi_i$.

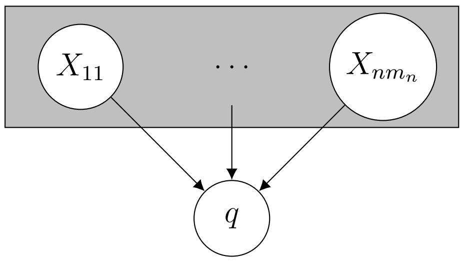


Figure 11.1: Bayesian Network representing the dependency between the query q and the random variables associated with groundings of the clauses with the body true.

This is an example of a *noisy-OR* model [38, 93]: an event is associated to a number of conditions each of which alone can cause the event to happen.

The conditions/causes are noisy, i.e., they have a probability of being active and they are mutually unconditionally independent. A liftable PLP program encodes a noisy-OR model where the event is the query q being true and causes are the ground instantiations of the clauses that have the body true: each can cause the query to be true with the probability given by the clause annotation.

Example 21. *Let us consider the UWCSE domain [64] where the objective is to predict the "advised_ by" relation between students and professors. In this case a program for advised_by/2 may be*

$$
\begin{array}{l}
advised\_by(A,B) : 0.3 :- \\
\quad student(A), professor(B), project(C,A), project(C,B). \\
advised\_by(A,B) : 0.6 :- \\
\quad student(A), professor(B), ta(C,A), taught\_by(C,B). \\
student(harry).\ professor(ben). \\
project(pr_1, harry).\ project(pr_1, ben). \\
project(pr_2, harry).\ project(pr_2, ben). \\
project(pr_3, harry).\ project(pr_3, ben). \\
project(pr_4, harry).\ project(pr_4, ben). \\
taught\_by(c_1, ben).\ ta(c_1, harry). \\
taught\_by(c_2, ben).\ ta(c_2, harry).
\end{array}
$$

where $project(C,A)$ means that C is a project with participant A, $ta(C,A)$ means that A is a teaching assistant for course C and $taught\_by(C,B)$ means that course C is taught by B. The probability that a student is advised by a professor depends on the number of joint projects and the number of courses the professor teaches where the student is a TA, the higher these numbers the higher the probability.

Suppose we want to compute the probability of $q = advised\_by(harry, ben)$ where harry is a student, ben is a professor, they have 4 joint projects and ben teaches 2 courses where harry is a TA as stated by the facts in the program. Then the first clause has 4 groundings with head q where the body is true, the second clause has 2 groundings with head q where the body is true and

$$P(advised\_by(harry, ben)) = 1 - (1 - 0.3)^4(1 - 0.6)^2 = 0.9615.$$

11.4 Parameter Learning

Learning problems can be divided into discriminative and generative [66]. Given the input data $\mathbf{x}$, while *generative learning* learns the joint distribution $P(\mathbf{x})$, *discriminative learning* instead identifies one of the data variables y which we want to predict and learns the conditional distribution $P(y|\mathbf{x})$. If y is Boolean, as in our case, it is natural to identify values $y = 1$ as positive examples and values $y = 0$ as negative examples. In generative learning identifying positive and negative examples is less obvious. We consider discriminative learning because we want to predict only atoms for the target predicate, while the atoms for the input predicates are assumed as given.

The problem of discriminative learning of the parameters of a liftable PLP $T = \{C_1, \ldots, C_n\}$ can be expressed as follows: given a liftable PLP T, a set

$$E^+ = \{e_1, \ldots, e_Q\}$$

of positive examples (ground atoms for the target predicate) and a set

$$E^- = \{e_{Q+1}, \ldots, e_R\}$$

of negative examples (ground atoms for the target predicate) and background knowledge B, find the parameters of T such that the likelihood

$$L = \prod_{q=1}^{Q} P(e_q) \prod_{r=Q+1}^{R} P(\neg e_r)$$

is maximized. The likelihood is given by the product of the probability of each example.

The background knowledge B is a normal logic program defining the input predicates with certainty. In the simplest case it is a set of ground facts, i.e. an interpretation I, describing the domain by means of the observed facts for the input predicates. It is also called a *mega-example* because we can consider the case where we have a set of interpretations $\mathcal{I} = \{I_1, \ldots, I_U\}$ each describing a different sub-domain from the universe considered. In that case, each mega-example I_u will be associated with its set of positive and negative examples E_u^+ and E_u^- that are to be evaluated against I_u. These examples can be opportunely encoded in I_u so that the training data is represented fully by $\mathcal{I}$, for example by encoding positive examples as facts for the target predicate and negative examples as facts of the form $neg(e_r)$ with $e_r \in E_u^-$. This is a common situation in StarAI. The likelihood can be unfolded to

$$L = \prod_{q=1}^{Q} \left(1 - \prod_{l=1}^{n} (1 - \Pi_l)^{m_{lq}}\right) \prod_{r=Q+1}^{R} \prod_{l=1}^{n} (1 - \Pi_l)^{m_{lr}} \tag{11.1}$$

where m_{lq} (m_{ir}) is the number of instantiations of C_l whose head is e_q (e_r) and whose body is true. We can aggregate the negative examples

$$L = \prod_{l=1}^{n} (1 - \Pi_l)^{m_{l-}} \prod_{q=1}^{Q} \left(1 - \prod_{l=1}^{n} (1 - \Pi_l)^{m_{lq}}\right) \tag{11.2}$$

where $m_{l-} = \sum_{r=Q+1}^{R} m_{lr}$.

L can be maximized using an Expectation Maximization (EM) algorithm [32] since the X_{ij} variables are hidden. To perform EM, we need to compute the conditional probabilities $P(X_{ij} = 1|e)$ and $P(X_{ij} = 1|\neg e)$ where e is an example (a ground atom) and X_{ij} are its parents.

Alternatively, we can use gradient descent to optimize L. In this case, we need to compute the gradient of the likelihood with respect to the parameters. In the following subsections we consider each method in turn.

11.4.1 EM Algorithm

The EM algorithm [32] as described in Section 10.2.1 finds the maximum likelihood estimates of parameters in models with hidden variables by alternating between an expectation (E) step and a maximization (M) step. The algorithm starts with random values for the parameters. Then, in the E step, it computes the distribution of values of the hidden variables given the observed ones and the current value of the parameters. In the M step it computes the value of the parameters that maximize the log-likelihood (LL). Then the parameters are updated and the algorithm goes back to the E step, stopping when the log-likelihood does not improve anymore.

To perform EM, we need to compute the distribution of the hidden variables given the observed ones, in our case $P(X_{ij} = 1|e)$ and $P(X_{ij} = 1|\neg e)$. e is a single example that is a ground atom for the target predicate. The X_{ij} variables are relative to the ground instantiations of the probabilistic clauses whose body is true when the head is unified with e. Different examples do not share clause groundings, as the constants in them are different. Therefore the X_{ij} variables are not shared among examples.

Let us now compute $P(X_{ij} = 1, e)$:

$$P(X_{ij} = 1, e) \;=\; P(e|X_{ij} = 1)P(X_{ij} = 1) = P(X_{ij} = 1) = \Pi_i$$

since $P(e|X_{ij} = 1) = 1$, so

$$\begin{aligned} P(X_{ij} = 1|e) &= \frac{P(X_{ij} = 1, e)}{P(e)} = \frac{\Pi_i}{1 - \prod_{i=1}^{n}(1 - \Pi_i)^{m_i}} && (11.3) \\ P(X_{ij} = 0|e) &= 1 - \frac{\Pi_i}{1 - \prod_{i=1}^{n}(1 - \Pi_i)^{m_i}} && (11.4) \end{aligned}$$

$P(X_{ij} = 1|\neg e)$ is given by

$$P(X_{ij} = 1, \neg e) \;=\; P(\neg e|X_{ij} = 1)P(X_{ij} = 1) = 0$$

since $P(\neg e|X_{ij} = 1) = 0$, so

$$\begin{aligned} P(X_{ij} = 1|\neg e) &= 0 && (11.5) \\ P(X_{ij} = 0|\neg e) &= 1. && (11.6) \end{aligned}$$

This leads to the EM algorithm of Algorithm 6, with the EXPECTATION and MAXIMIZATION functions shown in Algorithms 7 and 8. Function EM stops when the difference between the current value of the LL and the previous one is below a given threshold or when such a difference relative to the absolute value of the current one is below a given threshold.

Function EXPECTATION updates, for each clause C_i, two counters, c_{i1} and c_{i2}, one for each value of the random variables X_{ij} associated with clause C_i. These counters accumulate the values of the conditional probability of the values of the hidden variables. The counters are updated taking into account first the negative examples and then the positive ones. Negative examples can

Algorithm 6 Function EM

```
function EM($restarts, max\_iter, \epsilon, \delta$)
    $BestLL \leftarrow -inf$
    $BestPar \leftarrow []$
    for $j \leftarrow 1, restarts$ do
        for $i \leftarrow 1, n$ do                                  ▷ $n$: number of rules
            $\Pi_i \leftarrow random$
        end for
        $LL = -inf$
        $iter \leftarrow 0$
        repeat
            $iter \leftarrow iter + 1$
            $LL_0 = LL$
            $LL =$ EXPECTATION
            MAXIMIZATION
        until $LL - LL_0 < \epsilon \vee LL - LL_0 < -LL \cdot \delta \vee iter > max\_iter$
        if $LL > BestLL$ then
            $BestLL \leftarrow LL$
            $BestPar \leftarrow [\Pi_1, \ldots, \Pi_n]$
        end if
    end for
    return $BestLL, BestPar$
end function
```

be considered in bulk because their contribution is the same for all groundings of all examples, while positive examples must be considered one by one, for each one updating the counters of all the clauses.

Algorithm 7 Function Expectation

```
function EXPECTATION
    $LL \leftarrow \sum_{i \in Rules} m_{i-} \log(1 - \Pi_i)$
                    ▷ $m_{i-}$: total number of groundings of rule $i$ with the body true in a negative example
    for $i \leftarrow 1, n$ do
        $c_{i1} \leftarrow 0$
        $c_{i2} \leftarrow m_{i-}$
    end for
    for $r \leftarrow 1, P$ do                                      ▷ $P$: number of positive examples
        $probex \leftarrow 1 - \prod_{i \in Rules} (1 - \Pi_i)^{m_{ir}}$
                    ▷ $m_{ir}$: number of groundings of rule $i$ with the body true in example $r$
        $LL \leftarrow LL + \log probex$
        for $i \leftarrow 1, n$ do
            $condp \leftarrow \frac{\Pi_i}{probex}$
            $c_{i1} \leftarrow c_{i1} + m_{ir} condp$
            $c_{i2} \leftarrow c_{i2} + m_{ir}(1 - condp)$
        end for
    end for
    return $LL$
end function
```

Function Maximization then simply computes the new values of the parameters by dividing c_{i1} by the sum of c_{i1} and c_{i2}.

Algorithm 8 Function Maximization

```
1: procedure MAXIMIZATION
2:     for $i \leftarrow 1, n$ do
3:         $\Pi_i = \frac{c_{i1}}{c_{i1}+c_{i2}}$
4:     end for
5: end procedure
```

11.4.2 Gradient-Based Optimization

Gradient-based methods include gradient descent and its derivatives, such as second-order methods like Limited-memory BFGS (LBFGS) [92], an optimization algorithm in the family of quasi-Newton methods that approximates the Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm using a limited amount of computer memory, see Section 10.2.3.

To perform gradient-based optimization we need to compute the partial derivatives of the likelihood with respect to the parameters. Let us recall the likelihood

$$L = \prod_{l=1}^{n}(1-\Pi_l)^{m_{l-}} \prod_{q=1}^{Q}\left(1-\prod_{l=1}^{n}(1-\Pi_l)^{m_{lq}}\right) \tag{11.7}$$

where $m_{l-} = \sum_{r=Q+1}^{R} m_{lr}$. Its partial derivative with respect to Π_i is

$$\begin{aligned}
\frac{\partial L}{\partial \Pi_i} &= \frac{\partial \prod_{l=1}^{n}(1-\Pi_l)^{m_{l-}}}{\partial \Pi_i}\prod_{q=1}^{Q}\left(1-\prod_{l=1}^{n}(1-\Pi_l)^{m_{lq}}\right) \\
&\quad + \prod_{l=1}^{n}(1-\Pi_l)^{m_{l-}}\frac{\partial \prod_{q=1}^{Q}(1-\prod_{l=1}^{n}(1-\Pi_l)^{m_{lq}})}{\partial \Pi_i} \\
&= -m_{i-}(1-\Pi_i)^{m_{i-}-1}\prod_{l=1,l\neq i}^{n}(1-\Pi_l)^{m_{l-}}\prod_{q=1}^{Q}\left(1-\prod_{l=1}^{n}(1-\Pi_l)^{m_{lq}}\right) \\
&\quad + \prod_{l=1}^{n}(1-\Pi_l)^{m_{l-}}\sum_{q=1}^{Q} m_{iq}(1-\Pi_i)^{m_{iq}-1}\prod_{l=1,l\neq i}^{n}(1-\Pi_l)^{m_{lq}} \\
&\quad \cdot \prod_{q'=1,q'\neq q}^{Q}\left(1-\prod_{l=1}^{n}(1-\Pi_l)^{m_{lq'}}\right)
\end{aligned} \tag{11.8}$$

for the differentiation product rule, and

$$\begin{aligned}\frac{\partial L}{\partial \Pi_i} &= -m_{i-}(1-\Pi_i)^{m_{i-}-1}\prod_{l=1,l\neq i}^{n}(1-\Pi_l)^{m_{l-}}\frac{(1-\Pi_i)^{m_{i-}}}{(1-\Pi_i)^{m_{i-}}}\\ &\prod_{q=1}^{Q}\left(1-\prod_{l=1}^{n}(1-\Pi_l)^{m_{lq}}\right)+\\ &\prod_{l=1}^{n}(1-\Pi_l)^{m_{l-}}\sum_{q=1}^{Q}m_{iq}\frac{\prod_{l=1}^{n}(1-\Pi_l)^{m_{lq}}}{1-\Pi_i}\\ &\prod_{q'=1,q'\neq q}^{Q}\left(1-\prod_{l=1}^{n}(1-\Pi_l)^{m_{lq'}}\right)\cdot\frac{1-\prod_{l=1}^{n}(1-\Pi_l)^{m_{lq}}}{1-\prod_{l=1}^{n}(1-\Pi_l)^{m_{lq}}} \qquad (11.9)\end{aligned}$$

by dividing and multiplying for $(1-\Pi_i)^{m_{i-}}$, $(1-\Pi_i)$ and $1-\prod_{l=1}^{n}(1-\Pi_l)^{m_{lq}}$ various factors. Then

$$\begin{aligned}\frac{\partial L}{\partial \Pi_i} &= -\frac{m_{i-}(1-\Pi_i)^{m_{i-}-1}L}{(1-\Pi_i)^{m_{i-}}}+\sum_{q=1}^{Q}\frac{m_{iq}\prod_{l=1}^{n}(1-\Pi_l)^{m_{lq}}L}{(1-\Pi_i)(1-\prod_{l=1}^{n}(1-\Pi_l)^{m_{lq}})}\\ &= -\frac{m_{i-}L}{1-\Pi_i}+\sum_{q=1}^{Q}\frac{m_{iq}\prod_{l=1}^{n}(1-\Pi_l)^{m_{lq}}L}{(1-\Pi_i)(1-\prod_{l=1}^{n}(1-\Pi_l)^{m_{lq}})}\\ &= \frac{L}{1-\Pi_i}\left(\sum_{q=1}^{Q}\frac{m_{iq}\prod_{l=1}^{n}(1-\Pi_l)^{m_{lq}}}{1-\prod_{l=1}^{n}(1-\Pi_l)^{m_{lq}}}-m_{i-}\right)\\ &= \frac{L}{1-\Pi_i}\left(\sum_{q=1}^{Q}m_{iq}\frac{1-P(e_q)}{P(e_q)}-m_{i-}\right)\\ &= \frac{L}{1-\Pi_i}\left(\sum_{q=1}^{Q}m_{iq}\left(\frac{1}{P(e_q)}-1\right)-m_{i-}\right) \qquad (11.10)\end{aligned}$$

by simple algebra.

The equation $\frac{\partial L}{\partial \Pi_i}=0$ does not admit a closed form solution, not even when there is a single clause, so we must use optimization to find the maximum of L.

11.5 Structure Learning

The discriminative structure learning problem can be expressed as follows: given a set $E^+=\{e_1,\ldots,e_Q\}$ of positive examples, a set $E^-=\{e_{Q+1},\ldots,e_R\}$ of negative examples and a background knowledge B, find a liftable PLP T such that the likelihood is maximized. The background knowledge B may be a normal logic program defining all the predicates (the input predicates) except the target predicate.

We solve this problem by first identifying good clauses guided by the log likelihood (LL) of the data. Clauses are found by a top-down beam search. The refinement operator adds a literal to the body of the current clause, the literal is taken from a *bottom clause* built as in Progol [79]. The set of clauses found in this phase is then considered as a single theory and parameter learning is performed on it. Then the clauses with a parameter below a user define threshold *WMin* are discarded and the theory is returned. The resulting algorithm, LIFTCOVER, is very similar to SLIPCOVER [11]. The difference between the two is that LIFTCOVER uses lifted parameter learning instead of the EM algorithm over BDDs of [10]. Moreover they use a different approach for performing the selection of the rules to be included in the final model: while SLIPCOVER does a hill-climbing search in which it adds one clause at a time to the theory, learns the parameters and keeps the clause if the LL is smaller than before, LIFTCOVER learns the parameters for the whole set of clauses found during the search in the space of clauses. This is allowed by the fact that parameter learning in LIFTCOVER is much faster so it can be applied to large theories. Then rules with a small parameter can be discarded as they provide small contributions to the predictions. In practice structure search is thus performed in LIFTCOVER by parameter learning, as is done for example in [90, 148].

Algorithm 9 Function LIFTCOVER

```
1: function LIFTCOVER($NB, NI, NInt, NS, NA, NV$)
2:     $Beam$ =InitialBeam($NInt, NS, NA$)                         ▷ Bottom clauses building
3:     $CC \leftarrow \emptyset$
4:     $Steps \leftarrow 1$
5:     $NewBeam \leftarrow []$
6:     repeat
7:         Remove the first couple $((Cl, Literals), LL)$ from $Beam$        ▷ Remove the first clause
8:         $Refs \leftarrow$ClauseRefinements($(Cl, Literals, NV)$)   ▷ Find all refinements $Refs$ of $(Cl, Literals)$
9:         for all $(Cl', Literals') \in Refs$ do
10:            $(LL'', \{Cl''\}) \leftarrow$LearnWeights($I, \{Cl'\}$)
11:            $NewBeam \leftarrow$Insert($(Cl'', Literals'), LL'', NewBeam, NB$)  ▷ The refinement is inserted in
    the beam in order of likelihood, possibly removing the last clause if the size of the beam $NB$ is exceeded
12:            $CC \leftarrow CC \cup \{Cl'\}$
13:        end for
14:        $Beam \leftarrow NewBeam$
15:        $Steps \leftarrow Steps + 1$
16:    until $Steps > NI$ or $Beam$ is empty
17:    $(LL, Th) \leftarrow$LearnWeights($CC$)
18:    Remove from $Th$ the clauses with a weight smaller than $WMin$
19:    return $Th$
20: end function
```

Algorithm 9 shows the main LIFTCOVER function. Line 2 calls INITIALBEAM (see Algorithm 10) that builds an initial beam *Beam* consisting of bottom clauses.

Algorithm 10 Function INITIALBEAM

```
1: function INITIALBEAM($NInt, NS, NA$)
2:     $Beam \leftarrow []$
3:     for all modeh declarations $modeh(r, s)$ do
4:         for $i = 1 \rightarrow NInt$ do
5:             Select randomly a mega-example $I$
6:             for $j = 1 \rightarrow NA$ do
7:                 Select randomly an atom $h$ from $I$ matching $schema(s)$
8:                 Bottom clause $BC \leftarrow$SATURATION($h, r, NS$), let $BC$ be $Head$ :− $Body$
9:                 $Beam \leftarrow [((h : 0.5 :- true, Body), -\infty)|Beam]$
10:            end for
11:        end for
12:    end for
13:    return $Beam$
14: end function
```

11.5.1 Language bias

The set of literals allowed in the bottom clause is defined by the *language bias* that is expressed by means of *mode* declarations. They are atoms of the form $modeh(r, s)$ (head declarations) or $modeb(r, s)$ (body declaration), where s, the *schema*, is a ground literal and r is an integer called the *recall*. A schema is a template for literals in the head or body of a clause and can contain special placemarker terms of the form `#type`, `+type` and `-type`, which stand, respectively, for ground terms, input variables and output variables of a type. An input variable in a body literal of a clause must be either an input variable in the head or an output variable in a preceding body literal in the clause. If M is a set of mode declarations, $L(M)$ is the *language of* M, i.e. the set of clauses $\{C = h :- b_1, \ldots, b_m\}$ such that the head atom h (resp. body literals b_i) is obtained from some head (resp. body) declaration in M by replacing all `#type` placemarkers with ground terms and all `+type` (resp. `-type`) placemarkers with input (resp. output) variables. We extend this type of mode declarations with placemarker terms of the form `-#type`, which are treated as `#` when defining $L(M)$ but differ in the creation of the bottom clauses, see below.

11.5.2 Bottom Clauses Generation

The bottom clause is the clause with the longest true body in $L(M)$. The bottom clause is built with a method called *saturation*: an example e is randomly selected and the set of atoms *Body* that are true regarding the example e is built incrementally. by considering the constants in e and querying the background for true atoms regarding these constants. A list of constants is kept and it is enlarged with those in `-type` placemarkers in the answers to the queries. The recall indicates how many answers to the queries must be considered. Besides an integer, it may be the symbol `*`, indicating all answers.

The procedure is iterated a user-defined number of times. Then a bottom clause is obtained from the clause $e \leftarrow Body$ by replacing ground terms with variables respecting the mode declarations. Placemarkers `-#type` are treated as `#type` when variabilizing because they are not replaced by variables but as `-type` place-markers when building the *Body* because terms in those positions are added to the current list of constants.

Function SATURATION, shown in Algorithm 11, builds a bottom clause for an example $Head$, where *NS* is a user-defined number of saturation steps to be performed.

11.5.3 Clause refinement

In SLIPCOVER, a beam is a set of tuples $((Cl, Literals), LL)$ with Cl a clause, $Literals$ the set of literals admissible in the body of Cl and LL the log-likelihood of Cl. Function INITIALBEAM, shown in Algorithm 10, returns an initial beam containing tuples $((h : 0.5 :- \mathit{true}, Literals), -\infty)$ for each bottom clause $h : 0.5 :- Literals$. The likelihood is initialized to $-\infty$.

Then LIFTCOVER runs a beam search in the space of clauses for the target predicate. In each beam search iteration, the first clause of the beam is removed and all its refinements are computed. Each refinement Cl' is scored by performing parameter learning with $T = \{Cl'\}$ and using the resulting LL as the heuristic. The scored refinements are inserted back into the beam in order of heuristic. If the beam exceeds a maximum user-defined size, the bottom elements are removed. Moreover, the refinements are added to a set of clauses CC.

Algorithm 11 Function SATURATION

```
function SATURATION($Head, r, NS$)
    $InTerms \leftarrow \emptyset$,
    $BC = \emptyset$                                            ▷ $BC$: bottom clause
    for all arguments $t$ of $Head$ do
        if $t$ corresponds to a $+type$ then
            add $t$ to $InTerms$
        end if
    end for
    Let $BC$'s head be $Head$
    repeat
        $Steps \leftarrow 1$
        for all modeb declarations $modeb(r, s)$ do
            for all possible subs. $\sigma$ of variables corresponding to $+type$ in $schema(s)$ by terms from $InTerms$ do
                for $j = 1 \rightarrow r$ do
                    if goal $b = schema(s)$ succeeds with answer substitution $\sigma'$ then
                        for all $v/t \in \sigma$ and $\sigma'$ do
                            if $v$ corresponds to a $-type$ or $-\#type$ then
                                add $t$ to the set $InTerms$ if not already present
                            end if
                        end for
                        Add $b$ to $BC$'s body
                    end if
                end for
            end for
        end for
        $Steps \leftarrow Steps + 1$
    until $Steps > NS$
    Replace constants with variables in $BC$, using the same variable for equal terms
    return $BC$
end function
```

For each clause Cl with $Literals$ admissible in the body, Function CLAUSEREFINEMENTS, shown in Algorithm 12, computes refinements by adding a literal from $Literals$ to the body. Furthermore, the refinements must respect the input-output modes of the bias declarations, must be connected (i.e., each body literal must share a variable with the head or a previous body literal) and their number of variables must not exceed a user-defined number NV. Refinements are of the form (Cl', L') where Cl' is the refined clause Cl' and L' is the new set of literals allowed in the body of Cl'.

Beam search is iterated a user-defined number of times or until the beam becomes empty. The output of this search phase is represented by the set CC of clauses. Then parameter learning is applied to the whole set CC, i.e., $T = CC$. Finally clauses with a weight smaller than $WMin$ are removed.

The separate search for clauses has similarity with the covering loop of ILP systems such as Aleph [129] and Progol [79]. Differently from the ILP case, however, the positive examples covered are not removed from the training set because coverage is probabilistic, so an example that is assigned nonzero probability by a clause may have its probability increased by further clauses. The selection of clauses is performed by parameter learning: clauses with very small weights are removed.

Algorithm 12 Function ClauseRefinements

```
1: function ClauseRefinements($(Cl, Literals), NV$)
2:     $Refs = \emptyset$, $Nvar = 0$;        ▷ Nvar:number of different variables in a clause
3:     for all $b \in Literals$ do
4:         $Literals' \leftarrow Literals \setminus \{b\}$
5:         Add $b$ to $Cl$ body obtaining $Cl'$
6:         $Nvar \leftarrow$ number of $Cl'$ variables
7:         if $Cl'$ is connected $\wedge$ $Nvar < NV$ then
8:             $Refs \leftarrow Refs \cup \{(Cl', Literals')\}$
9:         end if
10:    end for
11:    return $Refs$
12: end function
```

11.6 Related Work

We first consider the work related to liftable PLP from the field of lifted inference and then that from the field of probabilistic rule learning.

Lifted inference for PLP under the distribution semantics is surveyed by [113] that discuss three approaches.

LP^2 [7] uses an algorithm that extends Generalized Counting First Order Variable Elimination (GC-FOVE) [135] for taking into account clauses that have variables in bodies not appearing in the head (existentially quantified variables). Weighted First Order Model Counting (WFOMC) [139] uses a Skolemization algorithm that eliminates existential quantifiers from a theory without changing its weighted model count. Kisynski and Poole [63] proposed an approach based on Aggregation Par-factors that can represent noisy-OR models. The three approaches have been compared experimentally in [113] for general PLP and WFOMC was found the fastest.

Relational Logistic Regression [52] is a generalization of logistic regression that can also be applied to PLP.

LP^2, Aggregation Parfactors and Relational Logistic Regression reduce to the same algorithm for performing inference when the language is restricted to liftable PLP. LP^2 is based on GC-FOVE that in turn is an extension of Variable Elimination (VE) [154, 155]. VE was designed from the start to be able to exploit *causal independence*, the situation where multiple causes contribute independently to a common effect. Noisy-OR is a prominent example of causal independence. The capacity of VE to deal with noisy-OR is exploited in LP^2 to aggregate the contributions of multiple ground clause to the probability of the same atom in a lifted way., i.e., without generating the groundings.

We now discuss Aggregation Par-factors and Relational Logistic Regression. We first introduce *parametrized random variables* (PRV) that are represented by logical atoms. Each logical variable in a PRV is typed with a population. A *parfactor* is a triple $\langle \mathcal{C}, V, F \rangle$ where $\mathcal{C}$ is a set of inequality constraints on parameters (logical variables), V is a set of PRV and F is a factor that is a function from the Cartesian product of ranges of PRVs in V to real values.

Aggregation parfactors [63] can represent different kind of causal independence models, of which noisy-OR and noisy-MAX are special cases. Aggregation par-factors are a generalization of parfactors that are defined over two of PRVs one of which contains one more logical variable that the other. Therefore, the contributions of the PRV with the extra logical variable have to be aggregated and this is done by converting the aggregation par-factor into two regular par-factors. We can correctly encode liftable PLP with aggregation parfactors obtaining the same formula for calculating the probability of queries. Relational Logistic Regression [52] generalizes logistic regression, where the probability of a child Boolean random variable Q is modeled on the basis of the values of parent random variables $\{X_1, \ldots, X_n\}$ as

$$P(q|X_1, \ldots, X_n) = \text{sigmoid}(w_0 + \sum_i w_i X_i)$$

where $q \equiv (Q = \mathit{true})$ and $\text{sigmoid}(x) = 1/(1 + e^{-x})$. For the case of Boolean variables, we can assume that the values are encoded with 0 for false and 1 for true.

To apply logistic regression to the relational case, the authors introduce the notion of *weighted parent formula* (WPF) for a PRV $Q(X)$, where X is a set of logical variables: a WPF is a triple $\langle L, F, w_i \rangle$ where L is a set of logical variables for which $L \cap X = \emptyset$, F is a Boolean formula of parent PRVs of $Q(X)$ such that each logical variable in F is either X or in L, and w_i is a weight.

Suppose $R_i(X_i)$ are the parents of PRV $Q(X)$, where X_i is the set of logical variables in R_i. A relational logistic regression (RLR) for Q with parents $R_i(X_i)$ is defined using a set of WPFs as:

$$P(Q(X)|\Pi) = \text{sigmoid}\left(\sum_{\langle L,F,w_i\rangle} w_i \sum_{L} F_{\Pi,X\to x}\right)$$

where Π represents the assigned values to parents of Q, x represents an assignment of an individual to each logical variable in X, and $F_{\Pi,X\to x}$ is formula F with each logical variable X in it being replaced according to x, and evaluated in Π. So RLR performs an aggregation of the parents of a PRV. The authors show that RLR can model noisy-OR therefore they can encode liftable PLP.

Lifted learning is still an open problem. An approach for performing lifted generative learning was proposed in [141]: while the paper discusses both weight and structure learning, it focuses on Markov Logic Networks and generative learning, so it is not directly applicable to the setting considered in this chapter.

Liftable PLP is very much related to [67, 56, 84] where the contributions of different rules and different rule groundings are combined with noisy-OR combining rules. First-Order Probabilistic Logic (FOPL) [67] and Bayesian Logic Programs (BLP) [56] consider ground atoms as random variables and admit rules with a single atom in the head and only positive literals in the body. The meaning of such rules is that, for each grounding, the head atom random variable directly depends from the body atoms random variables. Thus rules are simply templates that can be used to generate a Bayesian network by a Knowledge-Based Model Construction (KBMC) approach [149]. Ground rules determine the families of the network and the random variables may have non-Boolean domains. The rules are also associated with parameters that define the CPT of the head variable given the body variables. In the case where an atom h appears in the head of more than one ground rule, the Bayesian network contains an extra family where the child is the variable for atom h and there is a parent h' for each rule whose family and CPT is defined by the rule. The extra family containing h and h' encodes a *combining rule*, i.e., a way of combining the contributions of the different rules for the same atom. Both FOPL and BLP allow different combining rules, including a noisy-OR combining rule where the CPT of the extra family encodes a disjunction, so liftable PLP models can be encoded directly in both FOPL and BLP. Differently from Liftable PLP, FOPL and BLP allow multiple layers of rules. Works [67] and [56] also present learning algorithms: the first discusses an EM algorithm for parameter learning and the latter EM and gradient descent parameter learning algorithms together with a structure learning algorithm. The learning problems are similar to the ones considered in this chapter. Additionally, [56] consider the case where non-target atoms may be unobserved in the data. Both articles derive formulas for updating the parameters but, given the generality of the settings considered (non-Boolean domains, multiple layers of rules, different combining rules, incompleteness of the data), the formulas involve quantities to be computed by inference in the Bayesian network, while our formulas depend on the parameters only.

SCOOBY, the structure learning algorithm of [56], is similar to LIFTCOVER in the sense that it performs a greedy search in the space of programs evaluating each hypothesis by performing parameter learning. SCOOBY performs a local search by applying theory revisions to an initial hypothesis, while LIFTCOVER performs a beam search in the space of clauses followed by search in the space of theories by parameter learning.

The First-Order Conditional Influence Language (FOCIL) [84], like FOPL and BLP, considers probabilistic rules compactly encoding probabilistic dependencies. FOCIL is more similar to liftable PLP because it allows only one layer of rules. FOCIL uses different combining rules with respect to liftable PLP: the contributions of different groundings of the same rule with the same random variable in the head are combined by taking the mean and the contributions of different rules are combined either with a weighted mean or with a noisy-OR combining rule. Liftable PLP instead uses noisy-OR for both types of contributions. The authors in [84] also present parameter learning algorithms for optimizing the mean squared errors or the log likelihood using gradient descent or EM both for weighted mean and with noisy-OR. While the derivation of the update formulas for the weights are similar, they differ because we don't use the mean combining function.

11.7 Experiments

LIFTCOVER[1] has been implemented in SWI-Prolog [151] using a porting[2] of YAP-LBFGS[3], a foreign language interface to libLBFGS[4].

LIFTCOVER has been tested on the following 12 real world datasets: the 4 classic benchmarks UW-CSE [64], Mutagenesis [131], Carcinogenesis [130], Mondial [127]; the 4 datasets Bupa, Nba, Pyrimidine, Triazine from `https://relational.fit.cvut.cz/`, and the 4 datasets Financial, Sisya, Sisyb and Yeast from [133][5].

Statistics on all the domains are reported in Table 11.1. In all datasets the mega-examples are defined only by facts, there are no background non-probabilistic rules.

Dataset	P	T	PEx	NEx	F
Financial	9	92658	34	223	10
Bupa	12	2781	145	200	5
Mondial	11	10985	572	616	5
Mutagen.	20	15249	125	126	10
Sisyb	9	354507	3705	9229	10
Sisya	9	358839	10723	6544	10
Pyrimidine	29	2037	20	20	4
Yeast	12	53988	1299	5456	10
Nba	4	1218	15	15	5
Triazine	62	10079	20	20	4
UW-CSE	15	2673	113	20680	5
Carcinogen.	36	24533	182	155	1

Table 11.1: Characteristics of the datasets for the experiments: number of predicates (P), of tuples (T) (i.e., ground atoms), of positive (PEx) and negative (NEx) examples for target predicate(s), of folds (F). The number of tuples includes the target positive examples.

We would like to test the hypothesis that LIFTCOVER allows fast learning

[1]The code of the systems and the datasets are available at `https://bitbucket.org/machinelearningunife/liftcover`.

[2]`https://github.com/friguzzi/lbfgs`

[3]Developed by Bernd Gutmann.

[4]`http://www.chokkan.org/software/liblbfgs/`

[5]`https://dtai.cs.kuleuven.be/ACE/doc/`

without a significant degradation of the quality of the solution with respect to SLIPCOVER: In order to compare the two systems fairly, in all datasets the language bias for SLIPCOVER allows only one atom in the head and only input predicates in the body, so the space of allowed clauses is the same for the two algorithms.

To evaluate the performance, we drew Precision-Recall curves and Receiver Operating Characteristics curves, computing the Area Under the Curve (AUC-PR and AUC-ROC respectively) with the methods reported in [24, 35]. AUC is used to measure the quality of the learned models as classifiers for predicting the truth values of atoms for target predicates, larger areas means better classifiers.

SLIPCOVER was compared with Aleph [129], SLIPCASE [9], SEM-CP-logic [74] LSM [65], ALEPH++ExactL1 [48], LEMUR [33], BUSL [76], MLN-BC, MLN-BT [57] RDN-B [83], SLS [46] and RRR [152, 153] in [11] and [33] on 8 datasets. In almost all datasets SLIPCOVER was among the top 4 systems in terms of AUC-PR, thus showing that it belongs to the state of the art of StarAI. Thus comparing LIFTCOVER with SLIPCOVER will also provide an evaluation of its performance in the general context of StarAI.

LIFTCOVER-EM was run with the following parameters for EM: $restarts = 1$, $max\_iter = 10$, $\epsilon = 10^{-4}$ and $\delta = 10^{-5}$. The default parameters have been used for libLBFGS. The parameters controlling structure learning are: the number *NInt* of mega-examples on which to build the bottom clauses, the number *NA* of bottom clauses to be built for each mega-example, the number *NS* of saturation steps (for building the bottom clauses), the maximum number *NI* of clause search iterations, the size *NB* of the beam, the maximum number *NV* of variables in a rule, the threshold for the rule parameter *WMin* under which the rule is removed and the maximum numbers *NIS* of iterations of structure search of SLIPCOVER. Table 11.2 shows the values we have used. *WMin* was set to 0 in all dataset in order to perform the simplest pruning type, that of rules that don't influence at all the prediction. The other parameters for UW-CSE, Mutagenesis, Carcinogenesis, Mondial have been set as in [33]. For the other datasets they have been set by a random search with the objective of keeping the computation time of both algorithms within a few hundred seconds

All experiments were performed on GNU/Linux machines with an Intel

Dataset	*NB*	*NI*	*NInt*	*NS*	*NA*	*NV*	*WMin*	*NIS*
Financial	100	20	1	1	1	4	0	50
Bupa	100	20	1	1	1	4	0	50
Mondial	1000	10	1	2	6	5	0	10000
Mutagen.	100	10	1	1	1	4	0	500
Sisb	100	20	1	1	1	50	0	40
Sisya	100	20	1	1	1	4	0	40
Pyrimidine	100	20	1	1	1	100	0	50
Yeast	100	20	1	1	1	4	0	50
Nba	100	20	1	1	1	100	0	50
Triazine	100	20	1	1	1	4	0	50
UW-CSE	20	60	1	1	1	4	0	500
Carcinogen.	100	60	1	2	1	3	0	50

Table 11.2: Parameters controlling structure search for LIFTCOVER and SLIPCOVER.

Xeon Haswell E5-2630 v3 (2.40GHz) CPU with 8GB of memory allocated to the job.

Figures 11.2, 11.3 and 11.4 show histograms of the average over the folds of AUC-ROC, AUC-PR and the execution time respectively of LIFTCOVER-EM, LIFTCOVER-LBFGS and SLIPCOVER. Tables 11.3, 11.4 and 11.5 show the same data in a tabular way.

LIFTCOVER-EM beats SLIPCOVER 8 times and ties twice in terms of AUC-ROC and beats SLIPCOVER 5 times and ties twice in terms of AUC-PR, with two cases (Sisya and Sisyb) where it is nearly as good. In terms of execution time, LIFTCOVER-EM is faster than SLIPCOVER in 9 cases and in the other three cases is nearly as fast. In 7 cases the gap is of one or more orders of magnitude (UW-CSE, Carcinognesis, Nba, Triazine, Sisya, Sisyb, Yeast).

LIFTCOVER-LBFGS beats SLIPCOVER 4 times (in Sisya they are very close) and ties twice in terms of AUC-ROC and beats SLIPCOVER 5 times and ties once in terms of AUC-PR, with two cases where it is nearly as good. In terms of execution time, LIFTCOVER-LBFGS beats SLIPCOVER 9 times, with one case where SLIPCOVER is nearly as fast. In 5 cases the gap is of one or more orders of magnitude (UW- CSE, Carcinogenesis, Nba, Sisya,

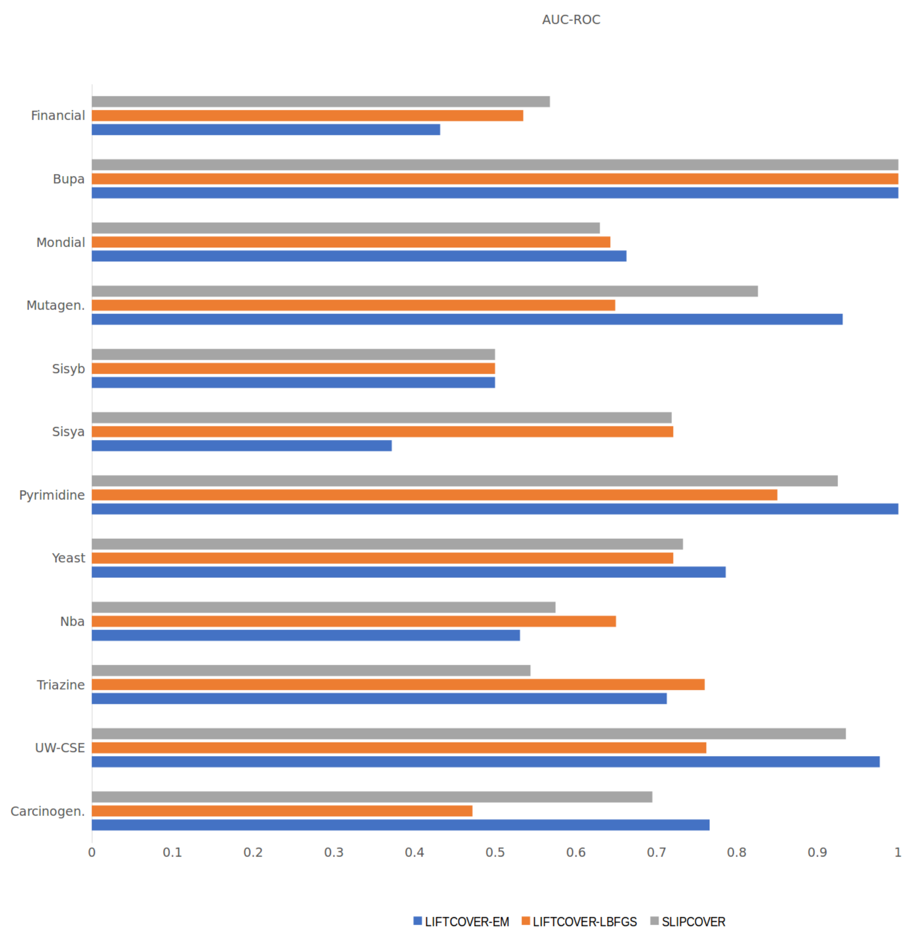


Figure 11.2: Histograms of average AUC-ROC.

AUC-ROC	LIFTCOVER-EM	LIFTCOVER-LBFGS	SLIPCOVER
Financial	0.432	0.535	0.568
Bupa	1.000	1.000	1.000
Mondial	0.663	0.643	0.630
Mutagen.	0.931	0.649	0.826
Sisyb	0.500	0.500	0.500
Sisya	0.372	0.721	0.719
Pyrimidine	1.000	0.850	0.925
Yeast	0.786	0.721	0.733
Nba	0.531	0.650	0.575
Triazine	0.713	0.760	0.544
UW-CSE	0.977	0.762	0.935
Carcinogen.	0.766	0.472	0.695

Table 11.3: Average AUC-ROC.

AUC-PR	LIFTCOVER-EM	LIFTCOVER-LBFGS	SLIPCOVER
Financial	0.126	0.187	0.173
Bupa	1.000	1.000	1.000
Mondial	0.763	0.723	0.776
Mutagen.	0.971	0.725	0.920
Sisyb	0.286	0.286	0.287
Sisya	0.706	0.706	0.708
Pyrimidine	1.000	0.819	0.956
Yeast	0.502	0.448	0.428
Nba	0.550	0.705	0.550
Triazine	0.734	0.760	0.560
UW-CSE	0.220	0.263	0.163
Carcinogen.	0.672	0.561	0.745

Table 11.4: Average AUC-PR.

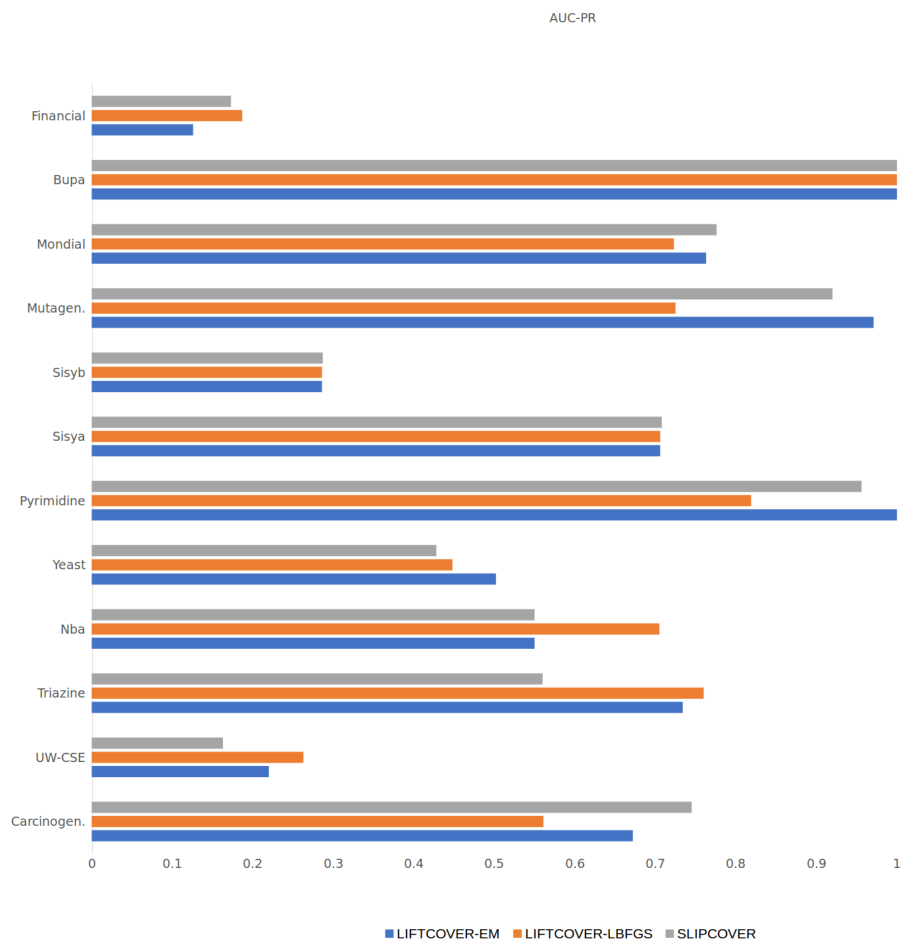


Figure 11.3: Histograms of average AUC-PR.

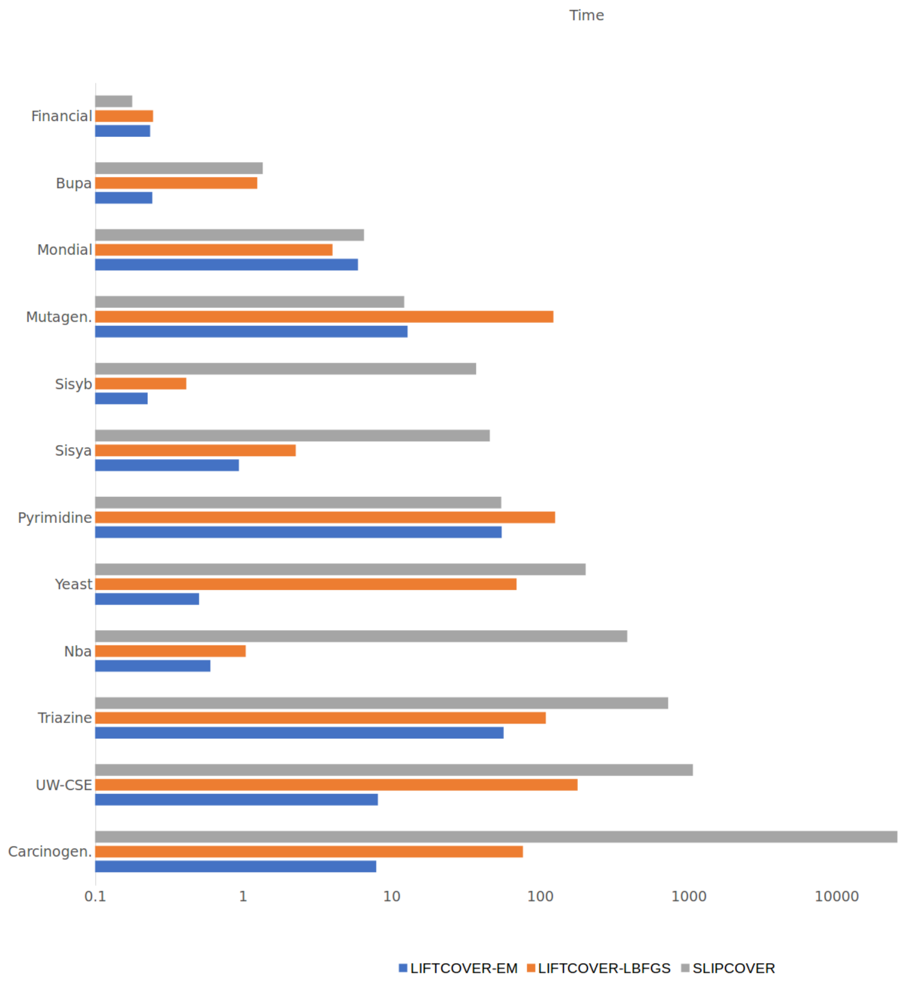


Figure 11.4: Histograms of average time in seconds. The scale of the Y axis is logarithmic.

Time	LIFTCOVER-EM	LIFTCOVER-LBFGS	SLIPCOVER
Financial	0.235	0.246	0.178
Bupa	0.243	1.239	1.349
Mondial	5.911	3.984	6.490
Mutagen.	12.77	122.8	12.11
Sisyb	0.226	0.412	37.00
Sisya	0.932	2.252	45.75
Pyrimidine	54.99	126.1	54.62
Yeast	0.502	69.30	202.4
Nba	0.599	1.036	386.0
Triazine	56.69	109.1	728.2
UW-CSE	8.054	178.6	1069
Carcinogen.	7.850	76.49	25568

Table 11.5: Average time in seconds.

Sisyb). In Mutagenesis and Pyrimidine LIFTCOVER-LBFGS takes about ten times and twice as much as SLIPCOVER respectively. The reason for these differences may be due to the fact that these are small-medium datasets which are relatively easy for the systems (and also for ILP systems in general [106]), so SLIPCOVER is able to achieve good performance even with the allowed small search space.

Overall we see that both lifted algorithms are usually faster, sometimes by a large margin, with respect to SLIPCOVER, especially LIFTCOVER-EM. Moreover, this system often finds better quality solutions, showing that structure search by parameter learning is effective.

Between the two lifted algorithms, the EM version wins 6 times and ties twice with respect to AUC-ROC and wins 5 times and ties 3 times with respect to AUC-PR. In terms of execution times, the EM version wins on all dataset except Mondial, with differences below one order of magnitude except UW-CSE and Yeast. So LIFTCOVER-EM appears to be superior both in terms of solution quality and of computation time. EM seems to be better at escaping local maxima and cheaper than LBFGS, possibly also due to the fact that LBFGS may require a careful tuning of parameters and that it is implemented as a foreign language library.

Therefore, LIFTCOVER represents a valid alternative to SLIPCOVER when learning the definition of a single predicate by using a single layer of rules with a single head.

While the problem of inducing general probabilistic logic program will come to fore soon, learning a restricted language may be a valid alternative in many cases. For example, in [11] and [33] SLIPCOVER was applied to the UW-CSE dataset with a language bias that allowed multiple heads and clauses for non-target predicates. The values of AUCPR obtained there are 0.13 and 0.11 respectively, that are lower than the values obtained by LIFTCOVER and SLIPCOVER shown in Table 11.4. Therefore restricting the language bias in this case actually improved the performance, probably because it has a regularizing effect.

Moreover, by looking at the characteristics of the datasets, there does not seem to exist a clear relationship between the number of predicates/number of tuples and the performance: complex datasets (large number of predicates) may be hard for LIFTCOVER (Carcinogenesis) or easy (Triazine) and large datasets (large number of tuples) may be hard for LIFTCOVER (Financial) or easy (Yeast).

In the next part of the thesis, we are going to present a more expressive language called hierarchical PLP, an extension of liftable PLP, that allows structuring clauses and predicates into layer and in which inference and learning still remaining cheaper than for general PLPs.

Part VI

Hierarchical Probabilistic Logic Programming

Chapter 12

Hierarchical Probabilistic Logic Programming

In this chapter we propose another restriction of the language of Logic Programs with Annotated Disjunctions [144], called *hierarchical PLP*, in which clauses and predicates are hierarchically organized. A hierarchical PLP can be converted into deep neural networks or arithmetic circuits in which inference is cheaper than for general PLP and therefore learning both the parameters and the structure should also be cheaper. The chapter is organized as follows: after a brief introduction in Section 12.1 and the description of *hierarchical PLP* in Section 12.2, we discuss how to perform inference and how to build the arithmetic circuit in Sections 12.3 and 12.4 respectively. Connections with related works are drawn in 12.5.

12.1 Motivation

In chapter 11 we have presented a restriction of LPADs called *liftable PLP* in which predicates in the head of clauses share the same atom. Inference in such language can be performed at the lifted level ensuring fast parameter and structure learning. In order to increase the expressiveness of *liftable PLP*, we present in this chapter an extension of *liftable PLP*, called *hierarchical PLP* in which clauses and predicates are hierarchically organized. The language in this case is truth-functional and is equivalent to the product fuzzy logic. Inference in *hierarchical PLP* is also cheaper as a simple dynamic programming algorithm similar to PRISM [121] is sufficient and knowledge compilation as performed for example by LFI-ProbLog [36] and cplint [108, 116, 117] is not necessary.

Programs in this language can be translated into arithmetic circuits or deep neural networks. From a network we can quickly recompute the probability of the root node if the clause parameters change, as the structure remains the same. Learning *hierarchical PLP* parameters can then be performed by applying techniques of deep learning such as gradient descent and back-propagation, see Section 13.2. An Expectation Maximization algorithm can also be applied, see Section 13.3.

12.2 Hierarchical PLP

Hierarchical PLPs (HPLPs) are extensions of liftable PLPs, see Section 11, in which clauses are organized in many layers rather than 1 as in liftable PLP. Moreover, beside input and output predicates used in liftable PLP, HPLPs introduce the notion of *hidden predicates* allowing an hierarchy among clauses and predicates. Now let us describe in details an HPLP.

Suppose we want to compute the probability of atoms for a predicate r using a probabilistic logic program under the distribution semantics [121]. In particular, we want to compute the probability of a ground atom $r(\boldsymbol{t})$, where $\boldsymbol{t}$ is a vector of term. $r(\boldsymbol{t})$ can be an example in a learning problem and r a *target predicate* (also called *output predicate*). We want to compute the probability of $r(\boldsymbol{t})$, starting from a set of ground atoms (an interpretation) that represent what is true about the example encoded by $r(\boldsymbol{t})$. These ground atoms are built over a set of predicates that we call *input predicates*, in the sense that their definition is given as input and is certain.

We consider a specific form of LPADs defining r in terms of the input predicates. The program contains a set of rules that define r using a number of input and *hidden predicates.* Hidden predicates are disjoint from input and target predicates. Each rule in the program has a single head atom annotated with a probability. Moreover, the program is hierarchically defined so that it can be divided into layers. Each layer contains a set of hidden predicates that are defined in terms of predicates of the layer immediately below or in terms of input predicates. The partition of predicates into layers induces a partition of clauses into layers, with the clauses of layer i defining the predicates of layer i. The clauses in layer 1 define r.

This is an extreme form of program stratification: it is stronger than acyclicity [4] because it is not imposed on the atom dependency graph but on the predicate dependency graph, and is also stronger than stratification [14] that, while applied to the predicate dependency graph, allows clauses with positive literals in the body built on predicates in the same layer. As such, it also prevents inductive definitions and recursion in general, thus making the language not Turing-complete.

A generic clauses C is of the form

$$C = p(\boldsymbol{X}) : \pi :- \phi(\boldsymbol{X}, \boldsymbol{Y}), b_1(\boldsymbol{X}, \boldsymbol{Y}), \ldots, b_m(\boldsymbol{X}, \boldsymbol{Y})$$

where $\phi(\boldsymbol{X}, \boldsymbol{Y})$ is a conjunction of literals for the input predicates using variables $\boldsymbol{X}, \boldsymbol{Y}$. $b_i(\boldsymbol{X}, \boldsymbol{Y})$ for $i = 1, \ldots, m$ is a literal built on a hidden predicate. $\boldsymbol{Y}$ is a possibly empty vector of variables. They are existentially quantified with scope the body. Only literals for input predicates can introduce new variables into the clause and all literals for hidden predicates must use the whole set of variables $\boldsymbol{X}, \boldsymbol{Y}$. Moreover, we require that the predicate of each $b_i(\boldsymbol{X}, \boldsymbol{Y})$ does not appear elsewhere in the body of C or in the body of any other clause. We call *hierarchical PLP* the language that admits only programs of this form.

A generic program defining r is thus:

$$\begin{array}{rcl}
C_1 = r(\boldsymbol{X}) : \pi_1 & :- & \phi_1, b_{11}, \ldots, b_{1m_1} \\
 & \ldots & \\
C_n = r(\boldsymbol{X}) : \pi_n & :- & \phi_n, b_{n1}, \ldots, b_{nm_n} \\
C_{111} = r_{11}(\boldsymbol{X}) : \pi_{111} & :- & \phi_{111}, b_{1111}, \ldots, b_{111m_{111}} \\
 & \ldots & \\
C_{11n_{11}} = r_{11}(\boldsymbol{X}) : \pi_{11n_{11}} & :- & \phi_{11n_{11}}, b_{11n_{11}1}, \ldots, b_{11n_{11}m_{11n_{11}}} \\
 & \ldots & \\
C_{n11} = r_{n1}(\boldsymbol{X}) : \pi_{n11} & :- & \phi_{n11}, b_{n111}, \ldots, b_{n11m_{n11}} \\
 & \ldots & \\
C_{n1n_{n1}} = r_{n1}(\boldsymbol{X}) : \pi_{n1n_{n1}} & :- & \phi_{n1n_{n1}}, b_{n1n_{n1}1}, \ldots, b_{n1n_{n1}m_{n1n_{n1}}} \\
 & \ldots &
\end{array}$$

where we omitted the variables except for rule heads. Such a program can be represented with the tree of Figure 12.1 that contains a node for each literal of hidden predicates and each clause. The tree is divided into levels with levels containing literal nodes alternating with levels with clause nodes.

Each clause node is indicated with $C_{\boldsymbol{p}}$ where $\boldsymbol{p}$ is a sequence of integers encoding the path from the root to the node. For example C_{124} indicates the clause obtained by going to the first child from the left of the root (C_1), then to the second child of C_1 (b_{12}) then to the fourth child of b_{12}. Each literal node is indicated similarly with $b_{\boldsymbol{p}}$ where $\boldsymbol{p}$ is a sequence of integers encoding the path from the root. Therefore, nodes have a subscript that is formed by as many integers as the number of levels from the root. The predicate of literal $b_{\boldsymbol{p}}$ is $r_{\boldsymbol{p}}$ which is different for every value of $\boldsymbol{p}$.

The clauses in lower layers of the tree include a larger number of variables with respect to upper layers: the head $r_{\boldsymbol{p}ij}(\boldsymbol{X}, \boldsymbol{Y})$ of clause $C_{\boldsymbol{p}ijk}$ contains more variables than the head $r_{\boldsymbol{p}}(\boldsymbol{X})$ of clause $C_{\boldsymbol{p}i}$ that calls it.

The constraints imposed on the program require different body literals to have different predicate.

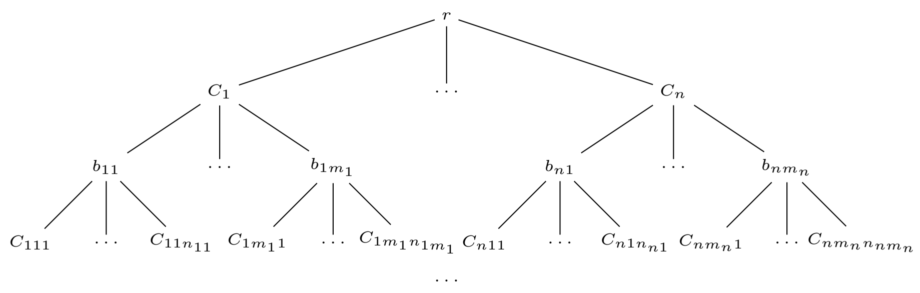


Figure 12.1: Probabilistic program tree.

Example 22. *Let us consider a modified version of the program of Example 21:*

$$\begin{array}{lcl}
C_1 & = & advised\_by(A,B) : 0.3 :- \\
 & & \quad student(A), professor(B), project(C,A), project(C,B), \\
 & & \quad r_{11}(A,B,C). \\
C_2 & = & advised\_by(A,B) : 0.6 :- \\
 & & \quad student(A), professor(B), ta(C,A), taught\_by(C,B). \\
C_{111} & = & r_{11}(A,B,C) : 0.2 :- \\
 & & \quad publication(D,A,C), publication(D,B,C).
\end{array}$$

where $publication(A,B,C)$ means that A is a publication with author B produced in project C. $advised\_by/2$ is the target predicate, $student/1$, $professor/1$, $project/2$, $ta/2$, $taught\_by/2$ and $publication/3$ are input predicates and $r_{11}/3$ is a hidden predicate.

In this case, the probability of $r = advised\_by(harry, ben)$ depends not only on the number of joint courses and projects but also on the number of joint publications from projects. The clause for $r_{11}(A,B,C)$ computes an aggregation over publications of a project and the clause level above aggregates over projects. Such a program can be represented with the tree of Figure 12.2.

Writing programs in hierarchical PLP may be unintuitive for humans because of the need of satisfying the constraints and because the hidden predicates may not have a clear meaning. However, the idea is that the structure of the program is learned by means of a specialized algorithm, with hidden predicates generated by a form of predicate invention. The learning algorithm should search only the space of programs satisfying the constraints, to ensure that the

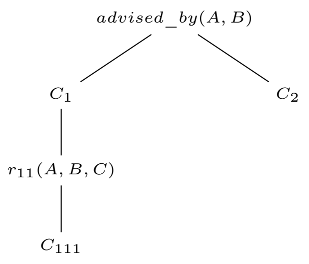


Figure 12.2: Probabilistic program tree for Example 22.

final program is a hierarchical PLP. In Chapter 14, we propose an algorithm for learning the structure.

12.3 Inference

In order to perform inference with such a program, we can generate its grounding. Each ground probabilistic clause is associated with a random variable whose probability of being true is given by the parameter of the clause and that is independent of all the other clause random variables.

Ground atoms are Boolean random variables as well whose probability of being true can be computed by performing inference on the program. Given a ground clause $C_{\boldsymbol{p}i} = a_{\boldsymbol{p}} : \pi_{\boldsymbol{p}i} :- b_{\boldsymbol{p}i1}, \ldots, b_{\boldsymbol{p}im_{\boldsymbol{p}}}$. where $\boldsymbol{p}$ is a path, we can compute the probability that the body is true by multiplying the probability of being true of each individual atom in positive literals and one minus the probability of being true of each individual atom in negative literals. In fact, different literals in the body depend on disjoint sets of clause random variables because of the structure of the program, so the random variables of different literals are independent as well. Therefore the probability of the body of $C_{\boldsymbol{p}i}$ is $P(b_{\boldsymbol{p}i1}, \ldots, b_{\boldsymbol{p}im_{\boldsymbol{p}}}) = \prod_{i=k}^{m_{\boldsymbol{p}}} P(b_{\boldsymbol{p}ik})$ and $P(b_{\boldsymbol{p}ik}) = 1 - P(a_{\boldsymbol{p}ik})$ if $b_{\boldsymbol{p}ik} = \neg a_{\boldsymbol{p}ik}$. Note that if a is a literal for an input predicate, then $P(a) = 1$ if a belongs to the example interpretation and $P(a) = 0$ otherwise.

Now let us determine how to compute the probability of atoms for hidden predicates. Given an atom $a_{\boldsymbol{p}}$ of a literal $b_{\boldsymbol{p}}$, to compute $P(a_{\boldsymbol{p}})$ we need to take into account the contribution of every ground clause for the predicate of $a_{\boldsymbol{p}}$. Suppose these clauses are $\{C_{\boldsymbol{p}1}, \ldots, C_{\boldsymbol{p}o_{\boldsymbol{p}}}\}$. If we have a single clause

$C_{\boldsymbol{p}1} = a_{\boldsymbol{p}} : \pi_{\boldsymbol{p}1} :- b_{\boldsymbol{p}11}, \ldots, b_{\boldsymbol{p}1m_{\boldsymbol{p}1}}.$, then $P(a_{\boldsymbol{p}}) = \pi_{\boldsymbol{p}1} \cdot P(body(C_{\boldsymbol{p}1}))$. If we have two clauses, then we can compute the contribution of each clause as above and then combine them with the formula

$$P(a_{\boldsymbol{p}i}) = 1 - (1 - \pi_{\boldsymbol{p}1} \cdot P(body(C_{\boldsymbol{p}1})) \cdot (1 - \pi_{\boldsymbol{p}2} \cdot P(body(C_{\boldsymbol{p}2})))$$

because the contributions of the two clauses depend on a disjoint set of variables and so they are independent. In fact, the probability of a disjunction of two independent random variables is $P(a \vee b) = 1 - (1 - P(a)) \cdot (1 - P(b)) = P(a) + P(b) - P(a) \cdot P(b)$. We can define the operator $\oplus$ that combines two probabilities as follows $p \oplus q = 1 - (1 - p) \cdot (1 - q)$. This operator is commutative and associative and we can compute sequences of applications as

$$\bigoplus_i p_i = 1 - \prod_i (1 - p_i)$$

The operators $\times$ and $\oplus$ so defined are respectively the t-norm and t-conorm of the product fuzzy logic [43]. They are called *product t-norm* and *probabilistic sum* respectively. For programs of the type above, their interpretation according to the distribution semantics is thus equivalent to that of a fuzzy logic. Therefore the probability of a query can be computed in a truth-functional way, by combining probability/fuzzy values in conjunctions and disjunctions using the t-norm and t-conorm respectively, without considering how the values have been generated: the probability of a conjunction or disjunction of literals depends only on the probability of the two literals. This is in marked contrast with general probabilistic logic programming where knowing the probability of two literals is not enough to compute the probability of their conjunction or disjunction, as they may depend on common random variables.

Therefore, if the probabilistic program of Figure 12.1 is ground, the probability of the example atom can be computed with the arithmetic circuit [22] of Figure 12.3, where nodes are labeled with the operation they perform and edges from $\oplus$ nodes are labeled with a probabilistic parameter that must be multiplied by the child output before combining it with $\oplus$.

The arithmetic circuit can also be interpreted as a deep neural network where nodes can have different activation functions: nodes labeled with $\times$

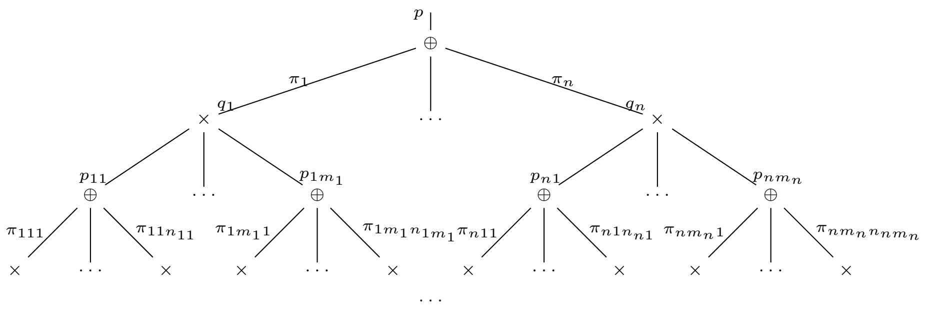


Figure 12.3: Arithmetic circuit/neural net.

compute the product of their inputs, nodes labeled with $\oplus$ compute a weighted probabilistic sum of this form:

$$\pi_1 \cdot q_1 \oplus \ldots \oplus \pi_n \cdot q_n$$

where q_i is the output of the ith child and π_i is the weight of the edge to the ith child.

The output of nodes is also indicated in Figure 12.3, using letter p for $\oplus$ nodes and letter q for $\times$ nodes, subscripted with the path from the root to the node. The network built in this way provides the value p of the probability of the example. Moreover, if we update the parameters π of clauses, we can quickly recompute p in time linear in the number of ground clauses.

When the program is not ground, we can build its grounding obtaining a circuit/neural network of the type of Figure 12.3, where however some of the parameters can be the same for different edges. So in this case the circuit will exhibit parameter sharing.

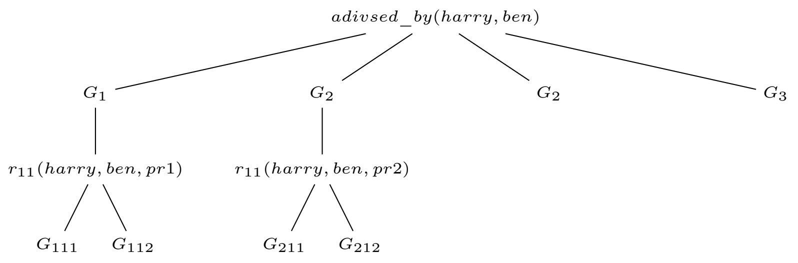
Example 23. *Consider the completed version of Example 22:*

$$
\begin{aligned}
&C_1 = advised\_by(A,B) : 0.3 :-\\
&\quad student(A), professor(B), project(C,A), project(C,B),\\
&\quad r_{11}(A,B,C).\\
&C_2 = advised\_by(A,B) : 0.6 :-\\
&\quad student(A), professor(B), ta(C,A), taughtby(C,B).\\
&C_{111} = r_{11}(A,B,C) : 0.2 :-\\
&\quad publication(P,A,C), publication(P,B,C).\\
&student(harry).\ professor(ben).\\
&project(pr_1, harry).\ project(pr_2, harry).\\
&project(pr_1, ben).\ project(pr_2, ben).\\
&taught\_by(c_1, ben).\ taught\_by(c_2, ben).\\
&ta(c_1, harry).\ ta(c_2, harry).\\
&publication(p_1, harry, pr_1).\ publication(p_2, harry, pr_1).\\
&publication(p_3, harry, pr_2).\ publication(p_4, harry, pr_2).\\
&publication(p_1, ben, pr_1).\ publication(p_2, ben, pr_1).\\
&publication(p_3, ben, pr_2).\ publication(p_4, ben, pr_2).
\end{aligned}
$$

where we suppose that harry and ben have two joint courses c_1 and c_2, two joint projects pr_1 and pr_2, two joint publications p_1 and p_2 from project pr_1 and two joint publications p_3 and p_4 from project pr_2. The resulting ground program is

$$
\begin{aligned}
G_1 &= advisedby(harry, ben) : 0.3 :- \\
&\quad student(harry), professor(ben), project(pr_1, harry), \\
&\quad project(pr_1, ben), r_{11}(harry, ben, pr_1). \\
G_2 &= advisedby(harry, ben) : 0.3 :- \\
&\quad student(harry), professor(ben), project(pr_2, harry), \\
&\quad project(pr_2, ben), r_{11}(harry, ben, pr_2). \\
G_3 &= advisedby(harry, ben) : 0.6 :- \\
&\quad student(harry), professor(ben), ta(c_1, harry), taughtby(c_1, ben). \\
G_4 &= advisedby(harry, ben) : 0.6 :- \\
&\quad student(harry), professor(ben), ta(c_2, harry), taughtby(c_2, ben). \\
G_{111} &= r_{11}(harry, ben, pr_1) : 0.2 :- \\
&\quad publication(p_1, harry, pr_1), publication(p_1, ben, pr_1). \\
G_{112} &= r_{11}(harry, ben, pr_1) : 0.2 :- \\
&\quad publication(p_2, harry, pr_1), publication(p_2, ben, pr_1). \\
G_{211} &= r_{11}(harry, ben, pr_2) : 0.2 :- \\
&\quad publication(p_3, harry, pr_2), publication(p_3, ben, pr_2). \\
G_{212} &= r_{11}(harry, ben, pr_2) : 0.2 :- \\
&\quad publication(p_4, harry, pr_2), publication(p_4, ben, pr_2).
\end{aligned}
$$

The program tree is shown in Figure 12.4. The corresponding arithmetic circuit is shown in Figure 12.5 together with the values computed by the nodes.



$adivsed\_by(harry, ben)$
G_1
G_2
G_2
G_3
$r_{11}(harry, ben, pr1)$
$r_{11}(harry, ben, pr2)$
G_{111}
G_{112}
G_{211}
G_{212}

Figure 12.4: Ground probabilistic program tree for Example 23.

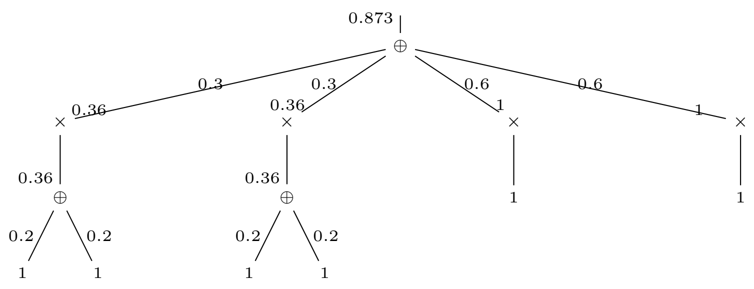


Figure 12.5: Arithmetic circuit/neural net for Example 23.

12.4 Building the Arithmetic Circuit

The network can be built by performing inference using tabling and answer subsumption using PITA(IND,IND) [110]: a program transformation is applied that adds an extra argument to each subgoal of the program and of the query. The extra argument is used to store the probability of answers to the subgoal: when a subgoal returns, the extra argument will be instantiated to the probability of the ground atom that corresponds to the subgoal without the extra argument. In programs of hierarchical PLP, when a subgoal returns the original arguments are guaranteed to be instantiated.

The program transformation also adds suitable literals to the body of clauses that combine the extra arguments of the subgoals: the probabilities of the answers for the subgoals in the body should be multiplied together to give the probability to be assigned to the extra argument of the head atom.

Since a subgoal may unify with the head of multiple groundings of multiple clauses, we need to combine the contributions of these groundings. This is achieved by means of tabling with answer subsumption. Tabling is a Logic Programming technique that reduces computation time and ensures termination for a large class of programs [134]. The idea of tabling is simple: keep a store of the subgoals encountered in a derivation together with answers to these subgoals. If one of the subgoals is encountered again, its answers are retrieved from the store rather than recomputed. Besides saving time, tabling ensures termination for programs without function symbols under the Well-Founded Semantics [140].

Answer subsumption [134] is a tabling feature that, when a new answer for a tabled subgoal is found, combines old answers with the new one. In PITA(IND, IND) the combination operator is probabilistic sum. Computation by PITA(IND, IND) is thus equivalent to the evaluation of the program arithmetic circuit.

Parameter learning can be performed by EM or back-propagation, see Chapter 13. In this case inference has to be performed repeatedly on the same program with different values of the parameters. So we could use an algorithm similar to PITA(IND,IND) to build a representation of the arithmetic circuit, instead of just computing the probability. To do so it is enough to use the extra argument for storing a term representing the circuit instead of the probability and changing the implementation of the predicates for combining the values of the extra arguments in the body and for combining the values from different clause groundings.

The results of inference would thus be a term representing the arithmetic circuit, that can be then used to perform regular inference several times with different parameters values. Implementing EM would adapt the algorithm of [10, 8] for hierarchical PLP.

12.5 Related Work

In [128] the authors discuss an approach for building deep neural networks using a template expressed as a set of weighted rules. Similarly to our approach, the resulting network has nodes representing ground atoms and nodes representing ground rules and the values of ground rule nodes are aggregated to compute the value of atom nodes. Differently from us, the contribution of different ground rules are aggregated in two steps, first the contributions of different groundings of the same rule sharing the same head and then the contributions of different rules, resulting in an extra level of nodes between the ground rule nodes and the atom nodes.

The proposal is parametric in the activation functions of ground rule nodes, extra level nodes and atom nodes. In particular, the authors introduce two families of activation functions that are inspired by Lukasiewicz fuzzy logic. By properly restricting the form of weighted rules and by suitably choosing the activation functions, we can build a neural network whose output is the probability of the example according to the distribution semantics.

Our proposal aims at combining deep learning with probabilistic programming as [137], where the authors propose the Turing-complete probabilistic programming language Edward. Programs in Edward define computational graphs and inference is performed by stochastic graph optimization using TensorFlow as the underlying engine. Hierarchical PLP is not Turing-complete as Edward but ensures fast inference by circuit evaluation. Moreover, it is based on logic so it handles well domains with multiple entities connected by relationships. Similarly to Edward, hierarchical PLP can be compiled to TensorFlow and investigating the advantages of this approach is an interesting direction for future work.

Hierarchical PLP is also related to Probabilistic Soft Logic (PSL) [5] which differs from Markov Logic because atom random variables are defined over the $[0, 1]$ unit interval and logic formulas are interpreted using Lukasiewicz fuzzy logic. We differ from PSL because PSL defines a joint probability distribution over fuzzy variables, while the random variables in hierarchical PLP are still Boolean and the fuzzy values are the probabilities that are combined with the product fuzzy logic. Moreover, the main inference problem in PSL is MAP, i.e., finding a most probable assignment, rather than MARG, i.e., computing the probability of a query, as in hierarchical PLP.

Hierarchical PLP is similar to sum-product networks [101, 145]: the circuits can be seen as sum-product networks where children of sum nodes are not mutually exclusive but independent and each product node has a leaf child that is associated to a hidden random variable. The aim however is different: while sum-product networks represent a distribution over input data, the programs in hierarchical PLP describe only a distribution over the truth values of the query.

Inference in hierarchical PLP is in a way “lifted” [7, 113]: the probability of the ground atoms can be computed knowing only the sizes of the populations of individuals that can instantiate the existentially quantified variables.

Chapter 13

Parameter learning for Hierarchical Probabilistic Logic Programming

In this chapter, we present an algorithm, called Parameter learning for HIerarchical probabilistic Logic programs (PHIL), that learns *hierarchical PLP* parameters from data. PHIL learns *hierarchical PLP* parameters by applying gradient descent or Expectation Maximization. Experiments show that PHIL beats state-of-the-art parameter learning algorithms either in terms of accuracies or in terms of time. The chapter is organized as follows: after presenting the parameter learning problem in Section 13.1, two versions of PHIL, Deep PHIL (DPHIL) and Expectation Maximization PHIL (EMPHIL) and their regularized versions are presented in Sections 13.2 and 13.3 respectively. Experiments comparing PHIL and other state-of-the-art parameter learning such as EMBLEM, [10], and LFI-ProbLog, [36], are presented in Section 13.5.

13.1 Introduction

In Probabilistic logic programs, one of the most interesting task is to estimate the parameters of a given program from data. As explained in Chapter 7.4, inference and learning general PLP is often expensive due to the high cost of inference. In order to speed up inference, we presented in Chapter 12 a new PLP language called *hierarchical PLP* in which inference is cheaper than for general PLP. The purpose of this part of the thesis is to propose algorithms for learning the parameters of such program from data. The parameter learning algorithm can be defined as follows:

Definition 15 (Parameter Learning Problem)**.** *Given an HPLP H with parameters $\Pi = \{\pi_1 \cdots \pi_n\}$, an interpretation I defining input predicates and a training set $E = \{e_1, \ldots, e_M, \mathbf{not}\ e_{M+1}, \ldots, \mathbf{not}\ e_N\}$ where each e_i is a ground atom for the target predicate r, find the values of Π that maximize the log likelihood (LL)*

$$LL = \arg\max_{\Pi} \sum_{i=1}^{M} \log P(e_i) + \sum_{i=M+1}^{N} \log(1 - P(e_i)) \tag{13.1}$$

where $P(e_i)$ is the probability assigned to e_i by $H \cup I$.

Maximizing the LL can be equivalently seen as minimizing the sum of *cross entropy errors* err_i for all the examples

$$err = \sum_{i=1}^{N+M} -y_i \log P(e_i) - (1 - y_i) \log(1 - P(e_i)) \tag{13.2}$$

where e_i is an example with y_i indicating its sign ($y_i = 1$ if the example is positive and $y_i = 0$ otherwise) and p_i indicating the probability that the atom is true.

DPHIL and EMPHIL minimize the cross entropy error or equivalently maximize the log-likelihood of the data. These algorithms (and their regularized versions) are presented in Sections 13.2 and 13.3 respectively.

13.2 Gradient Descent and Back-propagation DPHIL

Gradient descent is an algorithm that iteratively computes the gradients of an error (or a loss) function with respect to a model's parameters and updates the parameters with a fraction of the gradients. To simplify the gradient calculation in models organized hierarchically, a technique called back-propagation is often used to easily apply the chain rule during gradient calculation. Since an HPLP has random variables associated to hidden predicates organized hierarchically, we apply the back-propagation algorithm to compute the gradients and Adam optimizer to update the parameters at each iteration during parameter learning. We called the algorithm Deep Parameter learning for HIerarchical probabilistic Logic programming, DPHIL. The next section explains how DPHIL computes the gradients.

13.2.1 Gradient Calculation

DPHIL computes the gradient of the error *err* (13.2) with respect to each parameter by applying the back-propagation algorithm. We do this by building an AC for each example $e \in E$ and by running a dynamic programming algorithm for computing the gradients. Note that the outputs of ACs of positive examples are labeled 1 and the ones for negative examples are labeled 0. To simplify gradient computation, we transform the ACs (of the form in Figure 12.3) as follows: weight, π_i, labeling arcs from $\bigoplus$ to $\times$ nodes, are set as children leaves of $\times$ nodes and shared weights are considered as individual leaves with many $\times$ parents. Moreover, negative literals are represented by nodes of the form $not(a)$ with the single child a. The AC for Example 23 illustrates in Figure 12.5 is converted into the one shown in Figure 13.1.

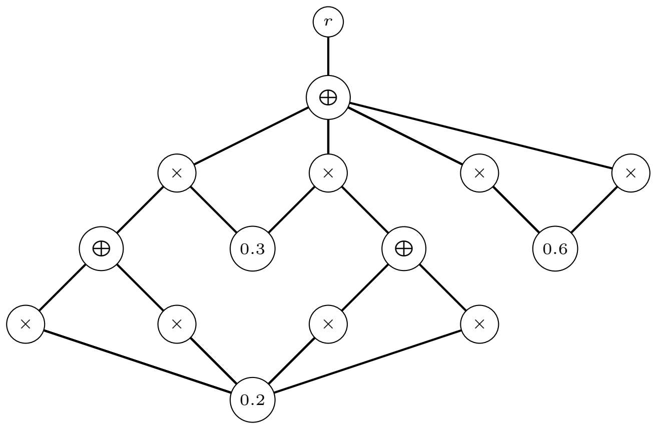


Figure 13.1: Converted arithmetic circuit of Figure 12.5.

.

The standard gradient descent algorithm computes gradients at each iteration using all the examples in the training set. If the training set is large, the algorithm can converge very slowly. To avoid slow convergence, gradients can be computed using a single example, randomly selected in the training set. Even if in this case the algorithm can converge quickly, it is generally hard to reach high training set accuracy. A compromise often used is mini batch stochastic gradient descent (SGD): at each iteration a mini batch of examples is randomly sampled to compute the gradient. This method usually provides fast convergence and high accuracy. DPHIL, shown in Algorithm 13, implements SGD.

After building the ACs and initializing the weights, the gradients and the moments, lines 2–6, DPHIL performs two passes over each AC in the current batch, lines 8–15. In the first, the circuit is evaluated so that each node is assigned a real value representing its probability. This step is bottom-up or forward (line 12) from the leaves to the root. The second step is backward (line 13) or top-down, from the root to the leaves, and computes the derivatives of the loss function with respect to each node. At the end of the backward step, G contains the vector of the derivatives of the error with respect to each parameter. Line 16 updates the weights.

The parameters are repeatedly updated until a maximum number of steps, *MaxIter*, is reached, or until the difference between the LL of the current and the previous iteration drops below a threshold, ϵ, or the difference is below a fraction δ of the current LL. Finally, function UPDATETHEORY (line 18) updates the parameters of the theory. We reparametrized the program using weights between -∞ and + ∞ and expressing the parameters using the sigma function $\pi_i = \sigma(W_i) = \frac{1}{1+e^{-W_i}}$ (13.3). In this way we do not have to impose the constraint that the parameters are in [0,1].

Function FORWARD in Algorithm 14 is a recursive function that takes as input an AC *node* (root node) and evaluates each node from the leaves to the root, assigning value $v(n)$ to each node n. If $node = not(n)$, $p = \text{FORWARD}(n)$ is computed and $1-p$ is assigned to $v(node)$, lines 2–5. If $node = \bigoplus(n_1, \ldots n_m)$, function $v(n_i) = \text{FORWARD}(n_i)$ is recursively called on each child node, and the node value is given by $v(node) = v(n_1) \oplus \ldots \oplus v(n_i)$, lines 7–13. If $node = \times(\pi_i, n_1, \ldots n_m)$, function $v(n_i) = \text{FORWARD}(n_i)$ is recursively called on each child node, and the node value is given by $v(n) = \pi_i \cdot v(n_1) \cdot \ldots \cdot v(n_n)$, lines 14–20.

Procedure BACKWARD takes an evaluated AC *node* and computes the

Algorithm 13 Function DPHIL.

```
function DPHIL($Theory, \epsilon, \delta, MaxIter, \beta_1, \beta_2, \eta, \hat{\epsilon}, Strategy, Min, Max$)
    $ACs \leftarrow$ BUILDACS($Theory$)    ▷ Build the set of ACs
    for $i \leftarrow 1 \rightarrow |Theory|$ do    ▷ Initialize weights, gradient and moments
        $W[i] \leftarrow random(Min, Max)$    ▷ initially $W[i] \in [Min, Max]$.
        $G[i] \leftarrow 0.0$, $M_0[i] \leftarrow 0.0$, $M_1[i] \leftarrow 0.0$
    end for
    $Iter \leftarrow 1$
    repeat
        $LL \leftarrow 0$
        $Batch \leftarrow$ NEXTBATCH($ACs$)    ▷ Select the batch according to the strategy $Strategy$
        for all $circuit \in Batch$ do
            $P \leftarrow$ FORWARD($circuit$)
            BACKWARD($G, -\frac{1}{P}, circuit$)
            $LL \leftarrow LL + \log P$
        end for
        UPDATEWEIGHTSADAM($W, G, M_0, M_1, \beta_1, \beta_2, \eta, \hat{\epsilon}, Iter$)
    until $LL - LL_0 < \epsilon \vee LL - LL_0 < -LL.\delta \vee Iter > MaxIter$
    $FinalTheory \leftarrow$ UPDATETHEORY($Theory, W$)
    return $FinalTheory$
end function
```

derivative of the contribution of the AC to the cost function, $err = -y\log(p) - (1-y)\log(1-p)$ where p is the probability of the atom representing the example.

Let us consider the root node r of the AC for an example e. We want to compute $\frac{\partial err}{\partial v(n)}$ for each node n in the AC. By the chain rule,

$$\frac{\partial err}{\partial v(n)} = \frac{\partial err}{\partial v(r)}\frac{\partial v(r)}{\partial v(n)}$$

Let us first compute $\frac{\partial err}{\partial v(r)}$ where $v(r)$ is the output of the AC

For a positive example, $p = v(r)$, while for a negative example $r = not(n)$, $p = 1 - v(r)$. In this case, the error defined in equation 13.2 becomes $err = -\log(v(r))$. Therefore

$$\frac{\partial err}{\partial v(r)} = -\frac{1}{v(r)} \tag{13.4}$$

Algorithm 14 FUNCTION FORWARD

```
function FORWARD($node$)                                        ▷ node is an AC
    if $node = not(n)$ then
        $v(node) \leftarrow 1 - \text{FORWARD}(n)$
        return $v(node)$
    else
            ▷ Compute the output example by recursively call Forward on its sub AC
        if $node = \bigoplus(n_1, \ldots n_m)$ then                ▷ $\bigoplus$ node
            for all $n_j$ do
                $v(n_j) \leftarrow \text{FORWARD}(n_j)$
            end for
            $v(node) \leftarrow v(n_1) \oplus \ldots \oplus v(n_m)$
            return $v(node)$
        else                                                     ▷ and Node
            if $node = \times(\pi_i, n_1, \ldots n_m)$ then
                for all $n_j$ do
                    $v(n_j) \leftarrow \text{FORWARD}(n_j)$
                end for
                $v(node) \leftarrow \pi_i \cdot v(n_1) \cdot \ldots \cdot v(n_m)$
                return $v(node)$
            end if
        end if
    end if
end function
```

Let us now compute the derivative, $d(n)$, of $v(r)$ with respect to each $v(n)$

$$d(n) = \frac{\partial v(r)}{\partial v(n)}$$

$d(n)$ can be computed by observing that $d(r) = 1$ and, by the chain rule of calculus, for an arbitrary non root node n with pa_n indicating its parents

$$d(n) = \sum_{pa_n} \frac{\partial v(r)}{\partial v(pa_n)} \frac{\partial v(pa_n)}{\partial v(n)} = \sum_{pa_n} d(pa_n) \frac{\partial v(pa_n)}{\partial v(n)}. \tag{13.5}$$

If parent pa_n is a $\times$ node with n' indicating its children $v(pa_n) = \prod_{n'} v(n')$ and if node n is not a leaf (not a parameter node), then

$$\frac{\partial v(pa_n)}{\partial v(n)} = \prod_{n' \neq n} v(n') = \frac{v(pa_n)}{v(n)} \tag{13.6}$$

if $n = \pi_i$ then

$$\frac{\partial v(pa_n)}{\partial \pi_i} = \prod_{n' \neq \pi_i} v(n') = \frac{v(pa_n)}{\pi_i} \tag{13.7}$$

The derivative of pa_n with respect to W_i corresponding to π_i is:

$$\begin{aligned}\frac{\partial v(pa_n)}{\partial W_i} &= \frac{\partial v(pa_n)}{\partial \sigma(W_i)}\frac{\partial \sigma(W_i)}{\partial W_i} = \frac{\partial v(pa_n)}{\partial \sigma(W_i)}\sigma(W_i)(1-\sigma(W_i)) \\ &= \frac{\partial v(pa_n)}{\partial \pi_i}\pi_i(1-\pi_i) = \frac{v(pa_n)}{\pi_i}\pi_i(1-\pi_i) \\ &= v(pa_n)(1-\sigma(W_i)) \end{aligned} \tag{13.8}$$

If parent pa_n is a $\bigoplus$ node with n' indicating its children

$$v(pa_n) = \bigoplus_{n'} v(n') = 1 - \prod_{n'}(1 - v(n'))$$

$$\frac{\partial v(pa_n)}{\partial v(n)} = \prod_{n' \neq n}(1 - v(n')) = \frac{1 - v(pa_n)}{1 - v(n)} \tag{13.9}$$

If the unique parent of n is a $not(n)$ $v(pa_n) = 1 - v(n)$ and

$$\frac{\partial v(pa_n)}{\partial v(n)} = -1 \tag{13.10}$$

Because of the graph construction, $\bigoplus$ and $\times$ nodes can only have one $\times$ and $\bigoplus$ parent respectively. Leaf nodes can have many $\times$ parent nodes. Therefore Equation 13.5 can be written as

$$d(n) = \begin{cases} d(pa_n)\frac{v(pa_n)}{v(n)} & \text{if n is a } \bigoplus node, \\ d(pa_n)\frac{1-v(pa_n)}{1-v(n)} & \text{if n is a } \times \text{ node} \\ \sum_{pa_n} d(pa_n).v(pa_n).(1-\pi_i) & \text{if n}=\sigma(W_i) \\ -d(pa_n) & pa_n = not(n) \end{cases} \tag{13.11}$$

Combining equation 13.4 and 13.11 we have:

$$\frac{\partial err}{\partial v(n)} = -d(n)\frac{1}{v(r)} \tag{13.12}$$

This leads to Procedure BACKWARDGD shown in Algorithm 15 which is a simplified version for the case $v(n) \neq 0$ for all $\bigoplus$ nodes. To compute $d(n)$, BACKWARDGD proceeds by recursively propagating the derivative of the parent node to the children. Initially, the derivative of the error with respect to the root node, $-\frac{1}{v(r)}$, is computed. If the current node is $not(n)$, with derivative $AccGrad$, the derivative of its unique child, n, is $-AccGrad$, lines 2-3. If the current node is a $\bigoplus$ node, with derivative $AccGrad$, the derivative of each child, n, is computed as follows:

$$AccGrad' = AccGrad \cdot \frac{1 - v(node)}{1 - v(n)}$$

and back-propagated, line 5-9. If the current node is a $\times$ node, the derivative of a non leaf child node n is computed as follows:

$$AccGrad'_1 = AccGrad \cdot \frac{v(node)}{v(n)}$$

the one for a leaf child node $n = \pi_i$ is

$$AccGrad'_2 = AccGrad \cdot v(node) \cdot (1 - \sigma(W_i)$$

and back-propagated, lines 11-15. For leaf nodes, i.e π_i nodes, the derivative is accumulated, line 20.

13.2.2 Parameters Update

After the computation of the gradients, weights are updated towards the direction of the optimal value of the error. Standard gradient descent adds a fraction η, called learning rate, of the gradient to the current weights. η is a value between 0 and 1 which is used to control the parameter update. Small η can slow down the algorithm and find local minimum. High η avoids local minima but can swing around global minima. A good compromise updates the

Algorithm 15 PROCEDURE BACKWARDGD

```
procedure BACKWARDGD($G, AccGrad, node$)
    if $node = not(n)$ then
        BACKWARD($G, -AccGrad, n$)
    else
        if $node = \bigoplus(n_1, \dots n_m)$ then                              ▷ $\bigoplus$ node
            for all $n_j$ do
                $AccGrad'_1 \leftarrow AccGrad \cdot \frac{v(node)}{v(n_i)}$
                BACKWARD($G, AccGrad', n_i$)
            end for
        else
            if $node = \times(\pi_i \cdot n_1, \dots n_m)$ then                 ▷ $\times$ node
                for all $n_j$ do                                          ▷ non leaf child
                    $AccGrad' \leftarrow AccGrad \cdot \frac{1-v(node)}{1-v(n_j)}$
                    BACKWARD($G, AccGrad'_1, n_j$)
                end for
                $AccGrad'_2 \leftarrow AccGrad \cdot v(node).(1 - \sigma(W_i))$        ▷ leaf child
                BACKWARD($G, AccGrad'_2, \pi_i$)
            else                                                          ▷ leaf node
                let $node = \pi_i$
                $G[i] \leftarrow G[i] + AccGrad$
            end if
        end if
    end if
end procedure
```

learning rate at each iteration by combining the advantages of both strategies. We apply the update method *Adam*, adaptive moment estimation [62], that uses the first order gradient to compute the exponential moving averages of the gradient and the squared gradient. Hyper-parameters β_1, $\beta_2 \in [0, 1)$ control the exponential decay rates of these moving averages. These quantities are estimations of the *first moment* (the mean M_0) and the *second moment* (the uncentered variance M_1) of the gradient. The weigths are updated with a fraction (the current learning rate) of the combination of these moments, see Procedure UPDATEWEIGHTSADAM in Algorithm 16.

Algorithm 16 PROCEDURE UPDATEWEIGHTSADAM

```
1: procedure UPDATEWEIGHTSADAM($W, G, M_0, M_1, \beta_1, \beta_2, \eta, \hat{\varepsilon}, iter$)
2:     $\eta_{iter} \leftarrow \eta \frac{\sqrt{1-\beta_2^{iter}}}{1-\beta_1^{iter}}$
3:     for $i \leftarrow 1 \rightarrow |W|$ do
4:         $M_0[i] \leftarrow \beta_1 \cdot M_0[i] + (1 - \beta_1) \cdot G[i]$
5:         $M_1[i] \leftarrow \beta_2 \cdot M_1[i] + (1 - \beta_2) \cdot G[i] \cdot G[i]$
6:         $W[i] \leftarrow W[i] - \eta_{iter} \cdot \frac{M_0[i]}{(\sqrt{M_1[i]})+\hat{\epsilon}}$
7:     end for
8: end procedure
```

13.2.3 DPHIL regularization: DPHIL$_1$ and DPHIL$_2$

In deep learning and machine learning in general, a technique called **regularization** is often used to avoid over-fitting. Regularization penalizes the loss function by adding a *regularization* term for favoring smaller parameters. In the literature, there exists two main regularization techniques called L_1 and L_2 regularization that differ from the way they penalize the loss function. While L_1 adds to the loss function the sum of the absolute values of the parameters, L_2 adds the sum of their squares. Given the loss function defined in Equation 13.2, the corresponding regularized loss functions are given by Equations 13.13 and 13.14.

$$err_1 = \sum_{i=1}^{N+M} -y_i \log P(e_i) - (1 - y_i) \log(1 - P(e_i)) + \gamma \sum_{i=1}^{k} |\pi_i| \qquad (13.13)$$

$$err_2 = \sum_{i=1}^{N+M} -y_i \log P(e_i) - (1 - y_i) \log(1 - P(e_i)) + \frac{\gamma}{2} \sum_{i=1}^{k} \pi_i^2 \qquad (13.14)$$

where the regularization hyper-parameter γ determines the strength of the regularization. When γ is zero, the regularized term becomes zero and only the initial loss function is considered. When γ is large, we penalize large values of the parameters and they tend to become small. Note also that we add the regularization term to the initial loss function because we are performing minimization. The main difference between these techniques is that while L_1 favor sparse parameters (many parameters at zero) L_2 favor small values for parameters but not necessarily at 0. Moreover in general, L_1 (resp. L_2) is computationally inefficient (resp. efficient due to having analytical solutions).

Now let us compute the derivative of the regularized error with respect to each node in the AC. The regularized term depends only on the leaves (the parameters π_i) of the AC. So the gradients of the parameters can be calculated by adding the derivative of the regularized term with respect to π_i to the one obtained in equation 13.11. The regularized errors are given by:

$$E_{reg} = \begin{cases} \gamma \sum_{i=1}^{k} \pi_i & \text{for } L_1 \\ \frac{\gamma}{2} \sum_{i=1}^{k} \pi_i^2 & \text{for } L_2 \end{cases} \tag{13.15}$$

where $\pi_i = \sigma(W_i)$. Note that since $0 \leq \pi_i \leq 1$ we can consider π_i rather than $|\pi_i|$ in L_1. So

$$\frac{\partial E_{reg}}{\partial W_i} = \begin{cases} \gamma \frac{\partial \sigma(W_i)}{\partial W_i} = \gamma \cdot \sigma(W_i) \cdot (1 - \sigma(W_i)) = \gamma \cdot \pi_i \cdot (1 - \pi_i) \\ \\ \frac{\gamma}{2} \frac{\partial \sigma(W_i)^2}{\partial W_i} = \gamma \cdot \sigma(W_i) \cdot \sigma(W_i) \cdot (1 - \sigma(W_i)) = \gamma \cdot \pi_i^2 \cdot (1 - \pi_i) \end{cases} \tag{13.16}$$

So equation 13.11 becomes

$$d(n) = \begin{cases} d(pa_n) \frac{v(pa_n)}{v(n)} & \text{if n is a } \bigoplus node, \\ d(pa_n) \frac{1 - v(pa_n)}{1 - v(n)} & \text{if n is a } \times \text{ node} \\ \sum_{pa_n} d(pa_n).v(pa_n).(1 - \pi_i) + \frac{\partial E_{reg}}{\partial W_i} & \text{if n=}\sigma(W_i) \\ -d(pa_n) & pa_n = not(n) \end{cases} \tag{13.17}$$

In order to implement the regularized version of DPHIL (DPHIL$_1$ and DPHIL$_2$), the forward and the backward passes described in algorithms 14 and 15 remain unchanged. The unique change occurs while updating the parameters in the algorithm 16. *UpdateWeightsAdam* line 6 becomes

$$W[i] \leftarrow W[i] - \eta_{iter} * \frac{M_0[i]}{(\sqrt{M_1[i]}) + \hat{\epsilon}} + \frac{\partial E_{reg}}{\partial W_i} \tag{13.18}$$

13.3 Expectation Maximization: EMPHIL

We propose another algorithm, **E**xpectation **M**aximization **P**arameter learning for **HI**erarchical probabilistic **L**ogic programs (EMPHIL) see [89], that learns the parameters of HPLP by applying Expectation Maximization. The algorithm maximizes the log-likelihood LL defined in Equation 13.1 by alternating between an Expectation (E) and a Maximization (M) step. The E-step computes the expected values of the incomplete data given the complete data and the current parameters and the M-step determines the new values of the parameters that maximize the likelihood. Each iteration is guaranteed to increase the log-likelihood. Given an HPLP $H = \{C_i | i = 1, \cdots, n\}$ (each C_i annotated with the parameter π_i) and a training set of positive and negative examples $E = \{e_1, \ldots, e_M, \textbf{not } e_{M+1}, \ldots, \textbf{not } e_N\}$, EMPHIL proceeds as follows:

For a single example e, the E-step computes $\mathbf{E}[c_{i0}|e]$ and $\mathbf{E}[c_{i1}|e]$ for all rules C_i. c_{ix} is the number of times a variable X_{ij} takes value x for $x \in \{0, 1\}$ and for all $j \in g(i)$ i.e $\mathbf{E}[c_{ix}|e] = \sum_{j \in g(i)} P(X_{ij} = x|e)$ where $g(i) = \{j | \theta_j \text{ is a substitution grounding } C_i\}$. These values are aggregated over all examples obtaining

$$N_0 = \mathbf{E}[c_{i0}] = \sum_{e \in E} \sum_{j \in g(i)} P(X_{ij} = 0|e) \tag{13.19}$$

$$N_1 = \mathbf{E}[c_{i1}] = \sum_{e \in E} \sum_{j \in g(i)} P(X_{ij} = 1|e) \tag{13.20}$$

Then the M-step computes π_i by maximum likelihood, i.e. $\pi_i = \frac{N_1}{N_0 + N_1}$. Note that for a single substitution θ_j of clause C_i we have $P(X_{ij} = 0|e) + P(X_{ij} = 1|e) = 1$. So $E[c_{i0}] + E[c_{i1}] = \sum_{e \in E} |g(i)|$. So the M-step computes

$$\pi_i = \frac{N_1}{\sum_{e \in E} |g(i)|} \tag{13.21}$$

Therefore to perform EMPHIL, we have to compute $P(X_{ij} = 1|e)$ for each example e. We do it using two passes over the AC, one bottom-up and one top-down. In order to illustrate the passes, we construct a graphical model (for example a Bayesian Network) associated with the AC and then apply the *belief propagation* (BP) algorithm [93].

A Bayesian Network (BN) can be obtained from the AC by replacing each node with a random variable. The variables associated with an $\bigoplus$ node have a conditional probabilistic table (CPT) that encodes an OR deterministic function, while variables associated with an $\times$ node have a CPT encoding an AND. Variables associated with a $\neg$ node have a CPT encoding the NOT function. Leaf nodes associated with the same parameter are split into as many nodes X_{ij} as the groundings of the rule C_i, each associated with a CPT such that $P(X_{ij} = 1) = \pi_i$. We convert the BN into a Factor Graph (FG) using the standard translation because BP can be expressed in a simpler way for FGs. The FG corresponding to the AC of Figure 13.1 is shown in Figure 13.2.

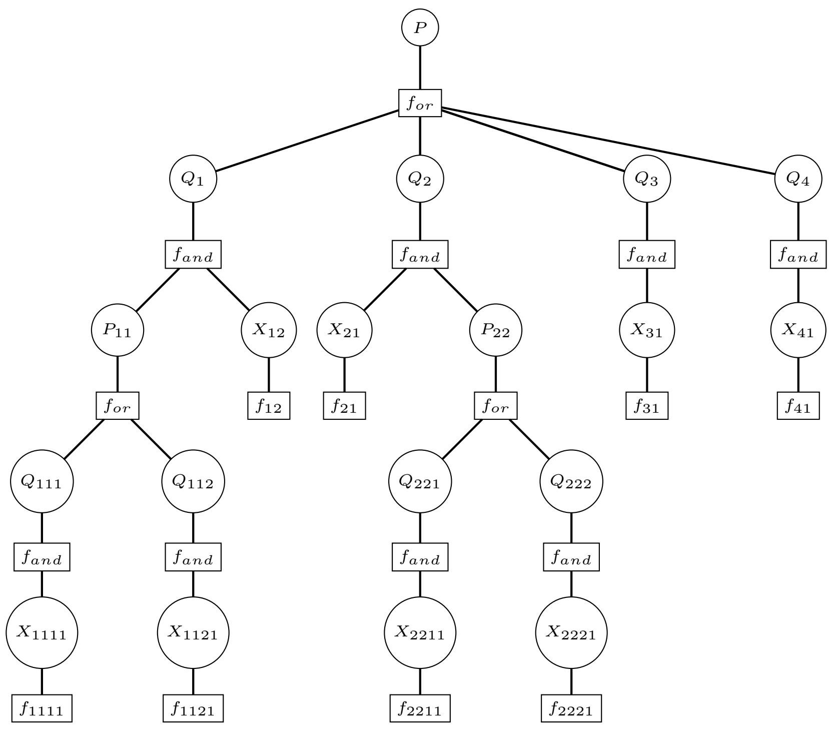


Figure 13.2: Factor graph.

.

13.3.1 Message Exchanges

After constructing the FG, $P(X_{ij} = 0|e)$ and $P(X_{ij} = 1|e)$ are computed by exchanging messages among nodes and factors until convergence. In the case of

FG obtained from an AC, the graph is a tree and it is sufficient to propagate the message first bottom-up and then top-down. The message from a variable N to a factor f is defined as follows: see [93]

$$\mu_{N\to f}(n) = \prod_{h\in nb(N)\backslash f} \mu_{h\to N}(n) \tag{13.22}$$

where $nb(X)$ is the set of neighbors of X (the set of factors X appears in). The message from a factor f to a variable N is:

$$\mu_{f\to N}(n) = \sum_{\neg N}(f(n,\mathbf{s}) \prod_{Y\in nb(f)\backslash N} \mu_{Y\to f}(y)) \tag{13.23}$$

where $nb(f)$ is the set of arguments of f and $\neg N$ means all values for the arguments of f with N fixed at n. After convergence, the belief of each variable N is defined as follows:

$$b(n) = \prod_{f\in nb(N)} \mu_{f\to N}(n) \tag{13.24}$$

that is the product of all incoming messages to the variable. By normalizing $b(n)$ we obtain $P(N = n|e)$. Evidence is taken into account by setting the cells of the factors that are incompatible with evidence to 0. We want to develop an algorithm for computing $b(n)$ over the AC. So we want the AC nodes to send messages. We call c_N the normalized message, $\mu_{f\to N}(N = 1)$, in the bottom-up pass and t_N the normalized message, $\mu_{f\to N}(N = 1)$, in the top-down pass. Let us now compute the messages in the forward pass. Different cases can occur: the leaf, the inner and the root node.

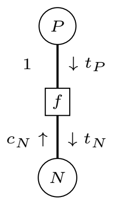
For a leaf node X, we have the factor graph in Figure 13.3b. From Table 13.1d, the message from f_x to X is given by:

$$\mu_{f_x\to X} = [\pi(x), 1-\pi(x)] = [v(x), 1-v(x)] \tag{13.25}$$

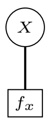
Note that the message is equal to the value of the node. Moreover, because of the construction of HPLP, for any variable node N

$$\mu_{f\to P}(p) = \mu_{P\to f}(p) \tag{13.26}$$

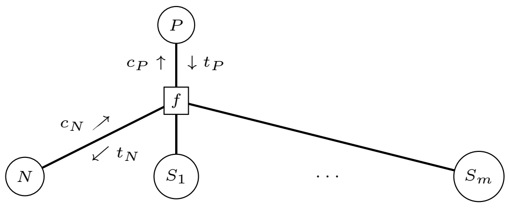
Figure 13.3: Examples of factor graph



(a) Factor graph of not node.



(b) Factor graph for a leaf node.



(c) Factor graph for inner or root node.

Table 13.1: CPTs of factors

(a) P is an *or* node

p	$n = 1$	$n = 0$, $\mathbf{S} = \mathbf{0}$	$n = 1$, $\neg$ $(\mathbf{S} = \mathbf{0})$
0	0	1	0
1	1	0	1

(b) P is an *and* node

p	$n = 0$	$n = 1$, $\mathbf{S} = \mathbf{1}$	$n = 1$, $\neg$ $(\mathbf{S} = \mathbf{1})$
0	1	0	1
1	0	1	0

(c) P is a not node

p	$n = 0$	$n = 1$
0	0	1
1	1	0

(d) Leaf node $f_x = \pi(x)$

x	f_x
0	1-$\pi(x)$
1	$\pi(x)$

where P is the parent of N.

Let us consider a node P with children N,$S_1 \dots S_m$ as shown in Figure 13.3c. We define $\mathbf{S} = S_1 \dots S_m$ and $\mathbf{s} = s_1 \dots s_m$. We prove by induction that $c_P = v(P)$. For leaf nodes it was proved above. Suppose that $c_C = v(C)$ for all children $N, S_1, ...S_m$:

If P is an $\times$ node, the CPT of P given its children is described in Table 13.1b and $\mu_{C \to f}(c) = v(c)$ for all children C. According to equation 13.23 we have:

$$\begin{aligned}
\mu_{f \to P}(1) &= \sum_{\neg P} f(p, n, \mathbf{s}) \prod_{Y \in nb(f) \setminus P} \mu_{Y \to f}(y) \\
&= \sum_{n,\mathbf{s}} (f(p, n, \mathbf{s}) \prod_{Y \in \{N, \mathbf{S}\}} \mu_{Y \to f}(y) \\
&= \mu_{N \to f}(1) \cdot \prod_{S_k} \mu_{S_k \to f}(1) \\
&= v(N) \cdot \prod_{s_k} v(S_k) = v(P)
\end{aligned} \tag{13.27}$$

In the same way, from Equation 13.27 we have:

$$\begin{aligned}
\mu_{f \to P}(0) &= \sum_{n,\mathbf{s}} (f(p, n, \mathbf{s}) \prod_{Y \in \{N, \mathbf{S}\}} \mu_{Y \to f}(y)) \\
&= \mu_{N \to f}(0) \cdot \prod_{S_k} \mu_{S_k \to f}(0) \\
&= 1 - (\mu_{N \to f}(1) \cdot \prod_{S_k} \mu_{S_k \to f}(1)) \\
&= 1 - (v(N) \cdot \prod_{s_k} v(S_k)) = 1 - v(P)
\end{aligned}$$

So $c_P = v(P)$

If P is an $\bigoplus$ node, the CPT of P given its children is described in Table

13.1a. From Equation 13.27 we have:

$$\begin{aligned}
\mu_{f\to P}(1) &= \sum_{n,\mathbf{s}} f(p,n,\mathbf{s}) \prod_{Y\in\{N,\mathbf{S}\}} \mu_{Y\to f}(y) \\
&= 1-\mu_{N\to\cdot}f(0)\cdot\prod_{S_k}\mu_{S_k\to f}(0) = 1-v(N)\cdot\prod_{S_k}v(S_k) \\
&= 1-(1-v(N))\cdot\prod_{S_k}(1-v(S_k)) = v(P)
\end{aligned}$$

In the same way we have:

$$\begin{aligned}
\mu_{f\to P}(0) &= \mu_{N\to f}(0).\prod_{S_k}\mu_{S_k\to f}(0) && (13.28)\\
&= v(N=0)\cdot\prod_{S_k}\mu_{S_k\to f}(0) \\
&= 1-[v(N=1)\cdot\prod_{S_k}\mu_{S_k\to f}(1)] && (13.29)\\
&= 1-[1-(1-v(N)\cdot\prod_{S_k}v(S_k)] = 1-v(P) && (13.30)
\end{aligned}$$

If P is a $\neg$ node with the single child N, its CPT is shown in Table 13.1c and we have:

$$\begin{aligned}
\mu_{f\to P}(1) &= \sum_{n} f(p,n) \prod_{Y\in\{N\}} \mu_{Y\to f}(y) \\
&= \mu_{N\to f}(0) = 1-v(N)
\end{aligned}$$

and

$$\mu_{f\to P}(0) = \mu_{N\to f}(1) = v(N)$$

Overall, exchanging message in the forward pass means evaluating the value of each node in the AC: Messages in the forward pass are computed by applying Algorithm 14.

Now let us compute the messages in the backward pass. Considering the factor graph in Figure 13.3c, we consider the message $t_P = \mu_{P\to f}(1)$ as known and we want to compute the message $t_N = \mu_{f\to N}(1)$.

If P is an *inner* $\bigoplus$ node (with children $N, S_1, ...S_m$), its CPT is shown in table 13.1a. Let us compute the messages $\mu_{f\to N}(1)$ and $\mu_{f\to N}(0)$:

$$\begin{aligned}\mu_{f\to N}(1) &= \sum_{\neg N}(f(p,n,\mathbf{s})\prod_{Y\in nb(f)\setminus N}\mu_{Y\to f}(y)) \\ &= [\sum_{p,\mathbf{s}}(f(p,n,\mathbf{s})\prod_{S}v(s)]\cdot[\mu_{P\to f}(1)] \\ &= \mu_{P\to f}(1) = t_P \qquad (13.31)\end{aligned}$$

In the same way

$$\begin{aligned}\mu_{f\to N}(0) &= \sum_{p,\mathbf{s}} f(p,n,\mathbf{s})\prod_{S}v(s)[\mu_{P\to f}(p)] \qquad (13.32) \\ &= [1-\prod_{S}(1-v(S))]\cdot[\mu_{P\to f}(1)] + \prod_{S}(1-v(S))[\mu_{P\to f}(0)] \\ &= v(P)\ominus v(N)\cdot t_P + (1-v(P)\ominus v(N))\cdot(1-t_P)\} \qquad (13.33)\end{aligned}$$

where the operator $\ominus$ is defined as follows:

$$v(p)\ominus v(n) = 1-\prod_{\mathbf{s}}(1-v(s)) = 1-\frac{1-v(p)}{1-v(n)} \qquad (13.34)$$

So we have

$$t_N = \frac{t_P}{t_P + v(P)\ominus v(N)\cdot t_P + (1-v(P)\ominus v(n))\cdot(1-t_P)} \qquad (13.35)$$

If P is a $\times$ node, its CPT is shown in Table 13.1b and we have:

$$\begin{aligned}\mu_{f\to N}(1) &= \sum_{\neg N}(f(p,n,\mathbf{s})\prod_{Y\in nb(f)\setminus N}\mu_{Y\to f}(y)) \\ &= \mu_{P\to f}(P=1)\cdot\prod_{S}\mu_{S\to f}(1) + \mu_{P\to f}(0)\cdot(1-\prod_{S}\mu_{S\to f}(1)) \\ &= t_P\cdot\prod_{S}\ v(S) + (1-t_P)\cdot(1-\prod_{S}v(S)) \\ &= t_P\cdot\frac{v(P)}{v(N)} + (1-t_P)\cdot(1-\frac{v(P)}{v(N)})\end{aligned}$$

In the same way,

$$\mu_{f\to N}(0) = \mu_{P\to f}(0) \cdot \sum_{\mathbf{S}} (f(p, n, \mathbf{s}) \prod_{\mathbf{S}} \mu_{S\to f}(s)) = 1 - t_P$$

So we have

$$t_N = \frac{t_P.\frac{v(P)}{v(N)} + (1 - t_P) \cdot (1 - \frac{v(P)}{v(N)})}{t_P.\frac{v(P)}{v(N)} + (1 - t_P).(1 - \frac{v(P)}{v(N)}) + (1 - t_P)} \tag{13.36}$$

If P is a $\neg$ node, its CPT is shown in Table 13.1c and we have:

$$\mu_{f\to N}(1) = \sum_{p} f(p, n) \prod_{Y\in\{P\}} \mu_{Y\to f}(y) = \mu_{P\to f}(0) = 1 - t_P$$

Equivalently

$$\mu_{f\to N}(0) = \sum_{p} f(p, n) \prod_{Y\in\{P\}} \mu_{Y\to f}(y) = \mu_{P\to f}(1) = t_P$$

And then

$$t_N = \frac{1 - t_P}{1 - t_P + t_P} = 1 - t_P \tag{13.37}$$

To take into account evidence, we consider $\mu_{P\to f} = [1, 0]$ as the initial messages in the backward pass (where P is the root) and use Equation 13.35 for $\bigoplus$ node. Overall, in the backward pass we have:

$$t_N = \begin{cases} \frac{t_P}{t_P + v(P)\ominus v(N)\cdot t_P + (1 - v(P)\ominus v(N))\cdot(1 - t_p)} & \text{if } P \text{ is a } \oplus \text{ node} \\ \frac{t_P\cdot\frac{v(P)}{v(N)} + (1 - t_P)\cdot(1 - \frac{v(P)}{v(N)})}{t_P\cdot\frac{v(P)}{v(N)} + (1 - t_P)\cdot(1 - \frac{v(P)}{v(N)}) + (1 - t_P)} & \text{if } P \text{ is a } \times \text{ node} \\ 1 - t_P & \text{if } P \text{ is a } \neg \text{ node} \end{cases} \tag{13.38}$$

Since the belief propagation algorithm (for ACs) converges after two passes, we can compute the unnormalized belief of each parameter during the backward pass by multiplying t_N by $v(N)$ (that is all incoming messages). Algorithm 17 performs the backward pass of belief propagation algorithm and computes the normalized belief of each parameter, i.e t_N. It also computes the expectations

N_0 and N_1 for each parameter, lines 17–19.

Algorithm 17 PROCEDURE BACKWARD IN EMPHIL

```
 1: procedure BACKWARDEM($t_p, node, N_0, N_1$)
 2:     if $node = not(n)$ then
 3:         BACKWARD($1 - t_p, n, B, Count$)
 4:     else
 5:         if $node = \bigoplus(n_1, \dots n_m)$ then                              ▷ $\bigoplus$ node
 6:             for all child $n_i$ do
 7:                 $t_{n_i} \leftarrow \frac{t_p}{t_p + v(node) \ominus v(n_i) \cdot t_p + (1 - v(node) \ominus v(n_i)) \cdot (1 - t_p)}$
 8:                 BACKWARDEM($t_{n_i}, n_i, B, Count$)
 9:             end for
10:         else
11:             if $node = \times(n_1, \dots n_m)$ then                             ▷ $\times$ node
12:                 for all child $n_i$ do
13:                     $t_{n_i} \leftarrow \frac{t_p \cdot \frac{v(node)}{v(n_i)} + (1 - t_p) \cdot (1 - \frac{v(node)}{v(n_i)})}{t_p \cdot \frac{v(node)}{v(n_i)} + (1 - t_p) \cdot (1 - \frac{v(node)}{v(n_i)}) + (1 - t_p)}$
14:                     BACKWARDEM($t_{n_i}, n_i, B, Count$)
15:                 end for
16:             else                                                            ▷ leaf node $\pi_i$
17:                 let $E = \frac{\pi_i t_p}{(\pi_i t_p + (1 - \pi_i)(1 - t_p)}$
18:                 $N_1[i] \leftarrow N_1[i] + E$
19:                 $N_0[i] \leftarrow N_0[i] + 1 - E$
20:             end if
21:         end if
22:     end if
23: end procedure
```

EMPHIL is then presented in Algorithm 18. After building the ACs (sharing parameters) for positive and negative examples and initializing the parameters, the expectations and the counters, lines 2–5, EMPHIL proceeds by alternating between expectation step 8–13 and maximization step 13–24. The algorithm stops when the difference between the current value of the LL and the previous one is below a given threshold or when such a difference relative to the absolute value of the current one is below a given threshold. The theory is then updated and returned (lines 26–27).

13.3.2 EMPHIL regularization: EMPHIL$_1$, EMPHIL$_2$ and EMPHIL$_3$

In this section, we propose three regularized versions of EMPHIL. As described in [71], EM can be regularized for two reasons: first, for highlighting the strong relationship existing between the incomplete and the missing data, assuming in the standard EM algorithm, and second for favoring smaller parameters. We regularize EMPHIL mainly for the latter reason. As in gradient descent regularization, we define the following regularized objective functions for L_1 and L_2 respectively in the M step.

$$J(\theta) = \begin{cases} N_1 \log\theta + N_0 \log(1-\theta) - \gamma\theta & \text{for } L_1 \\ N_1 \log\theta + N_0 \log(1-\theta) - \frac{\gamma}{2}\theta^2 & \text{for } L_2 \end{cases} \tag{13.39}$$

where $\theta = \pi_i$, N_0 and N_1 are the expectations computed in the E-step (see Equations 13.19 and 13.20). The M-step aims at computing the value of θ that maximizes $J(\theta)$. This is done by solving the equation $\frac{\partial J(\theta)}{\partial\theta} = 0$. The following theorems give the optimal value of θ in each case (see appendix A for the proofs).

Theorem 1. *The L_1 regularized objective function:*

$$J_1(\theta) = N_1 \log\theta + N_0 \log(1-\theta) - \gamma\theta \tag{13.40}$$

is maximum at

$$\theta_1 = \frac{4N_1}{2(\gamma + N_0 + N_1 + \sqrt{(N_0+N_1)^2 + \gamma^2 + 2\gamma(N_0 - N_1)})}$$

Theorem 2. *The L_2 regularized objective function:*

$$J_2(\theta) = N_1 \log\theta + N_0 \log(1-\theta) - \frac{\gamma}{2}\theta^2 \tag{13.41}$$

is maximum at

$$\theta_2 = \frac{2\sqrt{\frac{3N_0+3N_1+\gamma}{\gamma}}\cos\left(\frac{\arccos\left(\frac{\sqrt{\frac{\gamma}{3N_0+3N_1+\gamma}}\left(\frac{9N_0}{2}-9N_1+\gamma\right)}{3N_0+3N_1+\gamma}\right)}{3}-\frac{2\pi}{3}\right)}{3}+\frac{1}{3}$$

We consider another regularization method for EMPHIL (called EMPHIL$_3$) which is based on a Bayesian update of the parameters assuming a prior that takes the form of a Dirichlet with parameters $[a, b]$. In M-step instead of computing $\pi_i = \frac{N_1}{N_0+N_1}$, EMPHIL$_3$ computes

$$\pi_i = \frac{N_1 + a}{N_0 + N_1 + a + b} \tag{13.42}$$

as described in [12]. a and b are hyper-parameters. We choose $a = 0$ and b as a fraction of the training set size, see Section 13.5, since we want small parameters.

So algorithms EMPHIL (the standard EM), EMPHIL$_1$, EMPHIL$_2$ and EMPHIL$_3$ differ from the way they update the parameters in the M-step, Algorithm 18 lines 15-22.

13.4 Related Work

PHIL is related to LFI-ProbLog, [36] which is an algorithm for learning probabilistic Logic Programs parameters using Expectation Maximization, see Section 10.2.5. PHIL and LFI-ProbLog differ from the Probabilistic language they use. In order to perform inference, while PHIL converts a program into a set of ACs and evaluates the ACs bottom-up, LFI-ProbLog converts the program into a weighted Boolean formula an performs Weighted Model Counting (WMC) [119]. PHIL performs parameter learning by applying gradient descent (DPHIL) or EM (EMPHIL) on ACs and LFI-ProbLog performs parameter learning using EM on top of the WMC.

PHIL is also related to EMBLEM (Expectation maximization over binary decision diagrams for probabilistic logic programs) [10], an algorithm for learning general PLP parameters applying EM over Binary decision diagram [1], see

Section 10.2.4. EMPHIL, LFI-ProbLog and EMBLEM are strongly related as they all apply the EM algorithm.

13.5 Experiments

In this section, we present experiments comparing PHIL (DPHIL, EMPHIL and their regularized versions) with EMBLEM [10] and LFI-ProbLog, [36]. PHIL[1] has been implemented in SWI-Prolog [150] and C languages. It can be installed using *pack_install(phil)* on SWI-Prolog. We perform experiments [2] on GNU/Linux machines with an Intel Xeon E5-2697 core 2 Duo (2,335 MHz) comparing our algorithms with the state-of-the-art parameter algorithms LFI-ProbLog, [36], EMBLEM, [10]. While Section 13.5.1 describes the various datasets used for performing the experiments, Section 13.5.2 presents the methodology and the experiments.

13.5.1 Datasets

We experiment our algorithms on four well known datasets:

The Mutagenesis dataset, [131] contains information about a number of aromatic/heteroaromatic nitro drugs, and their chemical structures in terms of atoms, bonds and other molecular substructures. For example, the dataset contains atoms of the form $bond(compound, atom1, atom2, bondtype)$ which states that a bond of type *bondtype* can be found in the compound between the atoms $atom1$ and $atom2$. The goal is to predict the mutagenicity of drugs which is important for understanding carcinogenesis. The subset of the compounds having positive levels of log mutagenicity are labeled *active* (the target predicate) and the remaining ones are *inactive*.

The Carcinogenesis dataset [130] is similar to the Mutagenesis dataset and the objective is to predict the carcinogenicity of molecules.

The Mondial dataset [73] contains data from multiple geographical web data sources. The goal is to predict the religion of a country as Christian (target predicate $christian\_religion(A)$).

[1]The code and the datasets are available at `https://github.com/ArnaudFadja/phil`.

[2]Experiments are available at `https://bitbucket.org/ArnaudFadja/hierarchicalplp_experiments/src/master/`.

The UWCSE dataset [6] contains information about the Computer Science department of the University of Washington. The goal is to predict the target predicate $advised\_by(A, B)$ expressing that a student A is advised by a professor B.

13.5.2 Methodology

We manually built the HPLPs for the datasets UWCSE and generated those for Mutagenesis, Carcinogenesis and Mondial using algorithm SLEAHP described in Chapter 14. We use the following hyper-parameters: As stop conditions, we use $\epsilon = 10^{-4}$, $\delta = 10^{-5}$, $MaxIter = 1000$ for PHIL and EMBLEM and $MIN\_IMPROV = 10^{-4}$, $MaxIter = 1000$ for LFI-ProbLog. We use the Adam hyper-parameters $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\eta = 0.9$, $\hat{\epsilon} = 10^{-8}$ and we apply *batch gradient descent* (all ACs are used for computing gradients at each iteration) on every dataset except for UWCSE where we use *stochastic gradient descent* with batch size $BatchSize = 100$. In the regularized version of PHIL, clauses with parameters less than $MinProb = 10^{-5}$ are removed. We experiment with three versions of EMPHIL$_3$ (EMPHIL$_{3_1}$, EMPHIL$_{3_2}$, EMPHIL$_{3_3}$) which use $a = 0$ and differ from the fraction of the examples n they use at the M-step. They use $b = \frac{n}{10}$, $b = \frac{n}{5}$, $b = \frac{n}{4}$ respectively. The parameters in the gradient descent method are initialized between [-0.5, 0.5] and the ones in the EM between [0,1].

In order to test the performance of the algorithms, we apply the cross-validation method: each dataset is partitioned into NF folds of which one is used for testing and the remaining for training in turn. The characteristics of the datasets in terms of number of clauses NC, layers NL, folds NF and the average number of Arithmetic circuits NAC for each fold of each dataset are summarized in Table 13.2.

Table 13.2: Hyper-parameters

	Mutagenesis	Carcinogenesis	Mondial	UWCSE
NC	58	38	10	17
NL	9	7	6	8
NF	10	1	5	5
NAC	169.2	298	176	3353.6

We draw, for each test fold, the Receiver Operating Characteristics (ROC) and the Precision-Recall (PR) curves and compute the area under each curve (AUCROC and AUCPR) as described in [25]. The average values (over the folds) of the areas for each algorithm are shown in Tables 13.3 and 13.4. Table 13.5 shows the average training time. Note that EMBLEM ran out the memory on the Carcinogenesis and Mondial datasets and we could not compute the areas and the training time. Moreover, to start learning, LFI-ProbLog needed more memory than PHIL. From the experiment, PHIL beats EMBLEM and LFI-ProbLog either in terms of area or in terms of time in all datasets. In Table 13.5, we highlight in italic the times associated with the best accuracies from Tables 13.3 and 13.4. It can be observed that these times are either the best or in the same order of the best time, in bold. Among DPHIL (resp. EMPHIL) and its regularized versions, $DPHIL_2$ (resp. $EMPHIL_2$) is often a good compromise in terms of accuracy and time. Note also that regularization is often not necessary in dataset with few clauses such as the Mondial dataset. Between DPHIL and EMPHIL, DPHIL is often convenient in dataset with many clauses and examples (e.g. Mutagenesis).

AUCROC	Mutagenesis	Carcinogenesis	Mondial	UWCSE
DPHIL	**0.888943**	0.602632	0.531157	0.941525
$DPHIL_1$	0.841021	0.571053	0.534817	0.960876
$DPHIL_2$	0.880465	0.618421	0.534563	0.949548
EMPHIL	0.885358	**0.684211**	0.534822	0.968560
$EMPHIL_1$	0.884016	**0.684211**	0.536009	0.938121
$EMPHIL_2$	0.885478	0.623684	0.534622	**0.969046**
$EMPHIL_{3_1}$	0.833539	0.61973	0.536042	0.930243
$EMPHIL_{3_2}$	0.821356	0.64078	**0.537011**	0.930243
$EMPHIL_{3_3}$	0.820220	0.64078	0.534996	0.930243
EMBLEM	0.887695	-	-	0.968354
ProbLog2	0.828655	0.594737	0.533905	0.968909

Table 13.3: Average area under ROC curve.

AUCPR	Mutagenesis	Carcinogenesis	Mondial	UWCSE
DPHIL	**0.947100**	0.595144	0.138932	0.227438
DPHIL$_1$	0.886598	0.563875	0.142331	0.191302
DPHIL$_2$	0.929244	0.580041	**0.147390**	0.219806
EMPHIL	0.944511	0.679966	0.142374	0.277760
EMPHIL$_1$	0.944758	**0.679712**	0.142696	0.275985
EMPHIL$_2$	0.944517	0.655781	0.142066	**0.307713**
EMPHIL$_{3_1}$	0.880013	0.64909	0.142810	0.261578
EMPHIL$_{3_2}$	0.868837	0.64163	0.143275	0.261578
EMPHIL$_{3_3}$	0.867759	0.64163	0.142540	0.261578
EMBLEM	0.944394	-	-	0.262565
ProbLog2	0.901450	0.568821	0.132498	0.306378

Table 13.4: Average area under PR curve.

Time	Mutagenesis	Carcinogenesis	Mondial	UWCSE
DPHIL	*2.8573*	178.268	265.416	**0.0884**
DPHIL$_1$	5.2059	177.427	311.328	0.291
DPHIL$_2$	5.445	88.51	301.1392	0.2214
EMPHIL	4.443	*106.554*	270.6688	0.289
EMPHIL$_1$	4.894	*181.089*	317.4202	1.000
EMPHIL$_2$	5.0046	146.844	**245.383**	*0.8372*
EMPHIL$_{3_1}$	2.6478	85.3210	248.1978	0.157200
EMPHIL$_{3_2}$	**1.582**	**80.427**	*261.3612*	0.1266
EMPHIL$_{3_3}$	1.4937	94.6850	274.959800	0.119000
EMBLEM	125.621400	-	-	0.9666
ProbLog2	722.0	38685.0	1607.6	161.4

Table 13.5: Average time

Algorithm 18 Function EMPHIL.

```
function EMPHIL($Theory, \epsilon, \delta, MaxIter, \gamma, a, b, Type$)
    $Examples \leftarrow$ BUILDACS($Theory$)                    ▷ Build the set of ACs
    for $i \leftarrow 1 \rightarrow |Theory|$ do
        $\Pi[i] \leftarrow random; B[i], Count[i] \leftarrow 0$          ▷ Initialize the parameters
    end for
    $LL \leftarrow -inf; Iter \leftarrow 0$
    repeat
        $LL_0 \leftarrow LL, LL \leftarrow 0$                              ▷ Expectation step
        for all $node \in Examples$ do
            $P \leftarrow$ FORWARD($node$)
            BACKWARDEM($1, node, N_0, N_1$)
            $LL \leftarrow LL + \log P$
        end for                                                   ▷ Maximization step
        for $i \leftarrow 1 \rightarrow |Theory|$ do
            switch $Type$
              case 0: $\Pi[i] \leftarrow \frac{B[i]}{N_0[i]+N_1[i]}$
              case 1: $\Pi[i] \leftarrow \frac{4N_1[i]}{2(\gamma+N_0[i]+N_1[i]+\sqrt{(N_0[i]+N_1[i])^2+\gamma^2+2\gamma(N_0[i]-N_1[i])})}$
              case 2:
                 let $V = 2\sqrt{\frac{3N_0+3N_1+\gamma}{\gamma}} \cos\left(\frac{\arccos\left(\frac{\sqrt{\frac{\gamma}{3N_0+3N_1+\gamma}}\left(\frac{9N_0}{2}-9N_1+\gamma\right)}{3N_0+3N_1+\gamma}\right)}{3} - \frac{2\pi}{3}\right)$
                 $\Pi[i] \leftarrow \frac{V}{3} + \frac{1}{3}$
              case 3: $\Pi[i] \leftarrow \frac{N_1+a}{N_0+N_1+a+b}$
            end switch
            $B[i], Count[i] \leftarrow 0$
        end for
    until $LL - LL_0 < \epsilon \vee LL - LL_0 < -LL.\delta \vee Iter > MaxIter$
    $FinalTheory \leftarrow$ UPDATETHEORY($Theory, \Pi$)
    return $FinalTheory$
end function
```

Chapter 14

Structure learning of Hierarchical Probabilistic Logic Programming

In order to estimate the parameters of HPLPs in chapter 13, we considered their structure as known, possibly manually constructed by an expert or generated by a program, and presented algorithms for learning the parameters from data. The aim of this chapter is to describe another algorithm for learning both the structure and the parameters of HPLPs from data. The algorithm is called Structure LEArning of Hierarchical Probabilistic logic programming (SLEAHP). SLEAHP initially generates a large HPLP from bottom clauses obtained from a language bias as described in [11] and subsequently applies a regularized version of PHIL on the generated HPLP to cut clauses with small parameter values. We performed experiments comparing different regularization versions of SLEAHP with the state-of-the-art structure learning algorithms SLIPCOVER, [11] and ProbFOIL+, [105]. The results show that SLEAHP achieves similar and often better accuracies but in a shorter time. The chapter is organized as follows: after presenting the structure learning problem in Section 14.1, Sections 14.2 and 14.3 describe the algorithm and present related works respectively. Finally Section 14.4 presents some experiments.

14.1 Overview

In Probabilistic Inductive Logic Programming, one of the most challenging task is to induce a Probabilistic Logic Program from data. The task is known in literature as structure learning problem. Note that inducing the structure of PLP is often necessary in domains in which even an expert could not construct a good structure that generalizes well. Learning the structure means searching in space of potential programs (in this case HPLPs) the one, and its parameters, that best represents the data. The structure learning problem in HPLP can be defined as follows:

Definition 16 (Structure Learning Problem)**.** *Given an interpretation I defining input predicates and a training set of positive and negative examples $E = \{e_1, \ldots, e_M, \textbf{not } e_{M+1}, \ldots, \textbf{not } e_N\}$ where each e_i is a ground atom for the target predicate r, find an HPLP H and its parameters Π that maximize the log likelihood (LL)*

$$LL = \arg\max_{\Pi} \sum_{i=1}^{M} \log P(e_i) + \sum_{i=M+1}^{N} \log(1 - P(e_i)) \tag{14.1}$$

where $P(e_i)$ is the probability assigned to e_i by $H \cup I$.

SLEAHP learns HPLPs by generating an initial set of bottom clauses, from the language bias defined in Section 11.5.1, from which a large HPLP is derived. SLEAHP then proceeds by applying a regularized parameter learning on the initial HPLP. Regularization is used to bring as many parameters as possible close to 0 so that their clauses can be removed, thus pruning the initial large program and keeping only useful clauses.

14.2 Description of the algorithm

In order to learn an HPLP, SLEAHP (Algorithm 19) initially generates a set of bottom clauses, line 2. Then an n-ary tree whose nodes are literals appearing in the head or in the body of bottom clauses is constructed, line 3. An initial HPLP is generated from the tree, line 4, and a regularized version of PHIL is performed on the initial program. Finally clauses with very small probabilities are removed, line 5.

Note that the language bias and the algorithms for selecting the head and for saturating the body of bottom clauses are the same as those used in liftable

PLP, see sections 11.5.1 and 11.5.2. The unique difference is that the generated bottom clauses are not labeled with initial probabilities and are of the form

$$BC = h :- b_1, \ldots, b_m \tag{14.2}$$

Considering that bottom clauses have been generated and are of the form in Equation 14.2, the following sections describe how to create the tree from these bottom clauses and how to generate the initial HPLP from the tree.

Algorithm 19 Function STRUCTURE LEARNING

```
1: function SLEAHP($NInt, NS, NA, MaxProb, NumLayer, MaxIter, \epsilon, \delta, MinProb$)
2:     $Clauses$ =genClauses($NInt, NS, NA$)                         ▷ Generate clauses
3:     $Tree$=genTree(Clauses)                                          ▷ Build the tree
4:     $init\_HPLP$=genHPLP(Clauses,MaxProb,NumLayer)          ▷ Generate the initial HPLP
5:     ($LL, final\_HPLP$) ←PHIL_Reg($init\_HPLP, MaxIter, \epsilon, \delta$)      ▷ Learns the parameters
6:     return $final\_HPLP$
7: end function
```

14.2.1 Tree Generation

Since an HPLP can be mapped to a tree as described in Section 12.2, we create a tree whose nodes are literals appearing in the head or in the body of bottom clauses generated, Equation 14.2, as described in the previous section. Every node in the tree shares at least one variable with its parent. The tree is then converted to a large HPLP, see Section 14.2.2.

To create the tree, Algorithm 20 starts by considering each bottom clause in turn, line 3. Each bottom clause creates a small tree, lines 4 - 11. Consider the following bottom clause

$$BC = r(Arg) :- b_1(Arg_1), \ldots, b_m(Arg_m)$$

where Arg and Arg_i are tuples of arguments and $b_i(Arg_i)$ are literals. Initially $r(Arg)$ is set as the root of the tree, lines 5. Literals in the body are considered in turn from left to right. When a literal $b_i(Arg_i)$ is considered, the algorithm tries to insert the literal in the tree, see Algorithm 21. If $b_i(Arg_i)$ cannot be inserted, it is set as the right-most child of the root. The algorithm proceeds until all the $b_i(Arg_i)$ are inserted into the tree, lines 6–10. Then the resulting small tree is appended to a list of trees (initially empty), line 11, and the list is merged obtaining a unique tree, line 13. The trees in L are merged by unifying the arguments of their roots.

Algorithm 20 GENERATE TREE

```
function genTree($Bottoms$)
    $L \leftarrow []$
    for all $Bottom \in Bottoms$ do
        let $Bottom = r(Arg) :- b_1(Arg_1), \ldots, b_m(Arg_m)$
        $Tree \leftarrow r(Arg)$                    ▷ $r(Arg)$ is the root of the tree
        for all $b_i(Arg_i)$ do
            if $not$(insertTree($Tree, b_i(Arg_i)$)) then
                addChild($r(Arg), b_i(Arg_i)$))
            end if
        end for
        Append Tree to L
    end for
    $final\_Tree \leftarrow mergeTrees(L)$
    return $final\_Tree$
end function
```

To insert the literal $b_i(Arg_i)$ into the tree, Algorithm 21 visits the tree depth-first. When a node $b(Arg)$ is visited, if Arg and Arg_i share at least one variable, $b_i(Arg_i)$ is set as the right-most child of $b(Arg)$ and the algorithm stops and returns *True*. Otherwise InsertTree is recursively called on each child of $b(Arg)$, lines 6 - 12. The algorithm returns *False* if the literal cannot be inserted after visiting all the nodes, line 3.

Example 24. *Consider the following bottom clause from the UWCSE dataset:*

$advised\_by(A, B) :-$
$student(A), professor(B), has\_position(B, C),$
$publication(D, B), publication(D, E), in\_phase(A, F),$
$taught\_by(G, E, H), ta(I, J, H).$

In order to build the tree, $advised\_by(A, B)$ is initially set as the root of the tree. Then literals in the body are considered in turn. The literals $student(A)$ (resp. $professor(B)$, $hasposition(B, C)$ and $publication(D, B)$) are set as the children of $advised\_by(A, B)$ because they share variable A (resp. B). Then the literal $publication(D, E)$ is set as a child of $publication(D, B)$ because they share variable D, $in\_phase(A, F)$ as a child of $advised\_by(A, B)$ (they share

Algorithm 21 INSERT A LITERAL INTO A TREE

```
1: function INSERTTREE($Tree$,$b_i(Arg_i)$)
2:     if $Tree$=$NULL$ then                                ▷ All nodes are visited
3:         return $False$
4:     else
5:         let the root of $Tree = b(Arg)$
6:         if $shareArgument(Arg, Arg_i)$ then
7:             ADDCHILD($Tree$,$b_i(Arg_i)$)
8:             return $True$
9:         else
10:            for all $Child$ of $Tree$ do
11:                return INSERTTREE($Child, b_i(Arg_i)$)
12:            end for
13:        end if
14:    end if
15: end function
```

variable A), $taught\_by(G, E, H)$ as a child of $publication(D, E)$ (they share variable E), $taught\_by(G, E, H)$ as a child of $publication(D, E)$ (they share variable E) and finally $ta(I, J, H)$ as a child of $taught\_by(G, E, H)$ (they share variable H). The corresponding tree is shown in Figure 14.1.

14.2.2 HPLP Generation

Once the tree is built, an initial HPLP is generated at random from the tree. Before describing how the program is created, note that for simplicity, we considered clauses with at most two literals in the body. This can be extended to any number of literals. Algorithm 22 takes as input the tree, $Tree$, initial probability, $0 \leq MaxProb \leq 1$, a rate, $0 \leq rate \leq 1$, and the maximum number of layers $NumLayer$ of HPLP we are about to generate. Let $\mathbf{X}_k$, $\mathbf{Y}_l = Y_1, \cdots, Y_l$, $\mathbf{Z}_t$ and $\mathbf{W}_m$ be tuples of variables.

In order to generate the initial HPLP, the tree is visited breadth-first, starting from level 1. For each node n_i at level $Level$ ($1 \leq Level \leq NumLayer$), n_i is visited with probability $Prob$. Otherwise n_i and the subtree rooted at n_i are not visited. $Prob$ is initialized to $MaxProb$ which is typically 1.0 by default. The new value of $Prob$ at each level is $Prob * rate$ where $rate \in [0, 1]$ is a constant value typically 0.95 by default. Thus the deeper the level, the

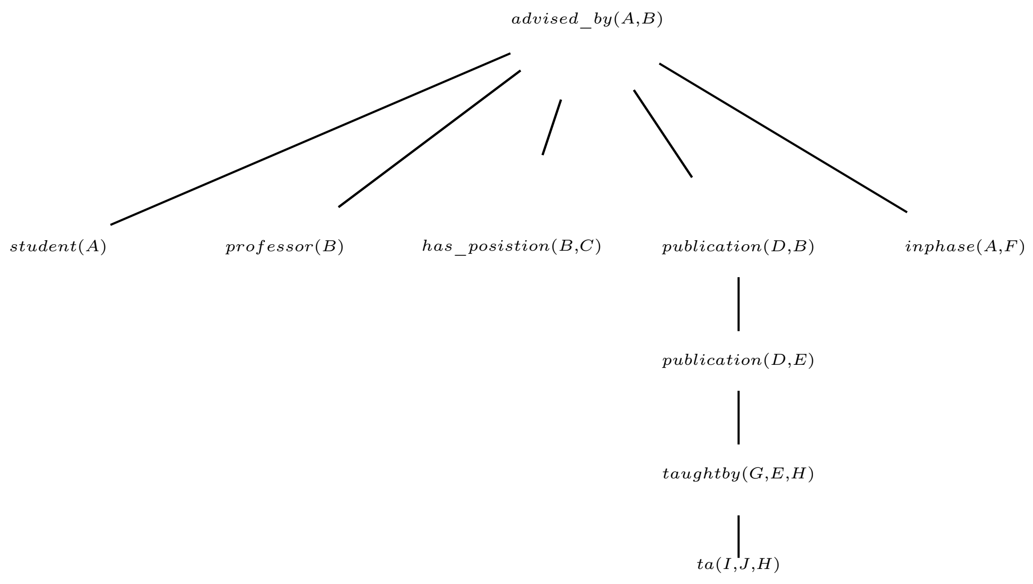


Figure 14.1: Tree created from the bottom clause of Example 24.

.

lower the probability value. Supposing that n_i is visited, two cases can occur: n_i is a leaf or an internal node.

If $n_i = b_i(\mathbf{Y}_i)$ is a leaf node with parent $Parent_i$, we consider two cases. If $Parent_i = r(\mathbf{X}_k)$ (the root of the tree), then the clause

$$C = r(\mathbf{X}_k) : 0.5 :-\ b_i(\mathbf{Y}_i).$$

is generated, lines 9-11. Otherwise

$$C = hidden_{path}(\mathbf{Z}_{t_i}) : 0.5 :-\ b_i(\mathbf{Y}_{l_i}).$$

is generated. $hidden_{path}(\mathbf{Z}_{t_i})$ is the hidden predicate associate with $Parent_i$ and *path* is the path from the root to $Parent_i$, lines 13-16.

If $n_i = b_i(\mathbf{Y}_i)$ is an internal node having parent $Parent_i$, we consider two cases. If $Parent_i$ is the root, the clause

$$C = r(\mathbf{X}_k) : 0.5 :-\ b_i(\mathbf{Y}_i), hidden\_l(\mathbf{Z}_{t_i}).$$

is generated. $hidden\_l(\mathbf{Z}_{t_i})$ is associated with $b_i(\mathbf{Y}_i)$ and $\mathbf{Z}_{t_i} = \mathbf{X}_k \cup \mathbf{Y}_i$, lines 20-21. If $Parent_i$ is an internal node with the associated hidden predicate $hidden_{path}(\mathbf{Z}_t)$ then the clause

$$C = hidden_{path}(\mathbf{Z}_t) : 0.5 :- \ b_i(\mathbf{Y}_i), hidden_{path\_i}(\mathbf{W}_{m_i}).$$

is generated where $\mathbf{W}_{m_i} = \mathbf{Z}_t \cup \mathbf{Y}_i$, lines 24-25.

The generated clause C is added to a list (initially empty), line 28, and the algorithm proceeds for every node at each level until layer $NumLayer$ is reached or all nodes in the tree are visited, line 5. Then hidden predicates appearing in the body of clauses without associated clauses (in the next layer) are removed, line 34, and the program is reduced, line 35. To reduce the program, clauses (without hidden predicates in the body) having the same input predicate (with different arguments) in the body are reduced into one clause (generally the first of them).

Example 25. *The HPLP generated from the tree of Figure 14.1 is:*

$advised\_by(A, B) : 0.5 :- \ student(A).$
$advised\_by(A, B) : 0.5 :- \ professor(B).$
$advised\_by(A, B) : 0.5 :- \ has\_position(B, C).$
$advised\_by(A, B) : 0.5 :- \ publication(D, B), hidden_1(A, B, D).$
$advised\_by(A, B) : 0.5 :- \ in\_phase(A, E).$
$hidden_1(A, B, D) : 0.5 :- \ publication(D, F), hidden_{11}(A, B, D, F).$
$hidden_{11}(A, B, D, F) : 0.5 :- \ taught\_by(G, F, H), hidden_{111}(A, B, D, F, G, H).$
$hidden_{111}(A, B, D, F, G, H) : 0.5 :- \ ta(I, J, H).$

which is a HPLP with 3 levels.

14.3 Related work

SLEAHP is related to SLIPCASE [9], SLIPCOVER [11] which are algorithms for learning general PLP, and LIFTCOVER which is an algorithm described in Chapter 11 for learning liftable PLP. In liftable PLP the head of all the clauses in the program share the same predicate (the target predicate) and

Algorithm 22 Function GenerateHPLP

```
function GenerateHPLP($Tree, MaxProb, Rate, NumLayer$)
    $HPLP \leftarrow []$
    $Level \leftarrow 1$
    $Prob \leftarrow MaxProb$
    while $Level < NumLayer$ and all nodes in $Tree$ are not visited do
        for all node $n_i$ at level $Level$ having parent $Parent_i$ do
            if maybe(Prob) then
                if $n_i$ is a leaf node then                          ▷ $n_i$ is a leaf node
                    if $Parent_i$ is the root node then
                        let $n_i = b_i(\mathbf{Y}_i)$ and $Parent_i = r(\mathbf{X}_k)$ then
                        $C = r(\mathbf{X}_k) : 0.5 :- b_i(\mathbf{Y}_i)$.
                    else
                        let $Parent_i = b_{path}(X_k)$
                        $hidden_{path}(\mathbf{Z}_{t_i})$ associate with $b_{path}(X_k)$
                        $n_i = b_i(\mathbf{Y}_i)$ then
                        $C = hidden_{path}(\mathbf{Z}_{t_i}) : 0.5 :- b_i(\mathbf{Y}_i)$.
                    end if
                else                                                  ▷ $n_i$ is an internal node
                    if $Parent_i$ is the root node then
                        let $n_i = b_i(\mathbf{Y}_i)$ and $\mathbf{Z}_{t_i} = \mathbf{X}_k \cup \mathbf{Y}_i$
                        $C = r(\mathbf{X}_k) : 0.5 :- b_i(\mathbf{Y}_i), hidden\_i(\mathbf{Z}_{t_i})$.
                    else
                        let $Parent = b_{path}(X_k)$
                        $n_i = b_i(\mathbf{Y}_i)$ and $\mathbf{W}_{m_i} = \mathbf{Z}_t \cup \mathbf{Y}_i$ then

            $C = hidden_{path}(\mathbf{Z}_t) : 0.5 :- b_i(\mathbf{Y}_i), hidden_{path\_i}(\mathbf{W}_{m_i})$.

                    end if
                end if
                $HPLP \leftarrow [C|HPLP]$
            end if
        end for
        $Prob \leftarrow Prob * Rate$
        $level \leftarrow level + 1$
    end while
    $HPLP \leftarrow removeHidden(HPLP)$
    $initial\_HPLP \leftarrow reduce(HPLP)$
    Return initial_HPLP
end function
```

their bodies contain only input predicates. Therefore, Liftable PLP can be seen as a restriction of HPLP without hidden predicates and clauses. Both LIFTCOVER and SLIPCOVER learn the program by performing a search in the space of clauses and then refine the search by greedily adding refined clauses into the theory, while SLIPCASE performs search by theory revision. SLIPCASE and SLIPCOVER uses EMBLEM to compute the log-likehood of the data and LIFTCOVER uses a Expectation Maximization and Gradient descent-based method, see Sections 11.4.1 and 11.4.2.

SLEAHP is also related to ProbFOIL+ [105] which is a generalization of mFOIL [34] that learns both the structure and the parameters of ProbLog programs by performing a hill climbing search in the space of programs as explained in Section 10.3.2.

Similar to LIFTCOVER, SLIPCOVER and ProbFOIL+, SLEAHP initially performs a search in the space of clauses but differently from these systems, it creates a tree of literals from which a large HPLP is generated. Then a regularized version of PHIL is applied to the HPLP.

SLEAHP performs a form of predicate invention: the hidden predicates represent new predicates that are not present in the data. Much work has been devoted to predicate invention. In [20, 19] for example the authors propose algorithms that are able to perform predicate invention. Both proposals rely on a form of language bias based on metarules, i.e., rule skeletons where the predicate of literals is a not specified. Learning is then performed by metainterpretation and new predicates are introduced when applying the metarules. These works focus on learning programs: the learned theory often involve recursion. Instead, the language of HPLP is less expressive, as recursion is not allowed, and we are focused on learning probabilistic classifiers, i.e., functions returning the class of an individual given what is not about him.

Ground HPLPs can be also seen as neural network (NNs) where the nodes in the arithmetic circuits are the neurons and the activation function of nodes is the probabilistic sum. Parameter learning by DPHIL is in fact performed as in NNs by backpropagation. Combining logical languages with NNs is an active research field, see [30] for an excellent review. For example, Relational Neural Networks (RelNNs) [53] generalize Relational Logistic Regression (RLR) by stacking multiple RLR layers together. The authors provide strong motivations for having multiple layers, highlighting in particular that they improve the representation power by allowing aggregation at different levels and on different object populations. HPLP benefit from the same advantage. Moreover, HPLP keep a semantics as probabilistic logic programs: the output of the network is a probability according to the distribution semantics.

14.4 Experiments

In this section, we present experiments [1] comparing SLEAHP with SLIPCOVER and ProbFOIL+ [105]. We used the same language bias expressed in terms of modes for SLEAHP, SLIPCOVER and ProbFOIL+. To generate the initial HPLP in SLEAHP, we used $MaxProb = 1.0$, $Rate = 0.95$ and $Maxlayer = +\infty$ for every dataset except in UWCSE where we used $Maxlayer = 3$. After generating the initial HPLP we use the same hyper-parameters presented in Section 13.5.2 to perform parameter learning. We performed five experiments (one for each regularization). SLEAHP$_{G_1}$ (resp. SLEAHP$_{G_2}$) uses DPHIL$_1$ (resp. DPHIL$_2$) and SLEAHP$_{E_1}$ (resp.SLEAHP$_{E_2}$ and SLEAHP$_{E_3}$) uses EMPHIL$_1$ (resp. EMPHIL$_2$ and EMPHIL$_{3_1}$). The average area under the ROC/PR curves and the average time are shown in Tables 14.1, 14.2, 14.3 respectively. Note that the training time and the average areas in SLIPCOVER have been taken from [33] where experiments were performed on GNU/Linux machines with an Intel Core 2 Duo E6550 (2,333 MHz). The training times for SLIPCOVER were obtained by converting a time from cpu with 2,333 MHz to the one with 2,355 MHz. The results illustrated with - for ProbFOIL+ indicate that it was not able to terminate in 24 hours, in Mondial on some folds and in UWCSE on all folds.

In terms of accuracy, SLEAHP outperforms SLIPCOVER in two datasets, Mutagenesis and UWCSE, and achieves similar quality solution in the other datasets. Note that SLEAHP$_{E_1}$ and other EM regularizations do not perform well, in terms of AUCROC and AUCPR, on the UWCSE dataset. This highlight the more restricted expressiveness of HPLPs in general. Moreover, on the same dataset SLEAHP$_{G_1}$ SLEAHP$_{G_2}$ outperform SLIPCOVER in terms of solution

[1]Experiments are available at `https://bitbucket.org/ArnaudFadja/hierarchicalplp_experiments/src/master/`.

quality and time. This motivate the use of different types of algorithms and regularizations.

In terms of computation time, SLEAHP outperforms SLIPCOVER in almost all datasets excepts in UWCSE in which the computation time still remains reasonable. In Table 14.3, we also highlight in italic, as done in Section 13.5.2, the times associated with the best accuracies from Tables 14.1 and 14.2. Between DPHIL and EMPHIL regularizations, as stated in Section 13.5.2, DPHIL is often preferred in dataset with large examples (see UWCSE).

In terms of accuracy and time, SLEAHP outperforms ProbFOIL+ in all datasets. To summarize, SLEAHP beats SLIPCOVER and ProbFOIL+ in terms of computation time in almost all datasets and achieves similar accuracy.

Table 14.1: Average area under ROC curve.

AUCROC	Mutagenesis	Carcinogenesis	Mondial	UWCSE
SLEAHP$_{G_1}$	0.889676	0.493421	0.483865	**0.93616**
SLEAHP$_{G_2}$	0.845452	0.544737	0.472843	0.925436
SLEAHP$_{E_1}$	0.878727	0.660526	0.433016	0.907789
SLEAHP$_{E_2}$	**0.904933**	0.414135	0.483798	0.904347
SLEAHP$_{E_3}$	0.822833	0.618421	0.464058	0.925099
SLIPCOVER	0.851	**0.676**	**0.600**	0.919
ProbFOIL+	0.881255	0.556578	-	-

Table 14.2: Average area under the PR curve.

AUCPR	Mutagenesis	Carcinogenesis	Mondial	UWCSE
SLEAHP$_{G_1}$	0.929906	0.498091	0.701244	**0.148115**
SLEAHP$_{G_2}$	0.918519	0.502135	0.690782	0.13175
SLEAHP$_{E_1}$	0.948563	0.598095	0.632270	0.059562
SLEAHP$_{E_2}$	**0.955678**	0.540510	0.623542	0.069861
SLEAHP$_{E_3}$	0.9003	0.552477	0.623542	0.059655
SLIPCOVER	0.885	**0.676**	**0.733792**	0.113
ProbFOIL+	0.937497	0.534393	-	-

Table 14.3: Average time.

Time	Mutagenesis	Carcinogenesis	Mondial	UWCSE
SLEAHP$_{G_1}$	41.825	**48.76**	59.5054	*219.641*
SLEAHP$_{G_2}$	47.1344	10524.09	**14.047**	194.9706
SLEAHP$_{E_1}$	48.0152	303.057	60.8316	387.665
SLEAHP$_{E_2}$	*45.9245*	92.382	61.09959	312.2604
SLEAHP$_{E_3}$	**13.1478**	1399.009	14.6698	295.6734
SLIPCOVER	74610.70	*17419.45*	*650.363*	**141.36**
ProbFOIL+	1726.6	15433	-	-

Part VII

Summary and Future Work

Chapter 15

Conclusion

Recently, because of the huge amount of devices inter-connected on the web and the large amount of data generated by these devices, it has become of foremost important to build systems that are both able to model and represent different real world domains and able to manage large quantities of data. With the advent of big data, present and future systems, to perform adequately, should be strongly scalable.

The Distribution Semantics is an expressive and mature formalism that integrates logic and probability for modeling domains characterized by uncertainty. This formalism is at the basis of many languages including LPADs and ProbLog, which have been recently used for representing data in various fields. Many systems have been proposed for reasoning and learning from data. Notwithstanding the attempt to implement distributed algorithms to achieve scalability, the problem of performance still remains a challenge when the size of the data increases.

The aim of this thesis is to provide languages under the DS in which reasoning and learning are less expensive. Systems adopting such languages will be more scalable and will be able to manage ever-increasing data. The contribution of this thesis is fourfold:

1. **Probabilistic Logic Program in action**
   We proposed a set of examples in real world domains modeled by LPADs. This set of examples shows the expressiveness and the maturity of LPADs and motivated the choice of this formalism for representing data, reasoning and learning from them.

2. **Languages under the distribution semantics**
   We proposed and described two languages under the DS, which are restrictions of LPADs called Liftable PLP (LPLP) and Hierarchical PLP (HPLP). Reasoning (inference) in these languages is less expensive than for general LPADs. Inference is performed in LPLPs at a lifted level, that is by reasoning on whole populations of individuals instead of considering each individual separately. In HPLPs, each program is converted into set of Arithmetic Circuits (ACs) (deep neural networks) sharing parameters and inference is done by evaluating the ACs.

3. **Parameter learning**
   We proposed two algorithms for learning the parameters of LPLPs. The first is based on the Expectation Maximization (EM) algorithm and the second is based on quasi-Newton optimization called Limited-memory BFGS (LBFGS).
   We also presented a parameter learning algorithm for HPLP, called Parameter learning for HIerarchical probabilistic Logic programs PHIL, that learns the parameters of HPLPs from data. We presented two versions of PHIL: Deep PHIL (DPHIL) that learns the parameters by applying a gradient descent method (in particular the ADAM optimizer) and Expectation Maximization PHIL (EMPHIL) which applies the EM algorithm. Different regularized versions of DPHIL (DPHIL$_1$, DPHIL$_2$) and EMPHIL (EMPHIL$_1$, EMPHIL$_2$, EMPHIL$_3$) have also been proposed. These regularizations favor small parameters during learning. We performed experiments on different real world datasets comparing PHIL with state of the art parameter learning algorithms EMBLEM and LFI-ProbLog. PHIL achieves similar and often better accuracies in a shorter time.

4. **Structure learning**
   For LPLPs, we presented the algorithm LIFTCOVER that is similar to SLIPCOVER and that learns the structure of LPLPs using either EM (LIFTCOVER-EM) or LBFGS (LIFTCOVER-LBFGS) for parameter learning. LIFTCOVER, as SLIPCOVER, initially performs a beam search in the space of clauses for finding the promising ones. Then, while SLIPCOVER performs a hill climbing search in the space of the-

ories, LIFTCOVER performs parameter learning on the whole set of promising clauses and reduces the theory by eliminating clauses whose parameter values are small. This is done because inference and learning in LPLPs are less expensive. The results show that LIFTCOVER-EM and LIFTCOVER-LBFGS are faster than SLIPCOVER and often more accurate, with LIFTCOVER-EM performing slightly better than LIFTCOVER-LBFGS. These results show that we can reduce the hypothesis space of LPADs while still keeping a reasonable accuracy and by considerably decreasing the inference and the learning time making inference and learning in many domains of interest more scalable.
We also proposed in this thesis an algorithm called Structure LEArning of Hierarchical Probabilistic logic programming (SLEAHP) that learns both the structure and the parameters of HPLPs from data. SLEAHP initially generates a large HPLP and then applies a regularized version of PHIL for performing parameter learning. Clauses with small parameters are removed from the final program. Experiments comparing SLEAHP with SLIPCOVER and ProbFOIL+ in different real world datasets show that, in terms of accuracies, SLEAHP achieves similar results as SLIPCOVER and ProbFOIL+ but beats them in terms of time in almost all datasets.

Chapter 16

Future work

Algorithms based on the languages proposed in this work are, in terms of time, better than the state of the art. Moreover, much work is still necessary for improving their expressiveness and scalability.

- **Expressiveness**
  Regarding the restriction imposed on the number of literals in the body of clauses in SLEAHP, we plan to increase the expressiveness of HPLPLs by increasing this number in order to explore a larger space of HPLPs. We also plan to extend HPLPs to domains with continuous random variables in order to apply PHIL and SLEAHP on data such as images.

- **Parameter learning**
  To improve the learning time of PHIL, we plan to implement its distributed versions based for example on Map-reduce strategy. In these versions, the generation of the ACs together with the computation of the expectations and the gradients could be parallelized. Systems implementing these parallelized versions will be more scalable and will be able to manage larger data.

- **Structure learning**
  We plan to implement a hill-climbing algorithm for learning the structure of HPLPs. After the generation of a large HPLP, as done in SLEAHP, the algorithm will perform a search in a space of theories guided by the likelihood of data.
  We also plan to implement distributed versions of LIFTCOVER and

SLEAHP and compare them with other systems designed for scalability such as [85, 49].

- **Integration of HPLPs and deep learning**
  Other interesting direction to investigate is how to integrate symbolic approaches of machine learning, such as HPLPs, with other approaches based on neural networks, such as deep learning, in order to combine their strengths and build better systems. Systems such as DeepLogic [16] and DeepProbLog [72] have proposed algorithms for combining both approaches but much work still remain useful to investigate.

Bibliography

[1] Sheldon B. Akers. Binary decision diagrams. 27(6):509–516, 1978.

[2] Marco Alberti, Elena Bellodi, Giuseppe Cota, Fabrizio Riguzzi, and Riccardo Zese. `cplint` on SWISH: Probabilistic logical inference with a web browser. 11(1):47–64, 2017.

[3] Dalal Alrajeh and Alessandra Russo. Logic-based learning: Theory and application. In *Machine Learning for Dynamic Software Analysis: Potentials and Limits*, pages 219–256. Springer, 2018.

[4] K. R. Apt and M. Bezem. Acyclic programs. 9(3/4):335–364, 1991.

[5] Stephen H. Bach, Matthias Broecheler, Bert Huang, and Lise Getoor. Hinge-loss markov random fields and probabilistic soft logic. *arXiv preprint arXiv:1505.04406 [cs.LG]*, 2015.

[6] Niko Beerenwinkel, Jörg Rahnenführer, Martin Däumer, Daniel Hoffmann, Rolf Kaiser, Joachim Selbig, and Thomas Lengauer. Learning multiple evolutionary pathways from cross-sectional data. *Journal of Computational Biology*, 12:584–598, 2005.

[7] Elena Bellodi, Evelina Lamma, Fabrizio Riguzzi, Vítor Santos Costa, and Riccardo Zese. Lifted variable elimination for probabilistic logic programming. 14(4-5):681–695, 2014.

[8] Elena Bellodi and Fabrizio Riguzzi. Experimentation of an expectation maximization algorithm for probabilistic logic programs. 8(1):3–18, 2012.

[9] Elena Bellodi and Fabrizio Riguzzi. Learning the structure of probabilistic logic programs. In StephenH. Muggleton, Alireza Tamaddoni-Nezhad,

and FrancescaA. Lisi, editors, *22nd International Conference on Inductive Logic Programming*, volume 7207 of *LNCS*, pages 61–75. Springer Berlin Heidelberg, 2012.

[10] Elena Bellodi and Fabrizio Riguzzi. Expectation maximization over binary decision diagrams for probabilistic logic programs. 17(2):343–363, 2013.

[11] Elena Bellodi and Fabrizio Riguzzi. Structure learning of probabilistic logic programs by searching the clause space. 15(2):169–212, 2015.

[12] C.M. Bishop. *Pattern Recognition and Machine Learning*. Information Science and Statistics. 2016.

[13] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent Dirichlet allocation. 3:993–1022, 2003.

[14] Ashok K. Chandra and David Harel. Horn clauses queries and generalizations. 2(1):1–15, 1985.

[15] Cheng Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary Bradski, Andrew Y Ng, and Kunle Olukotun. Map-reduce for machine learning on multicore. pages 281–288, 2006.

[16] Nuri Cingillioglu and Alessandra Russo. Deeplogic: Towards end-to-end differentiable logical reasoning. *arXiv preprint arXiv:1805.07433*, 2018.

[17] David A. Cox. *Galois Theory*. Pure and Applied Mathematics: A Wiley Series of Texts, Monographs and Tracts. 2012.

[18] Fábio Gagliardi Cozman and Denis Deratani Mauá. The structure and complexity of credal semantics. volume 1661 of *CEUR Workshop Proceedings*, pages 3–14. CEUR-WS.org, 2016.

[19] Andrew Cropper, Rolf Morel, and Stephen H Muggleton. Learning higher-order logic programs. *arXiv preprint arXiv:1907.10953*, 2019.

[20] Andrew Cropper and Stephen H. Muggleton. Learning efficient logic programs. *Machine Learning*, 108(7):1063–1083, Jul 2019.

[21] Evgeny Dantsin. Probabilistic logic programs and their semantics. In *Russian Conference on Logic Programming*, volume 592 of *LNCS*, pages 152–164. Springer, 1991.

[22] Adnan Darwiche. A differential approach to inference in bayesian networks. 50(3):280–305, 2003.

[23] Adnan Darwiche and Pierre Marquis. A knowledge compilation map. 17:229–264, 2002.

[24] J. Davis and M. Goadrich. The relationship between precision-recall and ROC curves. pages 233–240. ACM, 2006.

[25] Jesse Davis and Mark Goadrich. The relationship between precision-recall and roc curves. In *Proceedings of the 23rd international conference on Machine learning*, pages 233–240. ACM, 2006.

[26] L. De Raedt, B. Demoen, D. Fierens, B. Gutmann, G. Janssens, A. Kimmig, N. Landwehr, T. Mantadelis, W. Meert, R. Rocha, V. Santos Costa, I. Thon, and J. Vennekens. Towards digesting the alphabet-soup of statistical relational learning. In *NIPS 2008 Workshop on Probabilistic Programming*, 2008.

[27] Luc De Raedt and Saso Džeroski. First-Order jk-Clausal Theories are PAC-Learnable. 70(1-2):375–392, 1994.

[28] Luc De Raedt and Angelika Kimmig. Probabilistic (logic) programming concepts. 100(1):5–47, 2015.

[29] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. ProbLog: A probabilistic Prolog and its application in link discovery. volume 7, pages 2462–2467, 2007.

[30] Luc De Raedt, Robin Manhaeve, Sebastijan Dumancic, Thomas Demeester, and Angelika Kimmig. Neuro-symbolic= neural+ logical+ probabilistic. In *NeSy'19@ IJCAI, the 14th International Workshop on Neural-Symbolic Learning and Reasoning*, pages 1–4, 2019.

[31] Luc De Raedt and Ingo Thon. Probabilistic rule learning. volume 6489, pages 47–58, 2011.

[32] Arthur P Dempster, Nan M Laird, and Donald B Rubin. Maximum likelihood from incomplete data via the EM algorithm. 39(1):1–38, 1977.

[33] Nicola Di Mauro, Elena Bellodi, and Fabrizio Riguzzi. Bandit-based Monte-Carlo structure learning of probabilistic logic programs. 100(1):127–156, 2015.

[34] Saso Džeroski. Handling imperfect data in inductive logic programming. pages 111–125, 1993.

[35] Tom Fawcett. An introduction to ROC analysis. 27:861–874, 2006.

[36] Daan Fierens, Guy Van den Broeck, Joris Renkens, Dimitar Sht. Shterionov, Bernd Gutmann, Ingo Thon, Gerda Janssens, and Luc De Raedt. Inference and learning in probabilistic logic programs using weighted Boolean formulas. 15(3):358–401, 2015.

[37] Tiago Gomes and Vítor Santos Costa. Evaluating inference algorithms for the Prolog factor language. volume 7842, pages 74–85, 2012.

[38] Irving John Good. A causal calculus (i). *The British journal for the philosophy of science*, 11(44):305–318, 1961.

[39] Daniel M. Gordon. A survey of fast exponentiation methods. 27(1):129 – 146, 1998.

[40] Andrey Gorlin, C. R. Ramakrishnan, and Scott A. Smolka. Model checking with probabilistic tabled logic programming. 12(4-5):681–700, 2012.

[41] B. Gutmann, A. Kimmig, K. Kersting, and L. De Raedt. Parameter learning in probabilistic databases: A least squares approach. volume 5211, pages 473–488, 2008.

[42] Bernd Gutmann, Ingo Thon, Angelika Kimmig, Maurice Bruynooghe, and Luc De Raedt. The magic of logical inference in probabilistic programming. 11(4-5):663–680, 2011.

[43] Petr Hájek. *Metamathematics of fuzzy logic*, volume 4. 1998.

[44] Simon Haykin. *Neural networks: a comprehensive foundation.* Prentice Hall PTR, 1994.

[45] Jacques Herbrand. *Recherches sur la théorie de la démonstration.* PhD thesis, Université de Paris, 1930.

[46] Holger H. Hoos and Thomas Stützle. *Stochastic Local Search: Foundations & Applications.* Elsevier / Morgan Kaufmann, 2004.

[47] Joe Hurd. A formal approach to probabilistic termination. volume 2410, pages 230–245, 2002.

[48] Tuyen N. Huynh and Raymond J. Mooney. Discriminative structure and parameter learning for markov logic networks. In William W. Cohen, Andrew McCallum, and Sam T. Roweis, editors, *Proceedings of the 25th international conference on Machine learning*, pages 416–423. ACM, 2008.

[49] Tuyen N. Huynh and Raymond J. Mooney. Online structure learning for markov logic networks. volume 6912, pages 81–96, 2011.

[50] Muhammad Asiful Islam, CR Ramakrishnan, and IV Ramakrishnan. Inference in probabilistic logic programs with continuous random variables. 12:505–523, 2012.

[51] Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. Weakest precondition reasoning for expected run-times of probabilistic programs. volume 9632, pages 364–389, 2016.

[52] Seyed Mehran Kazemi, David Buchman, Kristian Kersting, Sriraam Natarajan, and David Poole. Relational logistic regression. AAAI Press, 2014.

[53] Seyed Mehran Kazemi and David Poole. Relnn: A deep neural model for relational learning. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[54] K. Kersting and L. De Raedt. Basic principles of learning Bayesian logic programs. In *Probabilistic Inductive Logic Programming*, volume 4911 of *LNCS*, pages 189–221. Springer, 2008.

[55] Kristian Kersting and Luc De Raedt. Towards combining inductive logic programming with bayesian networks. pages 118–131, 2001.

[56] Kristian Kersting and Luc De Raedt. Basic principles of learning bayesian logic programs. In *Institute for Computer Science, University of Freiburg*. Citeseer, 2002.

[57] Tushar Khot, Sriraam Natarajan, Kristian Kersting, and Jude W. Shavlik. Learning Markov Logic Networks via functional gradient boosting. In *Proceedings of the 11th IEEE International Conference on Data Mining*, pages 320–329. IEEE, 2011.

[58] Jörg-Uwe Kietz and Marcus Lübbe. An efficient subsumption algorithm for inductive logic programming. pages 130–138. Morgan Kaufmann, 1994.

[59] D. Marc Kilgour and Steven J. Brams. The truel. *Mathematics Magazine*, 70(5):315–326, 1997.

[60] Angelika Kimmig. *A Probabilistic Prolog and its Applications*. PhD thesis, Katholieke Universiteit Leuven, Belgium, 2010.

[61] Angelika Kimmig, Bart Demoen, Luc De Raedt, Vitor Santos Costa, and Ricardo Rocha. On the implementation of the probabilistic logic programming language ProbLog. 11(2-3):235–262, 2011.

[62] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[63] Jacek Kisynski and David Poole. Lifted aggregation in directed first-order probabilistic models. pages 1922–1929, 2009.

[64] Stanley Kok and Pedro Domingos. Learning the structure of Markov Logic Networks. pages 441–448. ACM, 2005.

[65] Stanley Kok and Pedro Domingos. Learning Markov Logic Networks using structural motifs. pages 551–558. Omnipress, 2010.

[66] Daphne Koller and Nir Friedman. *Probabilistic Graphical Models: Principles and Techniques*. Adaptive computation and machine learning. MIT Press, Cambridge, MA, 2009.

[67] Daphne Koller and Avi Pfeffer. Learning probabilities for noisy first-order rules. In *IJCAI*, pages 1316–1323, 1997.

[68] A. N. Kolmogorov. *Foundations of the Theory of Probability*. Chelsea Publishing Company, New York, 1950.

[69] Robert A. Kowalski. Predicate logic as programming language. In *IFIP Congress*, pages 569–574, 1974.

[70] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[71] Haifeng Li, Keshu Zhang, and Tao Jiang. The regularized em algorithm. In *AAAI*, pages 807–812, 2005.

[72] Robin Manhaeve, Sebastijan Dumancic, Angelika Kimmig, Thomas Demeester, and Luc De Raedt. Deepproblog: Neural probabilistic logic programming. In *Advances in Neural Information Processing Systems*, pages 3749–3759, 2018.

[73] Wolfgang May. Information extraction and integration: The mondial case study. Technical report, Universitat Freiburg, Institut für Informatik, 1999.

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

[75] W. Meert, J. Struyf, and H. Blockeel. CP-Logic theory inference with contextual variable elimination and comparison to BDD based inference methods. volume 5989, pages 96–109, 2010.

[76] Lilyana Mihalkova and Raymond J. Mooney. Bottom-up learning of Markov logic network structure. In *Proceedings of the 24th International Conference on Machine Learning*, pages 625–632. ACM, 2007.

[77] Tom M. Mitchell. *Machine learning*. McGraw Hill series in computer science. McGraw-Hill, 1997.

[78] Søren Mørk and Ian Holmes. Evaluating bacterial gene-finding hmm structures as probabilistic logic programs. *Bioinformatics*, 28(5):636–642, 2012.

[79] Stephen Muggleton. Inverse entailment and Progol. 13:245–286, 1995.

[80] Stephen Muggleton. Learning stochastic logic programs. 4(B):141–153, 2000.

[81] Stephen Muggleton and Luc De Raedt. Inductive logic programming: Theory and methods. 19:629–679, 1994.

[82] Arun Nampally and CR Ramakrishnan. Adaptive MCMC-based inference in probabilistic logic programs. *arXiv preprint arXiv:1403.6036*, 2014.

[83] 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*, 86(1):25–56, 2012.

[84] Sriraam Natarajan, Prasad Tadepalli, Gautam Kunapuli, and Jude Shavlik. Learning parameters for relational probabilistic models with noisy-or combining rule. In *Machine Learning and Applications, 2009. ICMLA'09. International Conference on*, pages 141–146. IEEE, 2009.

[85] Aniruddh Nath and Pedro M. Domingos. Learning relational sum-product networks. pages 2878–2886, 2015.

[86] Liem Ngo and Peter Haddawy. Answering queries from context-sensitive probabilistic knowledge bases. 171(1):147–177, 1997.

[87] Arnaud Nguembang Fadja and Fabrizio Riguzzi. Probabilistic logic programming in action. In Andreas Holzinger, Randy Goebel, Massimo Ferri, and Vasile Palade, editors, *Towards Integrative Machine Learning and Knowledge Extraction*, volume 10344. 2017.

[88] Arnaud Nguembang Fadja and Fabrizio Riguzzi. Lifted discriminative learning of probabilistic logic programs. *Machine Learning*, 108(7):1111–1135, 2019.

[89] Arnaud Nguembang Fadja, Fabrizio Riguzzi, and Evelina Lamma. Expectation maximization in deep probabilistic logic programming. In *International Conference of the Italian Association for Artificial Intelligence*, pages 293–306. Springer, 2018.

[90] Masaaki Nishino, Akihiro Yamamoto, and Masaaki Nagata. A sparse parameter learning method for probabilistic logic programs. In *Statistical Relational Artificial Intelligence, Papers from the 2014 AAAI Workshop*, volume WS-14-13 of *AAAI Workshops*, 2014.

[91] Davide Nitti, Tinne De Laet, and Luc De Raedt. Probabilistic logic programming for hybrid relational domains. 103(3):407–449, 2016.

[92] Jorge Nocedal. Updating Quasi-Newton matrices with limited storage. 35(151):773–782, 1980.

[93] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufmann, 1988.

[94] Y. Perov, B. Paige, and F. Wood. The Indian GPA problem, 2017. `https://bitbucket.org/probprog/anglican-examples/src/master/worksheets/indian-gpa.clj`, accessed June 1, 2018.

[95] Avi Pfeffer. *Practical Probabilistic Programming.* Manning Publications, 2016.

[96] Gordon D. Plotkin. A note on inductive generalization. In *Machine Intelligence*, volume 5, pages 153–163. Edinburgh University Press, 1970.

[97] D. Poole. The Independent Choice Logic for modelling multiple agents under uncertainty. 94:7–56, 1997.

[98] D. Poole. Abducing through negation as failure: Stable models within the independent choice logic. 44(1-3):5–35, 2000.

[99] David Poole. Probabilistic Horn abduction and Bayesian networks. 64(1):81–129, 1993.

[100] David Poole. First-order probabilistic inference. In *IJCAI*, volume 3, pages 985–991, 2003.

[101] Hoifung Poon and Pedro M. Domingos. Sum-product networks: A new deep architecture. pages 337–346. AUAI Press, 2011.

[102] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural networks*, 12(1):145–151, 1999.

[103] J. Ross Quinlan. Learning logical definitions from relations. 5:239–266, 1990.

[104] J. Ross Quinlan and R. Mike Cameron-Jones. Induction of logic programs: Foil and related systems. *New Generation Computing*, 13(3-4):287–312, 1995.

[105] Luc De Raedt, Anton Dries, Ingo Thon, Guy Van den Broeck, and Mathias Verbeke. Inducing probabilistic relational rules from probabilistic examples. pages 1835–1843, 2015.

[106] Peter Reutemann, Bernhard Pfahringer, and Eibe Frank. A toolbox for learning from relational data with propositional and multi-instance learners. volume 3339, pages 1017–1023, 2004.

[107] Matthew Richardson and Pedro Domingos. Markov logic networks. 62(1-2):107–136, 2006.

[108] Fabrizio Riguzzi. Extended semantics and inference for the independent choice logic. *Logic Journal of the IGPL*, 17(6):589–629, 2009.

[109] Fabrizio Riguzzi. MCINTYRE: A Monte Carlo system for probabilistic logic programming. 124(4):521–541, 2013.

[110] Fabrizio Riguzzi. Speeding up inference for probabilistic logic programs. 57(3):347–363, 2014.

[111] Fabrizio Riguzzi. The distribution semantics for normal programs with function symbols. 77:1–19, 2016.

[112] Fabrizio Riguzzi, Elena Bellodi, Riccardo Zese, Giuseppe Cota, and Evelina Lamma. Scaling structure learning of probabilistic logic programs by mapreduce. In *Proceedings of the Twenty-second European Conference on Artificial Intelligence*, pages 1602–1603. IOS Press, 2016.

[113] Fabrizio Riguzzi, Elena Bellodi, Riccardo Zese, Giuseppe Cota, and Evelina Lamma. A survey of lifted inference approaches for probabilistic logic programming under the distribution semantics. 80:313–333, 1 2017.

[114] Fabrizio Riguzzi and Nicola Di Mauro. Applying the information bottleneck to statistical relational learning. 86(1):89–114, 2012.

[115] Fabrizio Riguzzi, Evelina Lamma, Marco Alberti, Elena Bellodi, Riccardo Zese, and Giuseppe Cota. Probabilistic logic programming for natural language processing. In Federico Chesani, Paola Mello, and Michela Milano, editors, *Workshop on Deep Understanding and Reasoning, URANIA 2016*, volume 1802 of *CEUR Workshop Proceedings*, pages 30–37. Sun SITE Central Europe, 2017.

[116] Fabrizio Riguzzi and Terrance Swift. The PITA system: Tabling and answer subsumption for reasoning under uncertainty. 11(4–5):433–449, 2011.

[117] Fabrizio Riguzzi and Terrance Swift. Well-definedness and efficient inference for probabilistic logic programming under the distribution semantics. 13(2):279–302, 2013.

[118] Raul Rojas. The backpropagation algorithm. In *Neural networks*, pages 149–182. Springer, 1996.

[119] Tian Sang, Paul Beame, and Henry A. Kautz. Performing bayesian inference by weighted model counting. In *20th National Conference on Artificial Intelligence*, pages 475–482, Palo Alto, California USA, 2005.

[120] T. Sato and P. Meyer. Infinite probability computation by cyclic explanation graphs. 14:909–937, 11 2014.

[121] Taisuke Sato. A statistical learning method for logic programs with distribution semantics. pages 715–729, 1995.

[122] Taisuke Sato. A glimpse of symbolic-statistical modeling by PRISM. *Journal of Intelligent Information Systems*, 31(2):161–176, 2008.

[123] Taisuke Sato and Yoshitaka Kameya. PRISM: a language for symbolic-statistical modeling. volume 97, pages 1330–1339, 1997.

[124] Taisuke Sato and Yoshitaka Kameya. Parameter learning of logic programs for symbolic-statistical modeling. 15:391–454, 2001.

[125] Taisuke Sato and Keiichi Kubota. Viterbi training in PRISM. 15(02):147–168, 2015.

[126] Taisuke Sato and Philipp Meyer. Tabling for infinite probability computation. volume 17 of *LIPIcs*, pages 348–358, 2012.

[127] Oliver Schulte and Hassan Khosravi. Learning graphical models for relational data via lattice search. 88(3):331–368, 2012.

[128] Gustav Sourek, Vojtech Aschenbrenner, Filip Zelezný, and Ondrej Kuzelka. Lifted relational neural networks. volume 1583 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2016.

[129] Ashwin Srinivasan. The aleph manual, 2007. `http://www.cs.ox.ac.uk/activities/machlearn/Aleph/aleph.html`, accessed April 3, 2018.

[130] Ashwin Srinivasan, Ross D. King, Stephen Muggleton, and Michael J. E. Sternberg. Carcinogenesis predictions using ILP. volume 1297, pages 273–287, 1997.

[131] Ashwin Srinivasan, Stephen Muggleton, Michael J. E. Sternberg, and Ross D. King. Theories for mutagenicity: A study in first-order and feature-based induction. 85(1-2):277–299, 1996.

[132] S.M. Srivastava. *A Course on Borel Sets*. Graduate Texts in Mathematics. Springer, 2013.

[133] Jan Struyf, Jesse Davis, and David Page. An efficient approximation to lookahead in relational learners. pages 775–782, 2006.

[134] Terrance Swift and David Scott Warren. XSB: Extending prolog with tabled logic programming. 12(1-2):157–187, 2012.

[135] Nima Taghipour, Daan Fierens, Jesse Davis, and Hendrik Blockeel. Lifted variable elimination: Decoupling the operators from the constraint language. 47:393–439, 2013.

[136] Tijmen Tieleman and Geoffery Hinton. Rmsprop gradient optimization. *URL http://www. cs. toronto. edu/tijmen/csc321/slides/lecture_ slides_ lec6. pdf*, 2014.

[137] Dustin Tran, Matthew D. Hoffman, Rif A. Saurous, Eugene Brevdo, Kevin Murphy, and David M. Blei. Deep probabilistic programming. 2017.

[138] Leslie G Valiant. The complexity of enumeration and reliability problems. 8(3):410–421, 1979.

[139] Guy Van den Broeck, Wannes Meert, and Adnan Darwiche. Skolemization for weighted first-order model counting. pages 111–120. AAAI Press, 2014.

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

[141] Jan Van Haaren, Guy Van den Broeck, Wannes Meert, and Jesse Davis. Lifted generative learning of markov logic networks. 103(1):27–55, 2016.

[142] J. Vennekens and S. Verbaeten. Logic programs with annotated disjunctions. Technical Report CW386, KU Leuven, 2003.

[143] J. Vennekens, S. Verbaeten, and M. Bruynooghe. Logic programs with annotated disjunctions. volume 3131, pages 195–209, Berlin, 2004.

[144] J. Vennekens, S. Verbaeten, and M. Bruynooghe. Logic Programs With Annotated Disjunctions. volume 3132, pages 431–445, 2004.

[145] Antonio Vergari, Nicola Di Mauro, and Floriana Esposito. Visualizing and understanding sum-product networks. *Machine Learning*, 108(4):551–573, 2019.

[146] Richard Von Mises. *Probability, statistics, and truth.* Courier Corporation, 1981.

[147] John Von Neumann. Various techniques used in connection with random digits. *Nattional Bureau of Standard (U.S.), Applied Mathematics Series*, 12:36–38, 1951.

[148] William Yang Wang, Kathryn Mazaitis, and William W. Cohen. Structure learning via parameter learning. pages 1199–1208, 2014.

[149] Michael P Wellman, John S Breese, and Robert P Goldman. From knowledge bases to decision models. 7(1):35–53, 1992.

[150] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. Swi-prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012.

[151] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. SWI-Prolog. 12(1-2):67–96, 2012.

[152] Filip Železný, Ashwin Srinivasan, and C. David Page. Lattice-search runtime distributions may be heavy-tailed. In *Proceedings of the 12th International Conference on Inductive Logic Programming*, 2002.

[153] Filip Železný, Ashwin Srinivasan, and C. David Page Jr. Randomised restarted search in ILP. *Machine Learning*, 64(1-3):183–208, 2006.

[154] Nevin L Zhang and David Poole. A simple approach to bayesian network computations. pages 171–178, 1994.

[155] Nevin Lianwen Zhang and David L. Poole. Exploiting causal independence in Bayesian network inference. 5:301–328, 1996.

Appendix

Appendix A

Proofs of theorems

This appendix presents detail proofs of Theorems 1 and 2.

Theorem A.1. *The L_1 regularized objective function:*

$$J_1(\theta) = N_1 \log \theta + N_0 \log(1-\theta) - \gamma\theta \tag{A.1}$$

is maximum in

$$\theta_2 = \frac{4N_1}{2(\gamma + N_0 + N_1 + \sqrt{(N_0+N_1)^2 + \gamma^2 + 2\gamma(N_0 - N_1)})}$$

Proof. Deriving J_1 w.r.t. θ, we obtain

$$J_1(\theta) = \frac{N_1}{\theta} - \frac{N_0}{1-\theta} - \gamma \tag{A.2}$$

Solving $J_1' = 0$ we have:

$$\begin{aligned} \frac{N_1}{\theta} - \frac{N_0}{1-\theta} - \gamma &= 0 \\ N_1(1-\theta) - N_0\theta - \gamma\theta(1-\theta) &= 0 \\ N_1 - N_1\theta - N_0\theta - \gamma\theta + \gamma\theta^2 &= 0 \\ \gamma\theta^2 - (N_0 + N_1 + \gamma)\theta + N_1 &= 0 \end{aligned} \tag{A.3}$$

Equation A.3 is a second degree equation whose solutions are

$$\theta = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

with $a = \gamma$, $b = -N_0 - N_1 - \gamma$ and $c = N_1$ The determinant is

$$\Delta = b^2 - 4ac = (N_0+N_1)^2+\gamma^2+2(N_0+N_1)\gamma-4\gamma N_1 = (N_0+N_1)^2+\gamma^2-2\gamma N_1+2\gamma N_0$$

To see whether the determinant is positive we must solve the equation

$$\gamma^2 + (-2N_1 + 2N_0)\gamma + (N_0 + N_1)^2 = 0$$

which gives

$$\begin{aligned}\gamma &= \frac{2N_1 - 2N_0 \pm \sqrt{(2N_1 - 2N_0)^2 - 4(N_0 + N_1)^2}}{2} = \\ &\frac{2N_1 - 2N_0 \pm \sqrt{4N_1^2 + 4N_0^2 - 8N_0N_1 - 4N_0^2 - 4N_1^2 - 8N_0N_1}}{2} = \\ &\frac{6N_1 + 2N_0 \pm \sqrt{-16N_0N_1}}{2}\end{aligned}$$

Therefore there is no real value for γ for which Δ is 0, so Δ is always greater or equal to 0 because for $N_0 = N_1$ we have $\Delta = 4N_0^2 + \gamma^2$ which is greater or equal to 0. This means that $J_1' = 0$ has two real solutions.

Observe that $\lim_{\theta\to 0+} J_1(\theta) = \lim_{\theta\to 1-} J_1(\theta) = -\infty$. Therefore J_1 must have at least a maximum in $(0, 1)$. Since in such a maximum the first derivative must be 0 and J_1' has two zeros, then J_1 has a single maximum in $(0, 1)$.

Let us compute the two zeros of J_1':

$$\begin{aligned}
\theta &= \frac{\gamma + N_0 + N_1 \pm \sqrt{(N_0+N_1)^2 + \gamma^2 + 2\gamma(N_0 - N_1)}}{2\gamma} \\
\theta_1 &= \frac{\gamma + N_0 + N_1 - \sqrt{(N_0+N_1)^2 + \gamma^2 + 2\gamma(N_0 - N_1)}}{2\gamma} = \\
& \frac{(\gamma + N_0 + N_1)^2 - (N_0+N_1)^2 - \gamma^2 - 2\gamma(N_0 - N_1)}{2\gamma(\gamma + N_0 + N_1 + \sqrt{(N_0+N_1)^2 + \gamma^2 + 2\gamma(N_0 - N_1)})} = \\
& \frac{2\gamma(N_0+N_1) - 2\gamma(N_0 - N_1)}{2\gamma(\gamma + N_0 + N_1 + \sqrt{(N_0+N_1)^2 + \gamma^2 + 2\gamma(N_0 - N_1)})} = \\
& \frac{4\gamma N_1}{2\gamma(\gamma + N_0 + N_1 + \sqrt{(N_0+N_1)^2 + \gamma^2 + 2\gamma(N_0 - N_1)})} = \\
& \frac{4N_1}{2(\gamma + N_0 + N_1 + \sqrt{(N_0+N_1)^2 + \gamma^2 + 2\gamma(N_0 - N_1)})} \\
\theta_2 &= \frac{\gamma + N_0 + N_1 + \sqrt{(N_0+N_1)^2 + \gamma^2 + 2\gamma(N_0 - N_1)}}{2\gamma} = \\
& \frac{\gamma + N_0 + N_1 + \sqrt{N_0^2 + N_1^2 + 2N_0N_1 + \gamma^2 + 2\gamma N_0 - 2\gamma N_1}}{2\gamma} = \\
& \frac{\gamma + N_0 + N_1 + \sqrt{N_0^2 + N_1^2 + 2N_1(N_0 - \gamma) + \gamma^2 + 2\gamma N_0}}{2\gamma} = \\
& \frac{\gamma + N_0 + N_1 + \sqrt{(N_1 - \gamma)^2 + N_0^2 + 2N_0N_1 + 2\gamma N_0}}{2\gamma}
\end{aligned}$$

We can see that $\theta_1 \geq 0$ and

$$\theta_1 \leq \frac{4N_1}{2N_0 + 2N_1 + 2N_0 + 2N_1} = \frac{N_1}{N_0 + N_1} \leq 1$$

So θ_1 is the root of J_1' that we are looking for. □

Note that, as expected:

$$\lim_{\gamma \to 0} \theta_1 = \frac{4N_1}{2N_0 + 2N_1 + 2N_0 + 2N_1} = \frac{N_1}{N_0 + N_1}$$

Theorem A.2. *The L_2 regularized objective function:*

$$J_2(\theta) = N_1 \log\theta + N_0 \log(1-\theta) - \frac{\gamma}{2}\theta^2 \tag{A.4}$$

is maximum in

$$\theta = \frac{2\sqrt{\frac{3N_0+3N_1+\gamma}{\gamma}}\cos\left(\frac{\arccos\left(\frac{\sqrt{\frac{\gamma}{3N_0+3N_1+\gamma}}\left(\frac{9N_0}{2}-9N_1+\gamma\right)}{3N_0+3N_1+\gamma}\right)}{3}-\frac{2\pi}{3}\right)}{3}+\frac{1}{3}$$

Proof. We want to find the value of θ that maximizes $J_2(\theta)$ as a function of N_1, N_0 and γ, with $N_1 \geq 0, N_0 \geq 0$ and $\gamma \geq 0$.

The derivative of $J_2(\theta)$ is

$$J_2'(\theta) = \frac{N_1}{\theta} - \frac{N_0}{1-\theta} - \gamma\theta \tag{A.5}$$

Solving $J_2'(\theta) = 0$ we have:

$$\begin{aligned}
\frac{N_1}{\theta} - \frac{N_0}{1-\theta} - \gamma\theta &= 0 \\
N_1(1-\theta) - N_0\theta - \gamma\theta^2(1-\theta) &= 0 \\
N_1 - N_1\theta - N_0\theta - \gamma\theta^2 + \gamma\theta^3 &= 0 \\
\gamma\theta^3 - \gamma\theta^2 - (N_0+N_1)\theta + N_1 &= 0 \\
\theta^3 - \theta^2 - \frac{(N_0+N_1)}{\gamma}\theta + \frac{N_1}{\gamma} &= 0 \qquad \text{(A.6)}
\end{aligned}$$

Equation A.6 is a third degree equation. Let us consider $a = 1, b = -1, c = -\frac{N_0+N_1}{\gamma}, d = \frac{N_1}{\gamma}$. We want to solve the equation

$$a\theta^3 + b\theta^2 + c\theta + d = 0 \tag{A.7}$$

The number of real and complex roots is determined by the discriminant of the cubic equation [17]:

$$\Delta = 18abcd - 4b^3d + b^2c^2 - 4ac^3 - 27a^2d^2$$

In our case we have

$$\begin{aligned}\Delta \;=\;& -\frac{27N_1^2}{\gamma^2}+\frac{4N_1}{\gamma}-\frac{18N_1\left(-N_0-N_1\right)}{\gamma^2}+\frac{\left(-N_0-N_1\right)^2}{\gamma^2}-\frac{4\left(-N_0-N_1\right)^3}{\gamma^3}=\\ & \frac{4N_0^3}{\gamma^3}+\frac{12N_0^2N_1}{\gamma^3}+\frac{N_0^2}{\gamma^2}+\frac{12N_0N_1^2}{\gamma^3}+\frac{20N_0N_1}{\gamma^2}+\frac{4N_1^3}{\gamma^3}-\frac{8N_1^2}{\gamma^2}+\frac{4N_1}{\gamma}\end{aligned}$$

If $\Delta > 0$ the equation has three distinct real roots. If $\gamma \leq N_0 + N_1$

$$\begin{aligned}\Delta \geq\;& \tfrac{4N_0^3}{(N_0+N_1)^3}+\tfrac{12N_0^2N_1}{(N_0+N_1)^3}+\tfrac{N_0^2}{(N_0+N_1)^2}+\tfrac{12N_0N_1^2}{(N_0+N_1)^3}+\\ & \tfrac{20N_0N_1}{(N_0+N_1)^2}+\tfrac{4N_1^3}{(N_0+N_1)^3}-\tfrac{8N_1^2}{(N_0+N_1)^2}+\tfrac{4N_1}{(N_0+N_1)}=\\ & \tfrac{1}{(N_0+N_1)^3}(4N_0^3+12N_0^2N_1+N_0^2(N_0+N_1)+12N_0N_1^2+\\ & 20N_0N_1(N_0+N_1)+4N_1^3-8N_1^2(N_0+N_1)+4N_1(N_0+N_1)^2)=\\ & \tfrac{N_0(5N_0+32N_1)}{N_0^2+2N_0N_1+N_1^2}\geq\\ & 0\end{aligned}$$

If $\gamma \geq N_0 + N_1$

$$\begin{aligned}\Delta =\;& \tfrac{1}{\gamma^3}\left(4N_1\gamma^2+\gamma\left(-27N_1^2+18N_1\left(N_0+N_1\right)+\left(N_0+N_1\right)^2\right)+\right.\\ & \left.4\left(N_0+N_1\right)^3\right)\geq\\ & \tfrac{1}{\gamma^3}\left(4N_1\left(N_0+N_1\right)^2+4\left(N_0+N_1\right)^3+\right.\\ & \left.\left(N_0+N_1\right)\left(-27N_1^2+18N_1\left(N_0+N_1\right)+\left(N_0+N_1\right)^2\right)\right)=\\ & \tfrac{1}{\gamma^3}N_0\left(5N_0^2+37N_0N_1+32N_1^2\right)\geq\\ & 0\end{aligned}$$

So equation A.6 has 3 real roots that can be computed as [17]:

$$\theta_k = 2\sqrt{-\frac{p}{3}}\cos\left(\frac{1}{3}\arccos\left(\frac{3q}{2p}\sqrt{\frac{-3}{p}}\right)-\frac{2\pi k}{3}\right)+\frac{1}{3}\quad \text{for}\quad k=0,1,2$$

where

$$\begin{aligned}p =&\frac{3ac-b^2}{3a^2},\\ q =&\frac{2b^3-9abc+27a^2d}{27a^3}.\end{aligned}$$

θ_1 is

$$\theta_1 = \frac{2\sqrt{\frac{3N_0+3N_1+\gamma}{\gamma}}\cos\left(\frac{\arccos\left(\frac{\sqrt{\frac{\gamma}{3N_0+3N_1+\gamma}}\left(\frac{9N_0}{2}-9N_1+\gamma\right)}{3N_0+3N_1+\gamma}\right)}{3}-\frac{2\pi}{3}\right)}{3}+\frac{1}{3} \quad \text{(A.8)}$$

Since $\lim_{\theta\to0^+} J_2(\theta) = \lim_{\theta\to1^-} J_2(\theta) = -\infty$, then J_2 must have at least a maximum in $(0,1)$. In such a maximum the first derivative must be 0. J_2' has three zeros. Therefore there are only two possibilities: either J_2 has a single maximum in $(0,1)$ or it has two (with a minimum in between).

Note that

$$\begin{aligned} J_2(\theta) &= N_1\log\theta + N_0\log(1-\theta) - \frac{\gamma}{2}\theta^2 \\ &= \gamma\left\{\frac{N_1}{\gamma}\log\theta + \frac{N_0}{\gamma}\log(1-\theta) - \frac{\theta^2}{2}\right\} \\ &= \gamma\left\{A\log\theta + B\log(1-\theta) - \frac{\theta^2}{2}\right\}. \end{aligned}$$

Since we are interested only in maxima and minima, the γ factor can be ignored. Now $J_2' = 0$ if and only if $P(\theta) = \theta^3 - \theta^2 - (A+B)\theta + A = 0$. Since $P(0) = A > 0$ and $P(1) = -B < 0$, then P is zero at least once in $(0,1)$. The first derivative of P is

$$P'(\theta) = \theta^2 - 2\theta - (a+b).$$

$P'(\theta) = 0$ has two roots, one negative and the other larger than 1. So P' is decreasing in $(0,1)$ and therefore P has a single zero. So J_2' has a single zero that is a maximum.

Let us now prove that it is θ_1 of Equation A.8. We show that θ_1 is in $[0,1]$ for a specific value of γ. Since the fact that J_2' has a single zero that is a maximum doesn't depend on the values of γ, this means that θ_1 is the maximum we are looking for.

Let us choose $\gamma = N_0 + N_1$:

$$\begin{aligned}
\theta_1 &= \frac{2\sqrt{\frac{3N_0+3N_1+N_0+N_1}{N_0+N_1}}\cos\left(\frac{\arccos\left(\frac{\sqrt{\frac{N_0+N_1)}{3N_0+3N_1+N_0+N_1}}\left(\frac{9N_0}{2}-9N_1+N_0+N_1\right)}{3N_0+3N_1+N_0+N_1}\right)}{3}-\frac{2\pi}{3}\right)}{3}+\frac{1}{3}= \\
&\frac{2\sqrt{4}\cos\left(\frac{\arccos\left(\frac{\sqrt{\frac{N_0+N_1)}{4(N_0+N_1)}}\left(\frac{11N_0}{2}-8N_1\right)}{4(N_0+N_1)}\right)}{3}-\frac{2\pi}{3}\right)}{3}+\frac{1}{3}= \\
&\frac{4\cos\left(\frac{\arccos\left(\frac{\frac{1}{2}\left(\frac{11N_0}{2}-8N_1\right)}{4(N_0+N_1)}\right)}{3}-\frac{2\pi}{3}\right)}{3}+\frac{1}{3}= \\
&\frac{4\cos\left(\frac{\arccos\left(\frac{\left(\frac{11N_0}{2}-8N_1\right)}{8(N_0+N_1)}\right)}{3}-\frac{2\pi}{3}\right)}{3}+\frac{1}{3}= \\
&\frac{4\cos\left(\frac{\arccos\left(\frac{(11N_0-16N_1)}{16(N_0+N_1)}\right)}{3}-\frac{2\pi}{3}\right)}{3}+\frac{1}{3}
\end{aligned}$$

θ_1 is minimal when N_1 is 0 and we get

$$\begin{aligned}
\theta_1 &\geq \frac{4\cos\left(\frac{\arccos\left(\frac{11}{16}\right)}{3}-\frac{2\pi}{3}\right)}{3}+\frac{1}{3}= \\
&\frac{4\cos\left(\frac{\arccos 0.812755561368661}{3}-\frac{2\pi}{3}\right)}{3}+\frac{1}{3}= \\
&-\frac{4\cdot 0.25}{3}+\frac{1}{3}= \\
&-\frac{1}{3}+\frac{1}{3}=0
\end{aligned}$$

θ_1 is maximal when N_0 is 0 and we get

$$\begin{aligned}\theta_1 \;\leq\; & \frac{4\cos\left(\frac{\arccos\left(\frac{(-16N_1)}{16N_1}\right)}{3} - \frac{2\pi}{3}\right)}{3} + \frac{1}{3} = \\ & \frac{4\cos\left(\frac{\arccos(-1)}{3} - \frac{2\pi}{3}\right)}{3} + \frac{1}{3} = \\ & \frac{4\cos\left(\frac{\pi}{3} - \frac{2\pi}{3}\right)}{3} + \frac{1}{3} = \\ & \frac{4\cos -\frac{\pi}{3}}{3} + \frac{1}{3} = \\ & \frac{4}{3\cdot 2} + \frac{1}{3} = \\ & \frac{2}{3} + \frac{1}{3} = 1\end{aligned}$$

□

Note that, as expected, $\lim_{\gamma\to 0}\theta_1 = \frac{N_1}{N_0+N_1}$, i.e., with $\gamma = 0$ we get the formula for the case of no regularization. In fact, consider the Maclaurin expansion of $\arccos(z)$:

$$\arccos(z) = \frac{\pi}{2} - z - \left(\frac{1}{2}\right)\frac{z^3}{3} - \left(\frac{1\cdot 3}{2\cdot 4}\right)\frac{z^5}{5} - \left(\frac{1\cdot 3\cdot 5}{2\cdot 4\cdot 6}\right)\frac{z^7}{7} - \cdots = \frac{\pi}{2} - z - O(z^3)$$

so

$$\lim_{\gamma\to 0}\arccos\left(\frac{\sqrt{\frac{\gamma}{3N_0+3N_1+\gamma}}\left(\frac{9N_0}{2} - 9N_1 + \gamma\right)}{3N_0+3N_1+\gamma}\right) = \tag{A.9}$$

$$\lim_{\gamma\to 0}\arccos(z) = \tag{A.10}$$

$$\lim_{\gamma\to 0}\frac{\pi}{2} - \sqrt{\frac{\gamma}{(3N_0+3N_1+\gamma)^3}}\left(\frac{9N_0}{2} - 9N_1 + \gamma\right) - O(\gamma^{\frac{3}{2}}) \tag{A.11}$$

Then

$$
\begin{aligned}
\lim_{\gamma\to 0}\cos\left(\frac{\arccos\left(\frac{\sqrt{\frac{\gamma}{3N_0+3N_1+\gamma}}\left(\frac{9N_0}{2}-9N_1+\gamma\right)}{3N_0+3N_1+\gamma}\right)}{3}-\frac{2\pi}{3}\right) &= \quad (A.12)\\
\lim_{\gamma\to 0}\cos\left(\frac{\arccos(z)}{3}-\frac{2\pi}{3}\right) &= \quad (A.13)\\
\lim_{\gamma\to 0}\cos\left(\frac{\pi}{6}-\frac{z}{3}-O(\gamma^{\frac{3}{2}})-\frac{2\pi}{3}\right) &= \quad (A.14)\\
\lim_{\gamma\to 0}\cos\left(-\frac{z}{3}-O(\gamma^{\frac{3}{2}})-\frac{\pi}{2}\right) &= \quad (A.15)\\
\lim_{\gamma\to 0}\cos\left(-\frac{\pi}{2}\right)\cos\left(-\frac{z}{3}-O(\gamma^{\frac{3}{2}})\right)+\sin(-\frac{\pi}{2})\sin\left(\frac{z}{3}-O(\gamma^{\frac{3}{2}}))\right) &= \quad (A.16)\\
\lim_{\gamma\to 0}-\sin\left(\frac{z}{3}-O(\gamma^{\frac{3}{2}}))\right) &= \quad (A.17)\\
& \quad (A.18)
\end{aligned}
$$

Since the Maclaurin expansion of sin is

$$\sin y = y-\frac{y^3}{3!}+\frac{y^5}{5!}-\frac{y^7}{7!}+\cdots$$

then

$$
\begin{aligned}
\lim_{\gamma\to 0}-\sin\left(\frac{z}{3}-O(\gamma^{\frac{3}{2}}))\right) &= \quad (A.19)\\
\lim_{\gamma\to 0}-\frac{z}{3}+O(\gamma^{\frac{3}{2}}) &= \quad (A.20)\\
\lim_{\gamma\to 0}-\frac{1}{3}\sqrt{\frac{\gamma}{(3N_0+3N_1+\gamma)^3}}\left(\frac{9N_0}{2}-9N_1+\gamma\right)+O(\gamma^{\frac{3}{2}}) & \quad (A.21)
\end{aligned}
$$

So

$$\lim_{\gamma\to 0}\theta_1 = \lim_{\gamma\to 0}\frac{2\sqrt{\frac{3N_0+3N_1+\gamma}{\gamma}}\left(-\frac{1}{3}\sqrt{\frac{\gamma}{(3N_0+3N_1+\gamma)^3}}\left(\frac{9N_0}{2}-9N_1+\gamma\right)+O(\gamma^{\frac{3}{2}})\right)}{3}+$$

$$\frac{1}{3}=$$

$$-\lim_{\gamma\to 0}\frac{2\frac{1}{3N_0+3N_1+\gamma}\left(\frac{9N_0}{2}-9N_1+\gamma\right)+2\sqrt{\frac{3N_0+3N_1+\gamma}{\gamma}}O(\gamma^{\frac{3}{2}})}{9}+\frac{1}{3}=$$

$$-\frac{2\frac{\frac{9N_0}{2}-9N_1}{3N_0+3N_1}}{9}+\frac{1}{3}=$$

$$-2\frac{\frac{N_0}{2}-N_1}{3N_0+3N_1}+\frac{1}{3}=$$

$$-\frac{N_0-2N_1}{3N_0+3N_1}+\frac{1}{3}=$$

$$\frac{-N_0+2N_1+N_0+N_1}{3N_0+3N_1}=\frac{3N_1}{3N_0+3N_1}$$

$$=\frac{N_1}{N_0+N_1}$$