Noname manuscript No.
(will be inserted by the editor)

Learning Hierarchical Probabilistic Logic Programs

Arnaud Nguembang Fadja · Fabrizio Riguzzi · Evelina Lamma

Received: date / Accepted: date

Abstract Probabilistic logic programming (PLP) combines logic programs and probabilities. Due to its expressiveness and simplicity, it has been considered as a powerful tool for learning and reasoning in relational domains characterized by uncertainty. Still, learning the parameter and the structure of general PLP is computationally expensive due to the inference cost. We have recently proposed a restriction of the general PLP language called hierarchical PLP (HPLP) in which clauses and predicates are hierarchically organized. HPLPs can be converted into Arithmetic Circuits (ACs) or Deep Neural Networks and inference is much cheaper than for general PLP. In this paper we present algorithms for learning both the parameters and the structure of HPLPs from data. We first present an algorithm, called Parameter learning for HIerarchical probabilistic Logic programs (PHIL) which performs parameter estimation of HPLPs using Gradient Descent and Expectation Maximization. We also propose SLEAHP, Structure LEArning of Hierarchical Probabilistic logic programming, that learns both the structure and the parameters of HPLPs from data. Experiments were performed comparing PHIL and SLEAHP with PLP and Markov Logic Networks state-of-the art systems for parameter and structure learning respectively. PHIL was compared with EMBLEM and ProbLog2 and SLEAHP with SLIPCOVER, PROBFOIL+, MLB-BC, MLN-BT and RDN-B. The experiments on five well known datasets show

Dipartimento di Ingegneria – University of Ferrara
Via Saragat 1, I-44122, Ferrara, Italy
E-mail: arnaud.nguembafadja@unife.it

Dipartimento di Matematica e Informatica – University of Ferrara
Via Saragat 1, I-44122, Ferrara, Italy
E-mail: fabrizio.riguzzi@unife.it

Dipartimento di Ingegneria – University of Ferrara
Via Saragat 1, I-44122, Ferrara, Italy
E-mail: evelina.lamma@unife.it

that our algorithms achieve similar and often better accuracies but in a shorter time.

Keywords Probabilistic Logic Programming · Distribution Semantics · Arithmetic Circuits · Gradient Descent · Back-propagation

1 Introduction

Probabilistic Logic Programs (PLPs) extend Logic Programs (LPs) with probabilities [50]. Due to its expressiveness, PLPs have been, for more than two decades now, used for modeling, learning and reasoning in domains where uncertainty plays a crucial role, such as bioinformatics [36,15], natural language processing [52,40] and link prediction in social networks [35] just to cite a few.

Despite its expressiveness and simplicity, learning the parameters and the structure of general PLPs is still a computationally expensive task due to the high cost of inference often performed many times during the learning process. In [39] we proposed a new language called Hierarchical Probabilistic Logic Programming (HPLP) which is a restriction of the language of Logic Programs with Annotated Disjunctions [64] in which clauses and predicates are hierarchically organized. HPLPs can be translated in an efficient way into Arithmetic Circuits (ACs) from which computing the probability of queries is linear in the number of nodes of the circuit. This makes inference and learning in HPLPs faster than for general PLPs.

In this paper, we present and implement algorithms for learning both the parameters and the structure of HPLPs from data. In order to estimate the parameters of an HPLP, the algorithm, Parameter learning for HIerarchical probabilistic Logic programs (PHIL), first translates the HPLP into a set of Arithmetic circuits (ACs) sharing parameters and then applies gradient descent or Expectation Maximization (EM) over the ACs. The gradients are computed by evaluating the ACs bottom-up and top-down and the expectations by performing message passing over the ACs. Besides, we present an algorithm, called Structure LEArning of Hierarchical Probabilistic logic programming (SLEAHP), that learns both the structure and the parameters of HPLP from data. SLEAHP generates a large HPLP from a bottom clause obtained from a language bias as described in [7] and subsequently applies a regularized version of PHIL on the generated HPLP to cut clauses with a small value of their parameter.

The paper is organized as follows: Section 2 defines general concepts underlying First Order Logic and Logic Programming. Section 3 introduces general notions of PLP and investigates inference in general PLP. Then, Hierarchical Probabilistic Logic Programs are described in Section 4. Sections 5 and 6 present parameter and structure learning respectively. Related work is discussed in Section 7. Experiments are presented in Section 8 and Section 9 concludes the paper.

2 Background

Before describing Probabilistic Logic Programming (PLP) and Hierarchical PLP (HPLP) in the following sections, let us define some basic concepts of First Order Logic (FOL) and Logic Programming (LP). Readers familiar with these concepts can safely skip this section.

2.1 First Order Logic

Given a domain of interest, a *constant* identifies an individual entity in the domain. A *variable* refers to objects in the domain. A *function symbol* (or *functor*) univocally identifies an object of the domain as a function of n other (called arity) objects. A *predicate symbol* is a generic relation (which can be true or false) among n objects.

A *term* is a variable, a constant, or a functor, f, applied to terms, $f(t_1, t_2, \dots, t_n)$. An *atom* is a predicate, p, applied to terms, $p(t_1, t_2, \dots, t_n)$. A *literal* is an atom or its negation. A formula is built from atoms using universal and existential quantifier ($\exists, \forall$) and logical connectives ($\neg, \vee, \wedge, \rightarrow, \leftrightarrow$). A FOL theory is a conjunction of formulas. An expression (literal, term or formula) is *ground* if it does not contain any variable. A *clause* is a disjunction of literals with all variables universally quantified with scope the entire disjunction. A clause with exactly one positive literal is called *definite clause.*

The *Herbrand universe* of a theory T is the set of all ground terms constructed by using the function symbols and constants in T. If the theory does not contain function symbols, the Herbrand universe is finite. The *Herbrand base* of a theory T is the set of all ground atoms obtained from predicates appearing in T and terms of its Herbrand universe. If the theory does not contain function symbols, the Herbrand base is finite as well. A Herbrand interpretation is an assignment of a truth value to all atoms in the Herbrand base. An interpretation is a (Herbrand) model of a theory T, if all the formulas of T are evaluated to true with respect the interpretation.

2.2 Logic Programming

A definite Logic Program (LP), P, is a set of definite clauses represented as

$$h : -b_1, \dots, b_n$$

where the head of the clause, h, is an atom and its body, $b_1, \dots, b_n$, is a conjunction of atoms. A *fact* is a definite clause with empty body and is written h. For definite clauses, Herbrand models have an important property: the intersection of a set of Herbrand models of P is still a Herbrand model of P. The intersection of all Herbrand models is called the Minimal Herbrand Model (MHM) of P. Intuitively, the MHM is the set of all ground atoms that are entailed by the LP.

A Normal LP allows negative literals in the body of clauses. A normal clause is represented as

$$h :- b_1, \ldots, b_n, not\ c_1 \ldots not\ c_m.$$

where $h, b_1, \ldots, b_n, c_1 \ldots c_m$ are atoms. The literals $not\ c_i$ for $i = 1, \ldots, m$ are *default negation iterals* and are different from classical truth-functional logical negative literals $\neg c_i$. There are different semantics for default negation including Clark's completion [9], stable models [20] and well-founded semantics [46, 63]. The well-founded semantics assigns a three-valued model to a program, i.e., it identifies a three-valued interpretation as the meaning of the program. In this paper we use the well-founded semantics for range restricted normal programs. In such programs, variables appearing in the head of a clause must appear in positive literals in its body.

3 Probabilistic Logic Programming

Probabilistic Logic Programming (PLP) combines Logic Programming with probability. PLP under the distribution semantics has been shown expressive enough to represent a wide variety of domains characterized by uncertainty [3, 51,2]. A PLP under the distribution semantics defines a probability distribution over normal logic programs called *instances* or *worlds*. Each world is assumed to have a total well-founded model. The distribution is extended to queries and the probability of a query is computed by marginalizing the joint distribution of the query and the worlds. We consider in this paper a PLP language with a general syntax called *Logic Programs with Annotated Disjunctions* (LPADs) [64].

Programs in LPADs allow alternatives in the head of clauses. Each clause head is a disjunction of atoms annotated with probabilities. Consider a program P with p clauses: $P = \{C_1, \ldots, C_p\}$. Each clause C_i takes the form:

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

where $h_{i1}, \ldots, h_{in_i}$ are logical atoms, $b_{i1}, \ldots, b_{im_i}$ are logical literals and $\pi_{i1}, \ldots, \pi_{in_i}$ are real numbers in the interval $[0, 1]$ that sum up to 1. $b_{i1}, \ldots, b_{im_i}$ is indicated with $body(C_i)$. Note that if $n_i = 1$ the clause corresponds to a non-disjunctive clause. We also allow clauses where $\sum_{k=1}^{n_i} \pi_{ik} < 1$: in this case, the head of the annotated disjunctive clause implicitly contains an extra atom *null* that does not appear in the body of any clause and whose annotation is $1 - \sum_{k=1}^{n_i} \pi_{ik}$. We denote by $ground(P)$ the grounding of an LPAD P, i.e. the replacement of all variables with constants. We consider here programs without function symbols so ground(P) is always finite.

Each grounding $C_i\theta_j$ of a clause C_i corresponds to a random variable X_{ij} with values $\{1, \ldots, n_i\}$ where n_i is the number of head atoms of C_i. The random variables X_{ij} are independent of each other. An *atomic choice* [45] is a triple (C_i, θ_j, k) where $C_i \in P$, θ_j is a substitution that grounds C_i and $k \in$

$\{1, \ldots, n_i\}$ identifies one of the head atoms. In practice (C_i, θ_j, k) corresponds to an assignment $X_{ij} = k$.

A *selection* σ is a set of atomic choices that, for each clause $C_i\theta_j$ in $ground(P)$, contains an atomic choice (C_i, θ_j, k). Let us indicate with S_P the set of all selections. A selection σ identifies a normal logic program l_σ defined as $l_\sigma = \{(h_{ik} :- body(C_i))\theta_j | (C_i, \theta_j, k) \in \sigma\}$. l_σ is called an *instance* or *possible program* of P. Since the random variables associated to ground clauses are independent, we can assign a probability to instances:

$$P(l_\sigma) = \prod_{(C_i,\theta_j,k)\in\sigma} \pi_{ik}$$

We consider only *sound* LPADs where, for each selection σ in S_P, the well-founded model of the possible program l_σ chosen by σ is two-valued. We write $l_\sigma \models q$ to mean that the query q is true in the well-founded model of the possible program l_σ. Since the well-founded model of each possible program is two-valued, q can only be true or false in l_σ.

Let L_P denotes the set of all instances and $P(L_P)$ the distribution over instances. The probability of a query q given an instance l is $P(q|l) = 1$ if $l \models q$ and 0 otherwise. The probability of a query q is given by

$$P(q) = \sum_{l\in L_P} P(q, l) = \sum_{l\in L_P} P(q|l)P(l) = \sum_{l\in L_P : l\models q} P(l) \qquad (1)$$

Example 1 Let us consider the UWCSE domain [28] in which the objective is to predict the "advised_by" relation between students and professors. In this case a program for $advised\_by/2$ may be

$advised\_by(A, B) : 0.3 :-$
$\quad student(A), professor(B), project(C, A), project(C, B).$
$advised\_by(A, B) : 0.6 :-$
$\quad student(A), professor(B), ta(C, A), taught\_by(C, B).$
$student(harry).$
$professor(ben).$
$project(pr_1, harry).project(pr_1, ben).$
$taught\_by(c_1, ben).$
$ta(c_1, harry).$

where $project(C, A)$ means that C is a project with participant A, $ta(C, A)$ means that A is a teaching assistant (TA) 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 on the number of courses the professor teaches where the student is a TA, the higher these numbers the higher the probability.

The facts in the program state that *harry* is a student and *ben* is a professor. They have one joint project and *ben* teaches one course where *harry* is a TA. Suppose we want to compute the probability of $q = advised\_by(harry, ben)$, the first clause has one grounding with head q where the body is true and the second clause has one grounding with head q where the body is true. The possible programs where q is true are those that contain at least one of these ground clauses independently of the presence of other groundings so

$$P(advised\_by(harry, ben)) = 0.3 \cdot 0.6 + 0.3 \cdot 0.4 + 0.7 \cdot 0.6 = 0.72$$

3.1 Inference in PLP

Computing the probabilities of queries by generating all possible programs is infeasible as the number of possible programs is exponential in the number of probabilistic clauses. Therefore much work has been devoted to the development of faster alternatives.

The most common approach for exact inference involves knowledge compilation: the program is converted to a Boolean language from which the computation of the probability of the query is fast.

In order to convert the program to a Boolean language, we must consider *composite choices*: a composite choice κ is a consistent set of atomic choices, i.e., a set where the same ground clause $C_i\theta_j$ appears once.

Given a composite choice κ, we can define the *set of possible programs compatible with* κ L_κ as $L_\kappa = \{l_\sigma | \sigma \in S_P, \kappa \subseteq \sigma\}$. Given a set K of composite choices, L_K is the *set of possible programs compatible with* K and is defined as $L_K = \cup_{\kappa \in K} L_\kappa$. A composite choice κ is an *explanation* for a query q if q is true in all possible programs of L_κ. A set K of composite choices for a query q is *covering* if the set of all possible programs where q is true is L_K.

Given a covering set of explanations K of a query q, we can build the Boolean formula

$$B(q) = \vee_{\kappa \in K} \wedge_{(C,\theta,k) \in \kappa} X_{C\theta} = k$$

where $X_{C\theta}$ is a discrete random variable having values in $\{1, \ldots, n\}$ with n the number of heads of clause C. The probability that q is true is the probability that $B(q)$ takes value true, given the probability distributions over the variables $X_{C\theta}$ that are all independent of each other.

We can associate a probability to a composite choice as follows: $P(\kappa) = \prod_{(C_i,\theta_j,k) \in \kappa} \pi_{ik}$. However, to compute $P(q)$ it is not possible to use a summation even if $B(q)$ is a disjunction because the individual disjuncts may share some of the variables. We must make the disjuncts mutually exclusive so that their probabilities can be summed up.

This task is solved by knowledge compilation: $B(q)$ is converted into a language such as Binary Decision Diagrams (BDDs), Deterministic Decomposable Negation Normal Form (d-DNNF) or Sentential Decision Diagrams (SDD) from which computing $P(q)$ is linear in the number of nodes. However, compilation to one of these languages has a cost of #P in the number of random variables.

In order to speed up inference, some languages impose restrictions. For example, PRISM [57] requires the program to be such that queries always have a pairwise incompatible covering set of explanations. In this case, once the set is found, the computation of the probability amounts to performing a sum of products. For programs to allow this kind of approach, they must satisfy the assumptions of independence of subgoals and exclusiveness of clauses, which mean that [58]:

1. the probability of a conjunction (A, B) is computed as the product of the probabilities of A and B (*independent-and assumption*),
2. the probability of a disjunction $(A; B)$ is computed as the sum of the probabilities of A and B (*exclusive-or assumption*).

These assumptions can be stated more formally by referring to explanations. Given an explanation κ, let $RV(\kappa) = \{C_i\theta_j | (C_i, \theta_j, k) \in \kappa\}$ be the set of random variables in κ. Given a set of explanations K, let $RV(K) = \bigcup_{\kappa \in K} RV(\kappa)$. Two sets of explanations, K_1 and K_2, are *independent* if $RV(K_1) \cap RV(K_2) = \emptyset$ and *exclusive* if, $\forall \kappa_1 \in K_1, \kappa_2 \in K_2$, κ_1 and κ_2 are incompatible.

The independent-and assumption means that, when deriving a covering set of explanations for a goal, the covering sets of explanations K_i and K_j for two ground subgoals in the body of a clause are independent.

The exclusive-or assumption means that, when deriving a covering set of explanations for a goal, two sets of explanations K_i and K_j obtained for a ground subgoal h from two different ground clauses are exclusive. This implies that the atom h is derived using clauses that have mutually exclusive bodies, i.e., that their bodies are not true at the same time in any world.

These assumptions make the computation of probabilities ”truth-functional” [21] (the probability of conjunction/disjunction of two propositions depends only on the probabilities of those propositions), while in the general case this is false.

In [54] the authors proposed the system PITA(IND, IND) that performs inference on programs that satisfy the *independent-or assumption* instead of the exclusive-or assumption:

3. the probability of a disjunction $(A; B)$ is computed as if A and B were independent (*independent-or assumption*).

This means that, when deriving a covering set of explanations for a goal, two sets of explanations K_i and K_j obtained for a ground subgoal h from two different ground clauses are independent.

If A and B are independent, the probability of their disjunction is

$$\begin{aligned} P(A \vee B) = P(A) + P(B) - P(A \wedge B) = \\ P(A) + P(B) - P(A)P(B) \end{aligned}$$

by the laws of probability theory.

In the next section we present Hierarchical Probabilistic Logic Programs (HPLPs) which, by construction, satisfy the independent-or assumption. This makes the computation of probabilities in such programs “truth-functional” [21]. With the independent-or assumption we can collect the contribution of multiple groundings of a clause. Therefore it is suitable for domains where entities may be related to a varying number of other entities. The differences in numbers are taken into account by the semantics. The assumption is not weaker or stronger than the exclusive-or one. It is probably less easy for users to write HPLP programs than PRISM programs but our aim is to devise an algorithm for automatically learning them.

4 Hierarchical Probabilistic Logic Programs

Suppose we want to compute the probability of atoms for a single predicate r using a PLP. In particular, we want to compute the probability of a ground atom $r(\mathbf{t})$, where $\mathbf{t}$ is a vector of terms. Let us call r the target predicate. This is a common situation in relational learning.

We consider a specific form of LPADs defining r in terms of *input predicates* (their definition is given as input and is certain) and *hidden predicates*, defined by clauses of the program. We discriminate between input predicates, which encapsulate the input data and the background knowledge, and target predicates, which are predicates we are interested in predicting. Hidden predicates are disjoint from input and target predicates. Each clause in the program has a single head atom annotated with a probability. Furthermore, the program is hierarchically defined so that it can be divided into layers. Each layer defines a set of hidden predicates in terms of predicates of the layer immediately below or in terms of input predicates. A generic clause C is of the form

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

where $\phi(\mathbf{X}, \mathbf{Y})$ is a conjunction of literals for the input predicates. The vector $\mathbf{X}$ represents variables appearing in the head of C and $\mathbf{Y}$ represents the variables introduced by input predicates. $b_i(\mathbf{X}, \mathbf{Y})$ for $i = 1, \ldots, m$ is a literal built on a hidden predicate. Variables in $\mathbf{Y}$ are existentially quantified with scope the body. Only literals for input predicates can introduce new variables into the clause. Moreover, all literals for hidden predicates must use the whole set of variables of the predicate in the head $\mathbf{X}$ and input predicates $\mathbf{Y}$, see predicate $r_{1\_1}(A, B, C)$ Example 2. This restriction is imposed to allow independence among ground clauses associated to target/hidden predicates, see Section 4.1. Moreover, we require that the predicate of each $b_i(\mathbf{X}, \mathbf{Y})$ does not appear elsewhere in the body of C or in the body of any other clause, i.e each hidden predicate literal is unique in the program. We call hierarchical PLP (HPLP) the language that admits only programs of this form [39]. A generic hierarchical program is defined as follows:

$$
\begin{array}{ll}
C_1 = r(\mathbf{X}) : \pi_1 & :- \ \phi_1, b_{1\_1}, \ldots, b_{1\_m_1} \\
 & \ldots \\
C_n = r(\mathbf{X}) : \pi_n & :- \ \phi_n, b_{n\_1}, \ldots, b_{n\_m_n} \\
C_{1\_1\_1} = r_{1\_1}(\mathbf{X}) : \pi_{1\_1\_1} & :- \ \phi_{1\_1\_1}, b_{1\_1\_1\_1}, \ldots, b_{1\_1\_1\_m_{111}} \\
 & \ldots \\
C_{1\_1\_n_{11}} = r_{1\_1}(\mathbf{X}) : \pi_{1\_1\_n_{11}} & :- \ \phi_{1\_1\_n_{11}}, b_{1\_1\_n_{11}\_1}, \ldots, b_{1\_1\_n_{11}\_m_{11n_{11}}} \\
 & \ldots \\
C_{n\_1\_1} = r_{n\_1}(\mathbf{X}) : \pi_{n\_1\_1} & :- \ \phi_{n\_1\_1}, b_{n\_1\_1\_1}, \ldots, b_{n\_1\_1\_m_{n11}} \\
 & \ldots \\
C_{n\_1\_n_{n1}} = r_{n\_1}(\mathbf{X}) : \pi_{n\_1\_n_{n1}} & :- \ \phi_{n\_1\_n_{n1}}, b_{n\_1\_n_{n1}\_1}, \ldots, b_{n\_1\_n_{n1}\_m_{n1n_{n1}}} \\
 & \ldots
\end{array}
$$

where r is the target predicate and $r_{1\_1\ldots\_n}$ is the predicate of $b_{1\_1\ldots\_n}$, e.g. r_{1_1} and r_{n_1} are the predicates of b_{1_1} and b_{n_1} respectively. The bodies of the lowest layer of clauses are composed only of input predicates and do not contain hidden predicates, e.g C_2 and $C_{1\_1\_1}$ in Example 2. Note that here the variables were omitted except for rule heads.

A generic program can be represented by a tree, Figure 1 with a node for each clause and literal for hidden predicates. Each clause (literal) node is indicated with $C_\mathbf{p}$ ($b_\mathbf{p}$) where $\mathbf{p}$ is a sequence of integers encoding the path from the root to the node. The predicate of literal $b_\mathbf{p}$ is $r_\mathbf{p}$ which is different for every value of $\mathbf{p}$.

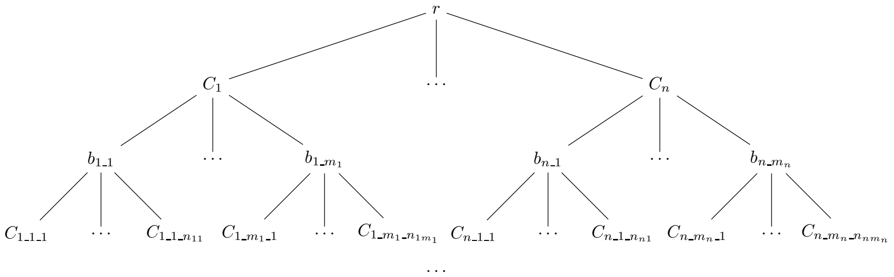

Fig. 1: Probabilistic program tree.

Example 2 Let us consider a modified version of the program of Example 1:

$C_1 = advised\_by(A, B) : 0.3 :-$
$student(A), professor(B), project(C, A), project(C, B),$
$r_{1\_1}(A, B, C).$
$C_2 = advised\_by(A, B) : 0.6 :-$
$student(A), professor(B), ta(C, A), taughtby(C, B).$
$C_{1\_1\_1} = r_{1\_1}(A, B, C) : 0.2 :-$
$publication(P, A, C), publication(P, B, C).$

where $publication(P, A, C)$ means that P is a publication with author A produced in project C and $student/1, professor/1, project/2, ta/2, taughtby/2$ and $publication/3$ are input predicates.

In this case, the probability of $q = advised\_by(harry, ben)$ depends not only on the number of joint courses and projects but also on the number of joint publications from projects. Note that the variables of the hidden predicate $r_{1\_1}(A, B, C)$ is the union of the variables $\mathbf{X} = \{A, B\}$ of the predicate $advised\_by(A, B)$ in the head of the clause and $\mathbf{Y} = \{C\}$ which is the variable introduced by the input predicate $project(C, A)$ in the body. The clause for $r_{1\_1}(A, B, C)$ $(C_{1\_1\_1})$ computes an aggregation over publications of a project and the clause level above (C_1) aggregates over projects.

This program can be represented with the tree of Figure 2.

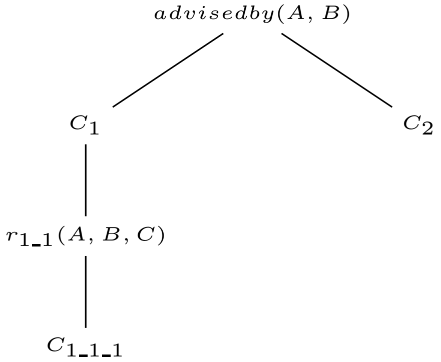

Fig. 2: Probabilistic program tree for Example 2.

4.1 Inference in HPLP

HPLPs satisfy the independent-or assumption as every hidden predicate appears just once in the body of a single clause and has all the variables of the clause as arguments. If we consider the grounding of the program, each ground atom for a hidden predicate appears in the body of a single ground clause. A ground atom for a hidden predicate may appear in the head of more than one ground clauses and each ground clause can provide a set of explanations for it but these sets do not share random variables. Moreover, HPLPs also satisfy the independent-and assumptions as every ground atom for a hidden predicate in the body of a ground clause has explanations that depend on disjoint sets of random variables.

Remember that atoms for input predicates are not random, they are either true or false in the data and they play a role in determining which groundings of clauses contribute to the probability of the goal: the groundings whose atoms for input predicates are false do not contribute to the computation of the probability.

Given a ground clause $G_{\mathbf{p}i} = a_{\mathbf{p}} : \pi_{\mathbf{p}i} :- b_{\mathbf{p}i1}, \ldots, b_{\mathbf{p}im_{\mathbf{p}_1}}$. where $\mathbf{p}$ is a path, we can compute the probability that the body is true by multiplying the probability of being true of each individual literals. If the literal is positive, its probability is equal to the probability of the corresponding atom. Otherwise it is one minus the probability of the corresponding atom. Therefore the probability of the body of $G_{\mathbf{p}i}$ is $P(b_{\mathbf{p}i1}, \ldots, b_{\mathbf{p}im_{\mathbf{p_l}}}) = \prod_{i=k}^{m_{\mathbf{p}}} P(b_{\mathbf{p}ik})$ and $P(b_{\mathbf{p}ik}) = 1 - P(a_{\mathbf{p}ik})$ if $b_{\mathbf{p}ik} = \neg a_{\mathbf{p}ik}$. If $P(b_{\mathbf{p}ik})$ is a literal for an input predicate, $P(b_{\mathbf{p}ik}) = 1$ if it is true and $P(b_{\mathbf{p}ik}) = 0$ otherwise. We can use this formula because HPLPs satisfy the independent-and assumption.

Given a ground atom $a_{\mathbf{p}}$ for a hidden predicate, to compute $P(a_{\mathbf{p}})$ we need to take into account the contribution of every ground clause for the predicate of $a_{\mathbf{p}}$. Suppose these clauses are $\{G_{\mathbf{p}1}, \ldots, G_{\mathbf{p}o_{\mathbf{p}}}\}$. If we have a single ground clause $G_{\mathbf{p}1} = a_{\mathbf{p}} : \pi_{\mathbf{p}1} :- \, b_{\mathbf{p}11}, \ldots, b_{\mathbf{p}1m_{\mathbf{p}1}}$, then

$$P(a_{\mathbf{p}}) = \pi_{\mathbf{p}1} \cdot P(body(G_{\mathbf{p}1})).$$

If we have two clauses, the contribution of each clause is computed as above. Note that, random variables associated with these contributions are independent since we allow hidden predicates in the body of each clause to use the whole set of variables appearing in the head and those introduced by input predicates. It is also worth noting that the probability of a disjunction of two independent random variables is

$$P(a \vee b) = P(a) + P(b) - P(a) \cdot P(b) = 1 - (1 - P(a)) \cdot (1 - P(b)).$$

Therefore, if we have two clauses their contributions are combined as follows:

$$\begin{aligned} P(a_{\mathbf{p}}) &= 1 - (1 - \pi_{\mathbf{p}1} \cdot P(body(G_{\mathbf{p}1})) \cdot (1 - \pi_{\mathbf{p}2} \cdot P(body(G_{\mathbf{p}2}))) \\ &= (\pi_{\mathbf{p}1} \cdot P(body(G_{\mathbf{p}1}))) \oplus (\pi_{\mathbf{p}2} \cdot P(body(G_{\mathbf{p}2}))). \end{aligned}$$

where we defined 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

$$\bigoplus_i p_i = 1 - \prod_i (1 - p_i) \tag{2}$$

The operator is called probabilistic sum and is the t-norm of the product fuzzy logic [22]. Therefore, if the probabilistic program is ground, the probability of the example atom can be computed with the Arithmetic Circuit (AC) of Figure 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 $\oplus$. We can thus compute the output p in time linear in the number of ground clauses. Note that, the AC can also be interpreted as a deep neural network with nodes (or neurons) whose (non linear) activation functions are × and $\oplus$. When the program is not ground, we can build its grounding obtaining a circuit/neural network of the type of Figure 3, where however some of the parameters can be the same for different edges. In this case the circuit will exhibit parameter sharing.

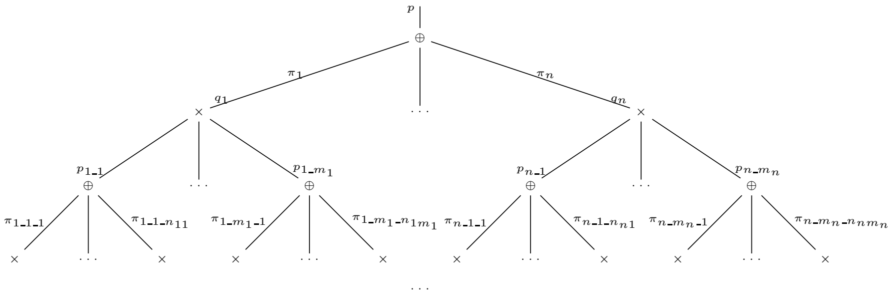

Fig. 3: Arithmetic circuit/neural net.

Example 3 Consider the completed version of Example 2: An online implementation can be found at `http://cplint.eu/e/phil/uwcse.pl`

$C_1 = advised\_by(A, B) : 0.3 :-$
$\quad student(A), professor(B), project(C, A), project(C, B),$
$\quad r_{1\_1}(A, B, C).$
$C_2 = advised\_by(A, B) : 0.6 :-$
$\quad student(A), professor(B), ta(C, A), taughtby(C, B).$
$C_{1\_1\_1} = r_{1\_1}(A, B, C) : 0.2 :-$
$\quad publication(P, A, C), publication(P, B, C).$
$student(harry).$
$professor(ben).$
$project(pr_1, harry).\ project(pr_2, harry).$
$project(pr_1, ben).\ project(pr_2, ben).$
$taught\_by(c_1, ben).\ taught\_by(c_2, ben).$
$ta(c_1, harry).\ ta(c_2, harry).$
$publication(p_1, harry, pr_1).\ publication(p_2, harry, pr_1).$
$publication(p_3, harry, pr_2).\ publication(p_4, harry, pr_2).$
$publication(p_1, ben, pr_1).\ publication(p_2, ben, pr_1).$
$publication(p_3, ben, pr_2).\ publication(p_4, ben, pr_2).$

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

$$\begin{aligned}
G_1 &= advisedby(harry, ben) : 0.3 :- \\
&\quad student(harry), professor(ben), project(pr_1, harry), \\
&\quad project(pr_1, ben), r_{1\_1}(harry, ben, pr_1). \\
G_2 &= advisedby(harry, ben) : 0.3 :- \\
&\quad student(harry), professor(ben), project(pr_2, harry), \\
&\quad project(pr_2, ben), r_{1\_1}(harry, ben, pr_2). \\
G_3 &= advisedby(harry, ben) : 0.6 :- \\
&\quad student(harry), professor(ben), ta(c_1, harry), taughtby(c_1, ben). \\
G_4 &= advisedby(harry, ben) : 0.6 :- \\
&\quad student(harry), professor(ben), ta(c_2, harry), taughtby(c_2, ben). \\
G_{1\_1\_1} &= r_{1\_1}(harry, ben, pr_1) : 0.2 :- \\
&\quad publication(p_1, harry, pr_1), publication(p_1, ben, pr_1). \\
G_{1\_1\_2} &= r_{1\_1}(harry, ben, pr_1) : 0.2 :- \\
&\quad publication(p_2, harry, pr_1), publication(p_2, ben, pr_1). \\
G_{2\_1\_1} &= r_{1\_1}(harry, ben, pr_2) : 0.2 :- \\
&\quad publication(p_3, harry, pr_2), publication(p_3, ben, pr_2). \\
G_{2\_1\_2} &= r_{1\_1}(harry, ben, pr_2) : 0.2 :- \\
&\quad publication(p_4, harry, pr_2), publication(p_4, ben, pr_2).
\end{aligned}$$

The program tree is shown in Figure 4 and the corresponding arithmetic circuit in Figure 5 together with the values computed by the nodes. Inference in the AC of Figure 5 can be computed at `http://cplint.eu/e/phil/uwcse.pl` by running the query

$$inference\_hplp(advisedby(harry, ben), ai, Prob).$$

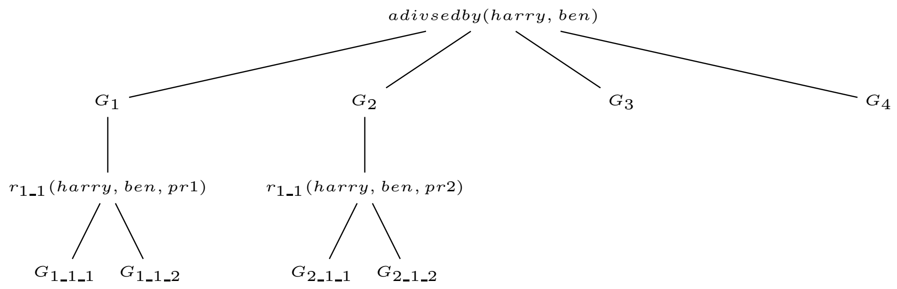

Fig. 4: Ground probabilistic program tree for Example 3.

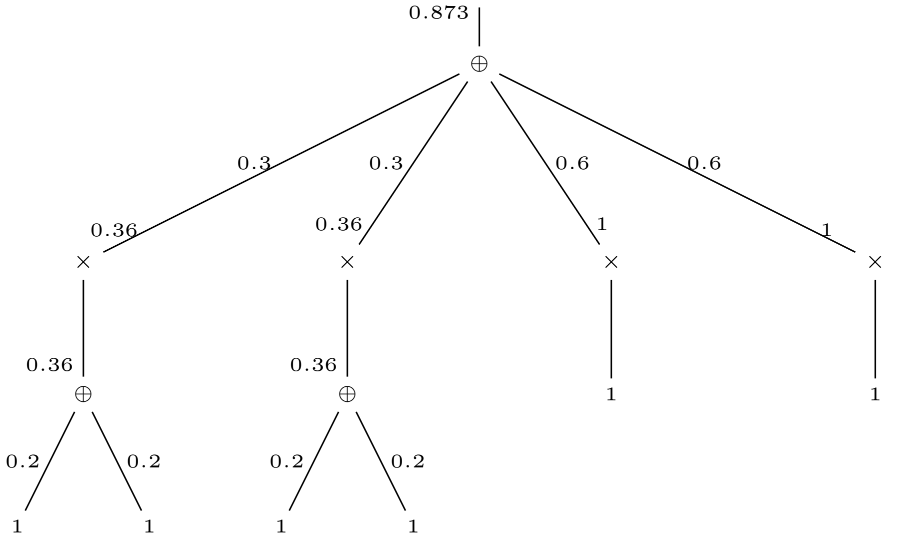

Fig. 5: Arithmetic circuit/neural net for Example 3.

4.2 Building the Arithmetic Circuit

The network can be built by performing inference using tabling and answer subsumption using PITA(IND,IND) [49,53]. Suppose we want to compute the probability of a query, which is typically the target predicate in hierarchical PLP. To speed the computation, PITA(IND,IND) applies a program transformation that adds an extra argument to the query and to each subgoal of the program. The probability of answers to each subgoal is stored in the extra argument. In fact, 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. Note that because of its structure, when a subgoal returns the original arguments are guaranteed to be instantiated in hierarchical PLP.

Moreover, suitable literals are also added to the body of clauses for combining the extra arguments of the subgoals: the probability to be assigned to the extra argument of the head atom is the product of probabilities of the answers for the subgoals in the body.

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, [62]. Tabling keeps a store of 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 recomputing. Answer subsumption [62] is a tabling feature that, when a new answer for a tabled subgoal is found, combines old answers with the new one. This combination operator is the probabilistic sum in PITA(IND, IND). Computation by PITA(IND, IND) is thus equivalent to the evaluation of the program arithmetic circuit.

We use an algorithm similar to PITA(IND,IND) to build the Arithmetic circuit or the neural network instead of just computing the probability. To build the arithmetic circuit, 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 result of inference would thus be a term representing the arithmetic circuit.

There is a one to one correspondence between the data structure built by PITA(IND,IND) and the circuit so there is no need of an expensive compilation step. The term representing the AC in Figure 5 is[1]

or([and([0, or([and([2])])]), and([0, or([and([2])])]),
and([0, or([and([2])])]), and([0, or([and([2])])]), and([1]), and([1])])

where the operators *and* and *or* represent the product × and the probabilistic sum ⊕ respectively. Once the ACs are built, parameter learning can be performed over them by applying gradient descent/back-propagation or Expectation Maximization. The following section presents these algorithms and their regularized versions. Because of the constraints imposed on HPLPs, writing these programs may be unintuitive for human so we also propose, in Section 6, an algorithm for learning both the structure and the parameters from data.

5 Parameter Learning

In this section, we present an algorithm, called Parameter learning for HIerarchical probabilistic Logic programs (PHIL), which learns HPLP's parameters from data. We present two versions of PHIL. The first, Deep PHIL (DPHIL), learns HPLP's parameters by applying a gradient-based method and the second, Expectation Maximization PHIL (EMPHIL), applies Expectation Maximization (EM). Different regularized versions of each algorithm will also be presented. The parameter learning algorithm can be defined as follows:

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

$$LL = \arg\max_{\Pi}(\sum_{i=1}^{M} \log P(e_i) + \sum_{i=M+1}^{N} \log(1 - P(e_i))) \quad (3)$$

[1] Obtaining by running the query $inference\_hplp(advisedby(harry, ben), ai, Prob, Circuit)$. at `http://cplint.eu/e/phil/uwcse.pl`

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

$$err = \sum_{i=1}^{N+M} (-y_i \log P(e_i) - (1 - y_i) \log(1 - P(e_i))) \tag{4}$$

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 variants) are presented in subsections 5.1 and 5.2 respectively.

5.1 Gradient Descent: DPHIL

DPHIL computes the gradient of the error *err* (4) with respect to each parameter and updates the parameters. We do this by building an AC for each example and by running a dynamic programming algorithm for computing the gradient. To simplify gradient computation, we transformed the AC of Figure 3 as follows: weight, π_i, labeling arcs from $\oplus$ 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 in Figure 5 is converted into the one shown in Figure 6. Note that, the ACs in Figure 5 and 6 are equivalent but the one in Figure 6 highlights parameters sharing which is more convenient for illustrating the algorithm.

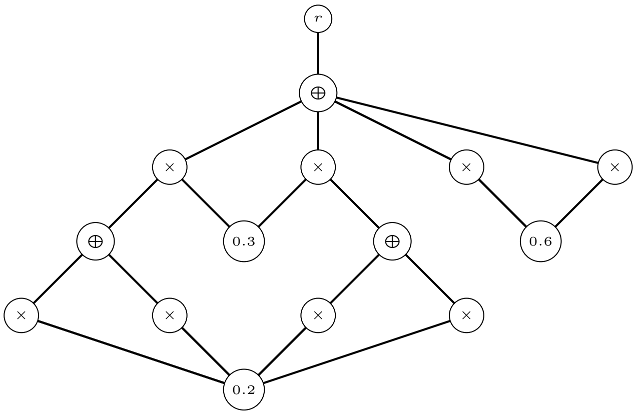

Fig. 6: Converted arithmetic circuit of Figure 5.

.

The standard gradient descent algorithm computes gradients at each iteration using all 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 converge and high accuracy. DPHIL, shown in Algorithm 1, implements SGD.

After building the ACs and initializing the weights, the gradients and the moments, line 2–6, DPHIL performs two passes over each AC in the current batch, line 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

Algorithm 1 Function DPHIL.

```
1: function PHIL($Theory, \epsilon, \delta, MaxIter, \beta_1, \beta_2, \eta, \hat{\epsilon}, Strategy$)
2:     $Examples \leftarrow$ BUILDACS($Theory$)                      ▷ Build the set of ACs
3:     for $i \leftarrow 1 \rightarrow |Theory|$ do          ▷ Initialize weights,gradient and moments vector
4:         $W[i] \leftarrow random(Min, Max)$                ▷ initially $W[i] \in [Min, Max]$.
5:         $G[i] \leftarrow 0.0$, $G[i] \leftarrow 0.0$ $M_0[i] \leftarrow 0.0$, $M_1[i] \leftarrow 0.0$
6:     end for
7:     $Iter \leftarrow 1$
8:     repeat
9:         $LL \leftarrow 0$
10:        $Batch \leftarrow$ NEXTBATCH($Examples$)   ▷ Select the batch according to the strategy
11:        for all $node \in Batch$ do
12:            $P \leftarrow$ FORWARD($node$)
13:            BACKWARD($G, -\frac{1}{P}, node$)
14:            $LL \leftarrow LL + \log P$
15:        end for
16:        UPDATEWEIGHTSADAM($W, G, M_0, M_1, \beta_1, \beta_2, \eta, \hat{\epsilon}, Iter$)
17:    until $LL - LL_0 < \epsilon \vee LL - LL_0 < -LL.\delta \vee Iter > MaxIter$
18:    $FinalTheory \leftarrow$ UPDATETHEORY($Theory, W$)
19:    return $FinalTheory$
20: end function
```

weights between -∞ and + ∞ and expressing the parameters using the sigma function $\pi_i = \sigma(W_i) = \frac{1}{1+e^{-W_i}}$ (5). In this way we do not have to impose the constraint that the parameters are in [0,1].

Function Forward 2 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 $node = not(n)$, $p =$ Forward(n) is computed and $1-p$ is assigned to $v(node)$, lines 2–5. If $node = \bigoplus(n_1, \ldots n_m)$, function $v(n_i) =$ 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 $v(n_i) =$ 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.

Algorithm 2 Function Forward

```
 1: function Forward($node$)                                         ▷ node is an AC
 2:     if $node = not(n)$ then
 3:         $v(node) \leftarrow 1 -$ Forward($n$)
 4:         return $v(node)$
 5:     else
 6:                  ▷ Compute the output example by recursively call Forward on its sub AC
 7:         if $node = \bigoplus(n_1, \ldots n_m)$ then                    ▷ $\bigoplus$ node
 8:             for all $n_j$ do
 9:                 $v(n_j) \leftarrow$ Forward($n_j$)
10:             end for
11:             $v(node) \leftarrow v(n_1) \oplus \ldots \oplus v(n_m)$
12:             return $v(node)$
13:         else                                                     ▷ and Node
14:             if $node = \times(\pi_i, n_1, \ldots n_m)$ then
15:                 for all $n_j$ do
16:                     $v(n_j) \leftarrow$ Forward($n_j$)
17:                 end for
18:                 $v(node) \leftarrow \pi_i \cdot v(n_1) \cdot \ldots \cdot v(n_m)$
19:                 return $v(node)$
20:             end if
21:         end if
22:     end if
23: end function
```

Procedure Backward takes an evaluated AC *node* and computes the 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. This derivative is given in Equation 6, see Appendix 10.1 for the proof.

$$\frac{\partial err}{\partial v(n)} = -d(n)\frac{1}{v(r)} \tag{6}$$

with

$$d(n) = \begin{cases} d(pa_n)\frac{v(pa_n)}{v(n)} & \text{if n is a } \bigoplus node, \\ d(pa_n)\frac{1-v(pa_n)}{1-v(n)} & \text{if n is a } \times \text{ node} \\ \sum_{pa_n} d(pa_n).v(pa_n).(1-\pi_i) & \text{if n}{=}\sigma(W_i) \\ -d(pa_n) & pa_n = not(n) \end{cases} \tag{7}$$

where pa_n indicate the parents of n.

This leads to Procedure BACKWARD shown in Algorithm 3 which is a simplified version for the case $v(n) \neq 0$ for all $\bigoplus$ nodes. To compute $d(n)$, Backward proceeds by recursily 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 $-AccGrad$, line 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 × 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, line 11-15. For leaf node, i.e a π_i node, the derivative is accumulated, line 20.

After the computation of the gradients, weights are updated. 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 that 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 learning rate at each iteration combining the advantages of both strategies. We use the update method *Adam*, adaptive moment estimation [27], 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 weights are updated with a fraction, current learning rate, of the combination of these moments, see Procedure UPDATEWEIGHTSADAM in Algorithm 4.

Algorithm 3 PROCEDURE BACKWARD

```
procedure BACKWARD($G, AccGrad, node$)
    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
```

Algorithm 4 PROCEDURE UPDATEWEIGHTSADAM

```
procedure UPDATEWEIGHTSADAM($W, G, M_0, M_1, \beta_1, \beta_2, \eta, \hat{\varepsilon}, iter$)
    $\eta_{iter} \leftarrow \eta \frac{\sqrt{1-\beta_2^{iter}}}{1-\beta_1^{iter}}$
    for $i \leftarrow 1 \rightarrow |W|$ do
        $M_0[i] \leftarrow \beta_1 \cdot M_0[i] + (1 - \beta_1) \cdot G[i]$
        $M_1[i] \leftarrow \beta_2 \cdot M_1[i] + (1 - \beta_2) \cdot G[i] \cdot G[i]$
        $W[i] \leftarrow W[i] - \eta_{iter} \cdot \frac{M_0[i]}{(\sqrt{M_1[i]}) + \hat{\epsilon}}$
    end for
end procedure
```

5.1.1 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 exist two main regularization techniques called L_1 and L_2 regularization that differ on the way they penalize the loss function. While L_1 adds to the loss function the sum of the absolute value of the parameters, L_2 adds the sum of their squares. Given the loss function defined in Equation 4, the corresponding regularized loss function are given by equations 8 and 9.

$$err_1 = \sum_{i=1}^{N+M} -y_i \log P(e_i) - (1-y_i)\log(1-P(e_i)) + \gamma \sum_{i=1}^{k} |\pi_i| \tag{8}$$

$$err_2 = \sum_{i=1}^{N+M} -y_i \log P(e_i) - (1-y_i)\log(1-P(e_i)) + \frac{\gamma}{2} \sum_{i=1}^{k} \pi_i^2 \tag{9}$$

where the regularization hyper-parameter γ determines how much to penalize the parameters and k the number of parameters. When γ is zero, the regularization term becomes zero and we are back to the original loss function. When γ is large, we penalize the parameters and they tend to become small. Note also that we add the regularization term from the initial loss function because we are performing minimization. The main difference between these techniques is that while L_1 favor sparse parameters (parameters closer to zero) L_2 does not. 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 regularization term with respect to π_i to the one obtained in Equation 7. The regularized errors are given by:

$$E_{reg} = \begin{cases} \gamma \sum_{i=1}^{k} \pi_i & \text{for } L_1 \\ \frac{\gamma}{2} \sum_{i=1}^{k} \pi_i^2 & \text{for } L_2 \end{cases} \tag{10}$$

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

$$\frac{\partial E_{reg}}{\partial W_i} = \begin{cases} \gamma \frac{\partial \sigma(W_i)}{\partial W_i} = \gamma \cdot \sigma(W_i) \cdot (1 - \sigma(W_i)) = \gamma \cdot \pi_i \cdot (1 - \pi_i) \\ \\ \frac{\gamma}{2} \frac{\partial \sigma(W_i)^2}{\partial W_i} = \gamma \cdot \sigma(W_i) \cdot \sigma(W_i) \cdot (1 - \sigma(W_i)) = \gamma \cdot \pi_i^2 \cdot (1 - \pi_i) \end{cases} \tag{11}$$

So Equation 7 becomes

$$d(n) = \begin{cases} d(pa_n)\frac{v(pa_n)}{v(n)} & \text{if n is a } \bigoplus node, \\ d(pa_n)\frac{1-v(pa_n)}{1-v(n)} & \text{if n is a } \times \text{ node} \\ \sum_{pa_n} d(pa_n).v(pa_n).(1-\pi_i) + \frac{\partial E_{reg}}{\partial W_i} & \text{if n=}\sigma(W_i) \\ -d(pa_n) & pa_n = not(n) \end{cases} \tag{12}$$

In order to implementent the regularized version of DPHIL (DPHIL$_1$ and DPHIL$_2$), the forward and the backward passes described in algorithms 2 and

3 remain unchanged. The unique change occurs while updating the parameters in the algorithm 4. *UpdateWeightsAdam* line 6 becomes

$$W[i] \leftarrow W[i] - \eta_{iter} * \frac{M_0[i]}{(\sqrt{M_1[i]}) + \hat{\epsilon}} + \frac{\partial E_{reg}}{\partial W_i} \tag{13}$$

5.2 Expectation Maximization: EMPHIL

We propose another algorithm, EMPHIL, that learns the parameters of HPLP by applying Expectation Maximization (EM). The algorithm maximizes the log-likelihood LL defined in Equation 3 by alternating between an Expectation (E) and a Maximization (M) step. 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 a *hierarchical* PLP $H = \{C_i | i = 1, \cdots, n\}$ (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:

Let C_i a generic rule and $g(i) = \{j | \theta_j$ is a substitution grounding $C_i\}$. 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 $x \in \{0, 1\}$, for all $j \in g(i)$. So $\mathbf{E}[c_{ix}|e] = \sum_{j \in g(i)} P(X_{ij} = x|e)$. These values are aggregated over all examples obtaining

$$N_0[i] = \mathbf{E}[c_{i0}] = \sum_{e \in E} \sum_{j \in g(i)} P(X_{ij} = 0|e) \tag{14}$$

$$N_1[i] = \mathbf{E}[c_{i1}] = \sum_{e \in E} \sum_{j \in g(i)} P(X_{ij} = 1|e) \tag{15}$$

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

$$\pi_i = \frac{N_1}{\sum_{e \in E} |g(i)|} \tag{16}$$

Therefore to perform EMPHIL, we have to compute $P(X_{ij} = 1|e)$ for each example e by performing the belief propagation algorithm over the factor graph associated with the AC. Message passing is then applied over the AC. We show, in the Appendix 10.2, that the messages in the bottom-up, c_N, and top-down, t_N, are respectively given as

$$c_N = v(N) \tag{17}$$

$$t_N = \begin{cases} \frac{t_P}{t_P + v(P) \ominus v(N) \cdot t_P + (1 - v(P) \ominus v(N)) \cdot (1 - t_p)} & \text{if } P \text{ is a } \oplus \text{ node} \\ \frac{t_P \cdot \frac{v(P)}{v(N)} + (1 - t_P) \cdot (1 - \frac{v(P)}{v(N)})}{t_P \cdot \frac{v(P)}{v(N)} + (1 - t_P) \cdot (1 - \frac{v(P)}{v(N)}) + (1 - t_P)} & \text{if } P \text{ is a } \times \text{ node} \\ 1 - t_P & \text{if } P \text{ is a } \neg \text{ node} \end{cases} \tag{18}$$

where $v(N)$ is the value of node N, P is its parent and the operator $\ominus$ is defined as

$$v(p) \ominus v(n) = 1 - \frac{1 - v(p)}{1 - v(n)} \tag{19}$$

c_N is performed by applying the forward pass described in Algorithm 2.

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 5 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 5 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 $node = \bigoplus(n_1, \ldots n_m)$ 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 $node = \times(n_1, \ldots n_m)$ 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 6. 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).

Algorithm 6 Function EMPHIL.

```
 1: function EMPHIL($Theory, \epsilon, \delta, MaxIter, \gamma, a, b, Type$)
 2:     $Examples \leftarrow$ BuildACs($Theory$)                          ▷ Build the set of ACs
 3:     for $i \leftarrow 1 \rightarrow |Theory|$ do
 4:         $\Pi[i] \leftarrow random; B[i], Count[i] \leftarrow 0$             ▷ Initialize the parameters
 5:     end for
 6:     $LL \leftarrow -inf; Iter \leftarrow 0$
 7:     repeat
 8:         $LL_0 \leftarrow LL, LL \leftarrow 0$                                ▷ Expectation step
 9:         for all $node \in Examples$ do
10:             $P \leftarrow$ Forward($node$)
11:             BackwardEM($1, node, N_0, N_1$)
12:             $LL \leftarrow LL + \log P$
13:         end for                                                   ▷ Maximization step
14:         for $i \leftarrow 1 \rightarrow |Theory|$ do
15:             switch $Type$
16:                 case 0: $\Pi[i] \leftarrow \frac{B[i]}{N_0[i]+N_1[i]}$
17:                 case 1: $\Pi[i] \leftarrow \frac{4N_1[i]}{2(\gamma+N_0[i]+N_1[i]+\sqrt{(N_0[i]+N_1[i])^2+\gamma^2+2\gamma(N_0[i]-N_1[i]))}}$
18:                 case 2:
19:                     let $V = 2\sqrt{\frac{3N_0+3N_1+\gamma}{\gamma}}\cos\left(\frac{\arccos\left(\frac{\sqrt{\frac{\gamma}{3N_0+3N_1+\gamma}}\left(\frac{9N_0}{2}-9N_1+\gamma\right)}{3N_0+3N_1+\gamma}\right)}{3}-\frac{2\pi}{3}\right)$
20:                     $\Pi[i] \leftarrow \frac{V}{3} + \frac{1}{3}$
21:                 case 3: $\Pi[i] \leftarrow \frac{N_1+a}{N_0+N_1+a+b}$
22:             end switch
23:             $B[i], Count[i] \leftarrow 0$
24:         end for
25:     until $LL - LL_0 < \epsilon \vee LL - LL_0 < -LL.\delta \vee Iter > MaxIter$
26:     $FinalTheory \leftarrow$ UpdateTheory($Theory, \Pi$)
27:     return $FinalTheory$
28: end function
```

5.2.1 EMPHIL regularization: EMPHIL$_1$, EMPHIL$_2$ and EMPHIL$_B$

In this section, we propose three regularized versions of EMPHIL. As described in [31], 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 regularization objective functions for

L_1 and L_2 respectively.

$$J(\theta) = \begin{cases} N_1 \log \theta + N_0 \log(1-\theta) - \gamma\theta & \text{for } L_1 \\ N_1 \log \theta + N_0 \log(1-\theta) - \frac{\gamma}{2}\theta^2 & \text{for } L_2 \end{cases} \tag{20}$$

where $\theta = \pi_i$, N_0 and N_1 are the expectations computed in the E-step (see equations 14 and 15). 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 10.3 for the proofs).

Theorem 1 *The L_1 regularized objective function:*

$$J_1(\theta) = N_1 \log \theta + N_0 \log(1-\theta) - \gamma\theta \tag{21}$$

is maximum in

$$\theta_1 = \frac{4N_1}{2(\gamma + N_0 + N_1 + \sqrt{(N_0+N_1)^2 + \gamma^2 + 2\gamma(N_0 - N_1)})}$$

Theorem 2 *The L_2 regularized objective function:*

$$J_2(\theta) = N_1 \log \theta + N_0 \log(1-\theta) - \frac{\gamma}{2}\theta^2 \tag{22}$$

is maximum in

$$\theta_2 = \frac{2\sqrt{\frac{3N_0+3N_1+\gamma}{\gamma}} \cos\left(\frac{\arccos\left(\frac{\sqrt{\frac{\gamma}{3N_0+3N_1+\gamma}}\left(\frac{9N_0}{2} - 9N_1 + \gamma\right)}{3N_0+3N_1+\gamma} \right)}{3} - \frac{2\pi}{3} \right)}{3} + \frac{1}{3}$$

We consider another regularization method for EMPHIL (called EMPHIL$_B$) 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$_B$ computes

$$\pi_i = \frac{N_1 + a}{N_0 + N_1 + a + b} \tag{23}$$

as described in [8]. a and b are hyper-parameters. We choose $a = 0$ and b as a fraction of the training set size, see Section 8, since we want small parameters.

So algorithms EMPHIL (the standard EM), EMPHIL$_1$, EMPHIL$_2$ and EMPHIL$_B$ differ in the way they update the parameters in the M-step, Algorithm 6 lines 15-22.

6 Structure learning

In the previous section, the structure of an HPLP was given and the task was to learn its parameters from data. Since hidden predicates could be difficult to interprete for humans in many domains of interest, providing the structure of an HPLP may be unintuitive and tedious even for experts in the domains. In this section, we propose an algorithm for learning both the structure and the parameters from data. The structure is learned by mean of predicate invention. The Structure learning problem is defined as follows:

Definition 2 (Structure Learning Problem) Given a set of mega-examples (interpretations), each containing positive and negative examples for a target predicate and facts for input predicates, $I = \{e_1, \ldots, e_M, \textbf{not } e_{M+1}, \ldots, \textbf{not } e_N\}$, find the HPLP with parameters $\mathit{\Pi}$ that maximizes the (log) likelihood

$$LL = \arg\max_{\mathit{\Pi}} \sum_{k}^{NI} (\sum_{i=1}^{M} \log P(e_{i_k}) + \sum_{i=M+1}^{N} \log(1 - P(e_{i_k}))) \tag{24}$$

where $P(e_{i_k})$ is the probability assigned to e_{i_k} (an example from the interpretation k). Note that a mega-example, also called mega-interpretation, describes one world in the domain of interest. It includes positive and negative examples for the target predicate and fact for input predicates. It is called mega because it can include more than one example, positive or negative. A mega-example is similar to a partial interpretation in a context-dependent partial interpretation defined in [29]

SLEAHP learns HPLPs by generating an initial set of bottom clauses (from a language bias) from which a large HPLP is derived. Then SLEAHP performs structure learning by using parameter learning. 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.

6.1 Language bias

An initial set of clauses are generated according to a language bias expressed by means of mode declaration. A mode declaration, [37], is a set of head, $\mathit{modeh}(\mathit{rec}, s)$, and body, $\mathit{modeb}(\mathit{rec}, s)$, declarations where rec is a recall and s, called schema, is a template for ground literals in the head or body of clauses. The schema contains special place-marker terms of the form `#type`, `+type` and `-type`, which stand, respectively, for ground terms, input variables and output variables of a type. The language defined by a set of mode declarations M is indicated with L(M). In clauses from L(M), 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. The head atoms (resp. body literals) of clauses are obtained from some head (resp. body) declaration in M by replacing all `#` place-markers with ground terms and all `+` (resp. `-`) place-markers with input (resp. output) variables. This type of mode declarations is extended with place-marker terms of the form -# which are treated as # when defining L(M) but differ in the creation of the bottom clauses, see Section 6.2.1.

6.2 Description of the algorithm

In order to learn an HPLP, SLEAHP (Algorithm 7) 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. Then 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. The components of this algorithm are described in the following subsections.

Algorithm 7 Function STRUCTURE LEARNING

```
1: function SLEAHP($NInt, NS, NA, MaxProb, NumLayer, MaxIter, \epsilon, \delta$,$MinProb$)
2:     $Clauses \leftarrow$genClauses($NInt, NS, NA$)                ▷ Generate clauses
3:     $Tree \leftarrow$genTree(Clauses)                             ▷ Build the tree
4:     $init\_HPLP \leftarrow$genHPLP(Clauses,MaxProb,NumLayer)      ▷ Generate the initial HPLP
5:     $(LL, final\_HPLP) \leftarrow$PHIL_Reg($init\_HPLP, MaxIter, \epsilon, \delta$)   ▷ Learns the parameters
6:     return $final\_HPLP$
7: end function
```

6.2.1 Clause generation

Algorithm 8 generates a set of bottom clauses as in Progol, [37]. Given a language bias and a set of mega-examples, Algorithm 8 generates a set of bottom clauses. These bottom clauses are then used for creating a tree of literals, Algorithm 9.

Consider the target predicate r/ar, the predicate of the schema s associates with a fact $modeh(rec, s)$ in the language bias. In order to create a clause, a mega-example I and an answer h for the goal $schema(s)$ are randomly selected with replacement, I from the available set of mega-examples and h from the set of all answers found for the goal $schema(s)$ in I. $schema(s)$ is the literal obtained from s by replacing all place-markers with distinct variables $X_1 \cdots X_{ar}$. Then h is saturated using Progol's saturation method as described in [37].

This cycle is repeated for a user-defined number NS of times and the resulting ground clause $BC = h :- b_1, \ldots, b_m$ is then processed to obtain a probabilistic program clause by replacing each term in a + or - place-marker with a variable, using the same variable for identical terms. Terms corresponding to # or -# place-markers are instead kept in the clause. This process is repeated for a number $NInt$ of input mega-examples and a number NA of answers, thus obtaining $NInt \cdot NA$ bottom clauses.

Algorithm 8 Function GenerateClauses

```
function genClauses($NInt, NS, NA$)
    for all predicates $P/Ar$ do
        $Clauses \leftarrow []$
        for all modeh declarations $modeh(rec, s)$ with $P/Ar$ predicate of $s$ do
            for $i = 1 \rightarrow NInt$ do
                Select randomly a mega-example $I$
                for $j = 1 \rightarrow NA$ do
                    Select randomly an atom $h$ from $I$ matching $schema(s)$
                    Bottom clause $BC \leftarrow$Saturation($h, rec, NS$), let $BC$ be $Head :- \ Body$
                    $Clauses \leftarrow [Head : 0.5 :- \ Body|Clauses]$
                end for
            end for
        end for
    end for
    return $Clauses$
end function
```

6.2.2 Tree generation

Since an HPLP can be mapped into a tree as described in Section 4, we create a tree whose nodes are literals appearing in the head or in the body of bottom clauses generated in the previous section. Every node in the tree share at least one variable with its parent. The tree is then converted into the initial HPLP, see Section 6.2.3.

To create the tree, Algorithm 9, starts by considering each Bottom clause in turn, line 3. Each bottom clause creates a 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 $b_i(Arg_i)$ (with arguments 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 chosen, the algorithm tries to insert the literal in the tree, see Algorithm 10. If $b_i(Arg_i)$ cannot be inserted, it is set as the right-most child of the root. This proceeds until all the $b_i(Arg_i)$ are inserted into a tree, lines 6–10. Then the resulted tree is appended into a list of trees (initially empty), line 11, and the list is merged obtaining a unique tree, 13. The trees in L are merged by unifying the arguments of the roots.

To insert the literal, $b_i(Arg_i)$, into the tree, Algorithm 10, nodes in the tree are visited in 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 4 Consider the following bottom clause from the UWCSE dataset:
$advised\_by(A, B) :-$
$student(A), professor(B), has\_position(B, C),$
$publication(D, B), publication(D, E), in\_phase(A, F),$
$taught\_by(G, E, H), ta(I, J, H).$

Algorithm 9 GENERATE TREE

```
function GENTREE($Bottoms$)
    $L \leftarrow []$
    for all Bottom in Bottoms do
        let $Bottom$ be $r(Arg) :- b_1(Arg_1), \ldots, b_m(Arg_m)$
        $Tree \leftarrow r(Arg)$                    ▷ $r(Arg)$ 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
        $L \leftarrow L \cdot append(Tree)$
    end for
    $final\_Tree \leftarrow mergeTrees(L)$
    return $final\_Tree$
end function
```

Algorithm 10 INSERT A LITERAL INTO A TREE

```
function INSERTTREE($Tree$,$b_i(Arg_i)$)
    if Tree=NULL then                    ▷ All nodes are visited
        return $False$
    else
        let $Tree$ be $b(Arg)$
        if $shareArgument(Arg, Arg_i)$ then
            addChild($Tree$,$b_i(Arg_i)$)
            return $True$
        else
            for all $Child$ of $Tree$ do
                return INSERTTREE($Child, b_i(Arg_i)$)
            end for
        end if
    end if
end function
```

In order to build the tree, $advised\_by(A, B)$ is initially set as the root of the tree. Then predicates in the body are considered in turn. The predicates $student(A)$ (resp. $professor(B)$, $hasposition(B, C)$ and $publication(D, B)$) are set as the children of $advised\_by(A, B)$ because their arguments share variable A (resp. B). Then the predicate $publication(D, E)$ is set as a child of $publication(D, B)$ because their arguments share variable D, $in\_phase(A, F)$ as a child of $advised\_by(A, B)$ (they share variable A), $taughtby(G, E, H)$ as a child of $publication(D, E)$ (they share variable E) and finally $ta(I, J, H)$ as a child of $taughtby(G, E, H)$ (they share variable H). The corresponding tree is shown in Figure 7.

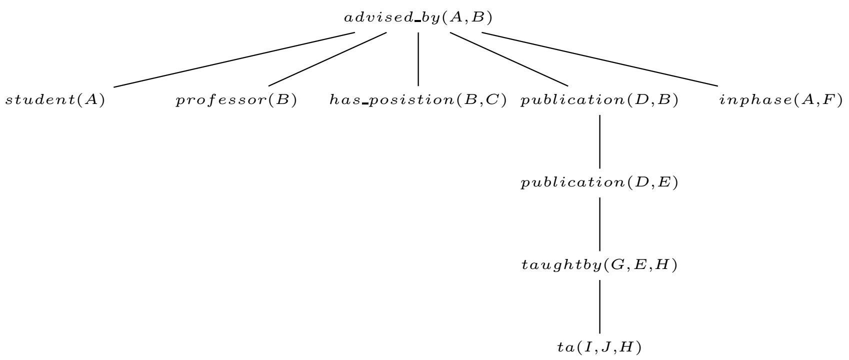

Fig. 7: Tree created from the bottom clause of Example 4.

6.2.3 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 computation purpose, we considered clauses with at most two literals in the body. This can be extended to any number of literals. Algorithm 11 takes as input the tree, *Tree*, an initial probability, $0 \leq MaxProb \leq 1$, a rate, $0 \leq rate \leq 1$, and the number of layers $1 \leq NumLayer \leq height(Tree)$ of HPLP we are about to generate. Let ($\mathbf{X}_k = X_1, \cdots, X_k$, $\mathbf{Y}_l = Y_1, \cdots, Y_l$, $\mathbf{Z}_t = Z_1, \cdots, Z_t$ and $\mathbf{W}_m = W_1, \cdots, W_m$) 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*, 1.0 by default. The new value of *Prob* at each level is $Prob * rate$ where $rate \in [0, 1]$ is a constant value, 0.95 by default. Thus the deeper the level, the lower the probability value. Supposing that n_i is visited, two cases can occur: n_i is a leaf or an internal node.

If $n_i = b_i(\mathbf{Y}_{l_i})$ 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

$$C = r(\mathbf{X}_k) : 0.5 :- \ b_i(\mathbf{Y}_{l_i}).$$

is generated, lines 9-11. Otherwise

$$C = hidden_{path}(\mathbf{Z}_{t_i}) : 0.5 :- \ b_{path\_i}(\mathbf{Y}_{l_i}).$$

is generated. $hidden_{path}(\mathbf{Z}_{t_i})$ is the hidden predicate associated with $Parent_i$ and *path* is the path from the root to $Parent_i$, lines 13-16.

If $n_i = b_i(\mathbf{Y}_{l_i})$ is an internal node having parent $Parent_i$, we consider two cases. If $Parent_i$ is the root, the clause

$$C = r(\mathbf{X}_k) : 0.5 :- \ b_i(\mathbf{Y}_{l_i}), hidden\_i(\mathbf{Z}_{t_i}).$$

is generated. $hidden\_i(\mathbf{Z}_{t_i})$ is associate with $b\_i(\mathbf{Y}_{l_i})$ and $\mathbf{Z}_{t_i} = \mathbf{X}_k \cup \mathbf{Y}_{l_i}$, lines 20-21. If $Parent_i$ is an internal node with the associated hidden predicate $hidden_{path}(\mathbf{Z}_t)$ then the clause

$$C = hidden_{path}(\mathbf{Z}_t) : 0.5 :- \; b_{path\_i}(\mathbf{Y}_{l_i}), hidden_{path\_i}(\mathbf{W}_{m_i}).$$

is generated where $\mathbf{W}_{m_i} = \mathbf{Z}_t \cup \mathbf{Y}_{l_i}$, lines 24-25.

The generated clause C is added into 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, a set of clauses (without hidden predicates in the body) that are renaming of each other are reduced to a single representative of the set. Note that, atoms in the head of the generated clauses are all annotated with probability 0.5 for exposition purposes. These values are replaced by random values between 0 and 1 at the beginning the parameter learning process, see Algorithm 7 line 5.

Example 5 The HPLP generated from the tree of Figure 7 is:
$advised\_by(A, B) : 0.5 :- \; student(A).$
$advised\_by(A, B) : 0.5 :- \; professor(B).$
$advised\_by(A, B) : 0.5 :- \; has\_position(B, C).$
$advised\_by(A, B) : 0.5 :- \; publication(D, B), hidden_1(A, B, D).$
$advised\_by(A, B) : 0.5 :- \; in\_phase(A, E).$
$hidden_1(A, B, D) : 0.5 :- \; publication(D, F), hidden_{1\_1}(A, B, D, F).$
$hidden_{1\_1}(A, B, D, F) : 0.5 :- \; taught\_by(G, F, H),$
$hidden_{1\_1\_1}(A, B, D, F, G, H).$
$hidden_{1\_1\_1}(A, B, D, F, G, H) : 0.5 :- \; ta(I, J, H).$

7 Related Work

PHIL is related to ProbLog2, [19] which is an algorithm for learning probabilistic Logic Programs parameters using Expectation Maximization. PHIL and ProbLog2 differ from the Probabilistic language they use. While PHIL uses *hierachical* PLP [39] which is a restriction of *Logic Programs with Annotated Disjunctions* (LPADs) [64], ProbLog2 uses an extended version of the ProbLog language in which both facts and rules can be annotated with probabilities. To perform inference, PHIL converts a program into ACs and evaluates the ACs bottom-up. ProbLog2 converts a program into a weighted boolean formulas (WBFs). The WBFs are then converted into deterministic-decomposable negation normal forms (d-DNNFs), see Darwiche [13], which are in turn converted into arithmetics circuits for performing Weighted Model Counting (WMC) [56]. ProbLog2 performs parameter learning using EM on top of the circuits.

Algorithm 11 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 $n_i$ be $b_i(\mathbf{Y}_{l_i})$ and $Parent_i$ be $r(\mathbf{X}_k)$
                        $C \leftarrow r(\mathbf{X}_k) : 0.5 :- b_i(\mathbf{Y}_{l_i})$.
                    else
                        let $Parent_i$ be $b_{path}(\mathbf{X}_k)$
                        $hidden_{path}(\mathbf{Z}_{t_i})$ is associated with $b_{path}(\mathbf{X}_k)$
                        $n_i \leftarrow b_{path\_i}(\mathbf{Y}_{l_i})$
                        $C \leftarrow hidden_{path}(\mathbf{Z}_{t_i}) : 0.5 :- b_{path\_i}(\mathbf{Y}_{l_i})$.
                    end if
                else                                                          ▷ $n_i$ is an internal node
                    if $Parent_i$ is the root node then
                        let $n_i$ be $\leftarrow b_i(\mathbf{Y}_{l_i})$ and $\mathbf{Z}_{t_i}$ be $\mathbf{X}_k \cup \mathbf{Y}_{l_i}$
                        $C \leftarrow r(\mathbf{X}_k) : 0.5 :- b_i(\mathbf{Y}_{l_i}), hidden\_i(\mathbf{Z}_{t_i})$.
                    else
                        let $Parent$ be $b_{path}(X_k)$
                        $n_i$ be $b_{path\_i}(\mathbf{Y}_{l_i})$ and $\mathbf{W}_{m_i}$ be $\mathbf{Z}_t \cup \mathbf{Y}_{l_i}$
                        $C \leftarrow hidden_{path}(\mathbf{Z}_t) : 0.5 :- b_{path\_i}(\mathbf{Y}_{l_i}), hidden_{path\_i}(\mathbf{W}_{m_i})$.
                    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
```

PHIL related to EMBLEM (Expectation maximization over binary decision diagrams for probabilistic logic programs [6]) which is an algorithm for learning PLP parameters by applying EM over Binary decision diagram, [1].

EMPHIL, ProbLog2 and EMBLEM are strongly related as they all apply the EM algorithm. The main difference among these languages is the computation cost for compiling the program. In EMBLEM and ProbLog2, the generation of BDDs and ACs respectively is #P in the number of ground clauses while the ACs generation in HPLP is linear in the number of ground clauses. Once the BDDs for EMBLEM and the ACs for HPLP and ProbLog2 are generated, inference is linear in the number of nodes in BDDs or circuits.

PHIL is also related to Tuffy [43] which is a machine learning algorithm that performs MAP/Marginal inference and parameter learning on Markov Logic Networks (MLNs). Differently from PHIL which uses a top-down approach to perform the groundings of clauses (as in ProLog), Tuffy uses a bottom-up approach. In fact, Tuffy expresses grounding as a sequence of SQL queries similarly to waht is done in Datalog. Indeed, it relies on a RDBMS such as PostgreSQL [17]. Each predicate in the input MLN is associated with a relation in the RDBMS where each row represents a ground atom. Given a MLN query (which could be a conjunction of predicates), Tuffy produces a SQL query that joins together the relations corresponding to the predicates in MLN to produce the atom of the ground clauses. While PHIL relies on ACs, generated from a PLP, to perform inference, Tuffy relies on the WalkSAT algorithm [24]. Two algorithms to perform inference are available in the Tuffy systems: the default WalkSAT and a modified version of MAxSAT [48] algorithm called SweepSAT [43]. To learn Markov logic networks weights, Tuffy implements the Diagonal Newton (DN) algorithm described in [32]. The DN algorithm is a gradient descent-based algorithm which relies on the Hessian matrix to update the weights. In fact at each iteration DN uses the inverse of the diagonalized Hessian to compute the gradient. Tuffy's parameter learning algorithm is related more to DPHIL than EMPHIL since both are based on gradient descent.

SLEAHP is related to SLIPCASE [5], SLIPCOVER [7] which are algorithms for learning general PLP, and LIFTCOVER, [41] which is an algorithm for learning liftable PLP, where the head of all the clauses in the program share the same predicate (the target predicate) and their bodies contain only input predicates. Therefore, Liftable PLP can be seen as a restriction of HPLP where hidden predicates and clauses are not allowed. 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 use EMBLEM to tune the parameters and compute the log-likehood of the data and LIFTCOVER uses a Gradient descent or EM-based method.

SLEAHP is also related to PROBFOIL+ [47] which is a generalization of mFOIL [18]. PROBFOIL+ learns both the structure and the parameters of ProbLog programs by performing a hill climbing search in the space of Programs. It consists of a covering loop in which one rule is added to the program at each iteration. The covering loop ends when a condition based on a global scoring function is satisfied. The rule added at each iteration is obtained by a nested loop which iteratively adds literals to the body of the rule performing a beam search in the space of clauses (as in mFOIL) guided by a local scoring function.

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 to tune the parameters and discard irrelevant rules.

SLEAHP finds prominent bottom clauses from which a tree of literals is built. To generate a bottom clause, SLEAHP randomly selects a mega-example and randomly selects in it an atom for the target predicate. Then, bottom clauses are buit as in Progol, [37]. Much work exists on using stochastic components for learning the structure of logic programs, see [66,68,67]. In [66] and [68], the authors present an empirical study of randomised restarted search in Inductive Logic Programming. Specifically, a search in the clause subsumption lattice is performed by applying different strategies. These strategies all include restarts and differ mainly on the starting clause and the refinement operation. Both algorithms described in [66] and SLEAHP saturate the bottom clauses as in Progol, see [37]. Differently from [66], SLEAHP generates a tree of literals from the bottom clauses instead of performing clause refinements. In [67] the authors propose an algorithm called Randomized Rapid Restarts (RRR). To randomize the lattice search of clauses, the algorithm begins by randomly selecting a starting clause, then it generates deterministic refinements through a non traditional refinement operator producing a radial lattice, and finally it returns the first clause satisfying conditions on minimal accuracy. This operation is done for a maximum number of restarts.

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 [12] and [11], 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 the focus here is on learning probabilistic classifiers, i.e., functions returning the class of an individual given what is known about him.

SLEAHP performs structure learning of HPLP by parameter learning, i.e, it initially generates a large HPLP and performs a regularized parameter learning on it. Then, clauses providing no contribution are pruned, typically those with low values of probabilities. Learning the structure of models by exploiting the regularization of parameter learning has been successfully investigated e.g. in [30,23]. In [30], the authors present an algorithm for learning the structure of log-linear graphical model, e.g Markov Networks (MNs), using L_1-regularization. The algorithm starts with a reasonable number of features and progressively introduces new features into the model letting the L_1-regularization scheme discard features providing no contribution, i.e. those with small weight, via an optimization process. In [23], the authors present an algorithm named OSL for learning the structure of MLNs by exploiting parameter estimation. OSL starts with a (possibly) empty set of clauses and, at each step, new clauses are added in order to improve the prediction of the model. Similarly to [30], regularization is applied to the new set of clauses to discard clauses which are not useful in the long run. SLEAHP and algorithms described in [30,23] all use regularization in parameter learning, i.e. L_1 and L_2 for SLEAHP and L_1 for the others, to perform structure learning. But differently from [30,23], which apply the regularization at each step, SLEAHP applies the regularization once on a large HPLP. This can be done since inference in HPLP is cheaper.

To perform structure learning, SLEAHP learns the parameters of an AC obtained from an initially large HPLP. Learning the structure of (graphical) models by exploiting ACs has also been explored in the literature, see [33, 55]. In [33] the authors describe an algorithm, named ACMN, which learns the structure of MNs using ACs. ACMN performs a greedy search in the structure space. It initially considers a MN structure which includes all single-variable features. The MN is then compiled into an AC and the algorithm gradually updates the AC, by splitting features, without recompiling it from scratch. In [55] the authors present an algorithm called DACLearn which discriminatively learns an AC which represents a conditional distribution. DACLearn searches in the combinatorial space of conjunctive features by greedily selecting features that increase the conditional likelihood. While SLEAHP, at each step, updates the parameters of the AC (i.e of the leaves), both ACMN and DACLearn update the AC that represents the model by adding features. The ACs learned in SLEAHP and DACLearn represent a conditional distribution instead of a full joint distribution as the one modelled by ACMN. In ACMN and DACLearn the AC is composed of sums and products while in SLEAHP the AC is composed of *probabilistic sums*, (Equation 2) and products.

Ground HPLPs can also be 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 [16] for an excellent review. For example, Relational Neural Networks (RelNNs) [25] generalize Relational Logistic Regression (RLR) by stacking multiple RLR layers together. The authors provide strong motivations for having multiple layers, highlighting in particular that the layers improve the representation power by allowing aggregation at different levels and on different object populations. HPLPs benefit from the same advantage. Moreover, HPLPs keep a semantics as probabilistic logic programs: the output of the network is a probability according to the distribution semantics.

In [59] 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 groundings for 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. In this paper we show that 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.

8 Experiments

PHIL[2] has been implemented in SWI-Prolog [65] and C as programming languages. It can be installed using *pack_install(phil)* on SWI-Prolog. Experiments[3] were performed 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 and structure learning algorithms including ProbLog2 [19], EMBLEM [6], Tuffy [43], SLIPCOVER [7], PROBFOIL+ [47], MLN-BC, MLN-BT [26] and RDN-B [38]. While SLIPCOVER learns Logic Programs with Annotated Disjunctions PROBFOIL+ learns ProbLog programs as described in Section 7.

MLN-BC and MLN-BT learn Markov Logic Networks (MLNs) considering a series of relational functional approximation problems. Two kinds of representations for the gradients on the pseudo-likelihood are used: clause-based (MLN-BC) and tree-based (MLN-BT). At each gradient step, MLN-BC simply learns a set of Horn clauses with an associated regression value, while MLN-BT consider MLNs as a set of relational regression trees, in which each path from the root to a leaf can be seen as a clause and the regression values in the leaves are the clause weights. The goal is to minimize the squared error between the potential function and the functional gradient over all training examples.

RDN-B learns Relational Dependency Networks (RDNs) considering a series of relational function approximation problems using Friedman's functional gradient-based boosting. RDN-B approximates the joint distribution of a relational model to a product of conditional distributions over ground atoms. It considers the conditional probability distribution of each predicate as a set of relational regression trees each of which approximates the corresponding gradient. These regression trees serve as the individual components of the final potential function. They are learned such that at each iteration the new set of regression trees aims at maximizing the likelihood. The different regression trees provide the structure of the conditional distributions while the regression values at the leaves form the parameters of the distributions.

Section 8.1 describes the various datasets used in the experiments. Then in Section 8.2, PHIL (DPHIL and EMPHIL) and its regularized versions was compared with ProbLog2 and EMBLEM. Finally, Section 8.4 presents experiments comparing SLEAHP with SLIPCOVER, PROBFOIL+, MLN-BC, MLN-BT and RDN-B. Moreover, we provide phil in cplint on SWISH[4] which

[2] The code is available at `https://github.com/ArnaudFadja/phil`.

[3] Experiments are available at `https://bitbucket.org/ArnaudFadja/hierarchicalplp_experiments/src/master/` .

[4] phil on SWISH is available at `http://cplint.eu/e/phil/phil_examples.swinb`. The manual is available at `https://arnaudfadja.github.io/phil`

is a web application for reasoning and learning with probabilistic logic programs. In the application, the user can write hierarchical PLPs and perform inference, parameter and structure learning without installing phil locally.

8.1 Datasets

The proposed systems were experimented on five well known datasets:

The Mutagenesis dataset [61] 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 in the compound, a bond of type *bondtype* can be found between the atoms $atom1$ and $atom2$. The goal is to predict the mutagenicity of the 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 [60] is similar to Mutagenesis dataset whose objective is to predict the carcinogenicity of molecular.

The Mondial dataset [34] contains data from multiple geographical web data sources. The goal is to predict the religion of a country as Christian (target predicate $christian\_religion(A)$).

The UWCSE dataset [4] contains information about the Computer Science department of the University of Washington. The goal is to predict the target predicate $advisedby(A, B)$ expressing that a student (A) is advised by a professor (B).

In the Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD15) dataset [5] the task is to predict gender of a client based on E-commerce data. The dataset was also used in [25].

8.2 Experiments: Parameter learning

In this section, we present experiments comparing PHIL with EMBLEM and ProbLog2 on the five datasets. We manually build the initial HPLPs for the datasets UWCSE and PAKDD15 and generate those for Mutagenesis[6], Carcinogenesis and Mondial using SLEAHP. The following hyper-parameters were used:

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 ProbLog2. 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*

[5] A full description is available at `https://knowledgepit.ml/pakdd15-data-mining-competition/`

[6] An online version is available at `http://cplint.eu/e/phil/muta.pl`

gradient descent with batch size $BatchSize = 100$. In the regularized version of PHIL, $\gamma = 10$ is used as regularization coefficient and clauses with parameters less than $MinProb = 10^{-5}$ are removed. We experiment with three versions of EMPHIL$_B$ (EMPHIL$_{B_1}$, EMPHIL$_{B_2}$, EMPHIL$_{B_3}$) which use $a = 0$ and differ from the fraction of the examples n they use at the M-step. They respectively use $b = \frac{n}{10}$, $b = \frac{n}{5}$, $b = \frac{n}{4}$. 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. 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 1.

Table 1: Dataset characteristics.

Characteristic	Mutagenesis	Carcinogenesis	Mondial	UWCSE	PAKDD15
NC	58	38	10	17	11
NL	9	7	6	8	4
NF	10	1	5	5	10
NAC	169.2	298	176	3353.6	3000

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 [14]. The average values (over the folds) of the areas for each algorithm are shown in Tables 2, 3. In these tables, the best values are highlighted in bold. Table 4 shows the average training time. Note that EMBLEM ran out of memory on the datasets Carcinogenesis and Mondial and we could not compute the areas and the training time. Moreover, to start learning, ProbLog2 needed more memory. Tuffy ran out of memory on all datasets. We found it was due to the fact that, before performing inference/parameter learning, Tuffy maps all predicates and ground clauses into relations in a PostgreSQL database as explained in Section 7. Since all programs have hidden predicates which are not included in the input predicates, Tuffy attempts to populate the database with all possible groundings of the hidden predicates which requires much memory. In all datasets, Tuffy ran out of 500 Giga hard disk memory available.

Table 2: Average area under ROC curve for parameter learning.

AUCROC	Mutagenesis	Carcinogenesis	Mondial	UWCSE	PAKDD15
DPHIL	**0.888943**	0.602632	0.531157	0.941525	0.506362
DPHIL$_1$	0.841021	0.571053	0.534817	0.960876	0.504912
DPHIL$_2$	0.880465	0.618421	0.534563	0.949548	0.514218
EMPHIL	0.885358	**0.684211**	0.534822	0.968560	0.504742
EMPHIL$_1$	0.884016	**0.684211**	0.536009	0.938121	0.504741
EMPHIL$_2$	0.885478	0.623684	0.534622	**0.969046**	0.504742
EMPHIL$_{B_1}$	0.833539	0.619730	0.536042	0.930243	0.504196
EMPHIL$_{B_2}$	0.821356	0.640780	**0.537011**	0.930243	0.504196
EMPHIL$_{B_3}$	0.820220	0.640780	0.534996	0.930243	0.475363
EMBLEM	0.887695	-	-	0.968354	0.501014
ProbLog2	0.828655	0.594737	0.533905	0.968909	0.500000
Tuffy	-	-	-	-	-

Table 3: Average area under PR curve for parameter learning.

AUCPR	Mutagenesis	Carcinogenesis	Mondial	UWCSE	PAKDD15
DPHIL	**0.947100**	0.595144	0.138932	0.227438	0.223219
DPHIL$_1$	0.886598	0.563875	0.142331	0.191302	0.222783
DPHIL$_2$	0.929244	0.580041	**0.147390**	0.219806	0.218153
EMPHIL	0.944511	0.679966	0.142374	0.277760	0.222618
EMPHIL$_1$	0.944758	**0.679712**	0.142696	0.275985	0.222609
EMPHIL$_2$	0.944517	0.655781	0.142066	**0.307713**	0.222618
EMPHIL$_{B_1}$	0.880013	0.649090	0.142810	0.261578	0.222043
EMPHIL$_{B_2}$	0.868837	0.641630	0.143275	0.261578	0.222043
EMPHIL$_{B_3}$	0.867759	0.641630	0.142540	0.261578	0.250876
EMBLEM	0.944394	-	-	0.262565	0.231962
ProbLog2	0.901450	0.568821	0.132498	0.306378	0.220833
Tuffy	-	-	-	-	-

Table 4: Average time for parameter learning

Time	Mutagenesis	Carcinogenesis	Mondial	UWCSE	PAKDD15
DPHIL	*2.8573*	178.2680	265.4160	**0.0884**	34.8170
DPHIL$_1$	5.2059	177.4270	311.3280	0.2910	20.5704
DPHIL$_2$	5.4450	88.5100	301.1392	0.2214	4.9517
EMPHIL	4.4430	*106.5540*	270.6688	0.2890	6.6106
EMPHIL$_1$	4.8940	*181.0890*	317.4202	1.0000	7.0691
EMPHIL$_2$	5.0046	146.8440	**245.3830**	*0.8372*	6.6334
EMPHIL$_{B_1}$	2.6478	85.3210	248.1978	0.1572	6.0990
EMPHIL$_{B_2}$	**1.5820**	**80.4270**	*261.3612*	0.1266	8.8722
EMPHIL$_{B_3}$	1.4937	94.6850	274.9598	0.1190	6.0985
EMBLEM	125.62	-	-	0.9666	154.51
ProbLog2	722.00	38685.00	1607.60	161.40	596.90
Tuffy	-	-	-	-	-

8.2.1 Discussion of the Results for Parameter Learning.

The experiments show that PHIL beats EMBLEM and ProbLog2 either in terms of area or in terms of time in all datasets. In Table 4, we highlight in italic the times associated with the best accuracies from Tables 2 and 3. 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 like the Mondial dataset. Between DPHIL and EMPHIL, DPHIL is often convenient in dataset with many clauses and examples (see Mutagenesis).

Tables 2 and 3 present several values that are very close to each other both for PHIL systems and the other systems (EMBLEM and ProbLog2). This probably means that EMBLEM, ProbLog2, PHIL and their regularized versions return solutions of the same quality. To verify this assertion, a statistical significance test (t-test) was performed among these systems. Tables showing the p-values are in Supplementary Material and in the Experiments repository [3]. We compute the t-test between the fold values in datasets splitting in more than 2 folds i.e in Mutagenesis (10 folds), UWCSE (5 folds) and Mondial (5 folds). We use a paired two-tailed t-test. The test was done w.r.t the AUCROC (highlighted in yellow) and w.r.t to the AUCPR (highlighted in green). In the tables showing the p-values, EMPHIL_B refers to EMPHIL_{B_1}.

The tests show that PHIL systems are statistically equivalent both in terms of AUCROC and in terms of AUCPR in all dataset except in Mutagenesis in which EMPHIL_B and DPHIL_1 are almost always significantly worse than the others. Indeed, in Mutagenesis, EMPHIL_B is always significantly worse than the other systems ($p \leq 0.009$) for both AUCROC and AUCPR except for DPHIL_1 where $p = 0.15$ and $p = 0.19$ for AUCROC and AUCPR respectively. The same situation can be observed for DPHIL_1 ($p \leq 0.01$) with the other systems.

EMBLEM, on the Mutagenis dataset, is statistically equivalent to EMPHIL and its regularized versions both in terms of AUCROC and in terms of AUCPR. This does not happen in the gradient versions of PHIL except for DHPIL. This situation is clearly understandable since EM versions of PHIL and EMBLEM all rely on the EM algorithm. They compute the same expectations. The former on a factor graph and the latter on a Binary Decision Diagram.

ProbLog is not statistically equivalent to the other systems except in the Mondial dataset in which there is a possible equivalence with other systems both in terms of AUCROC and in terms of AUCPR.

8.3 Experiments: Structure learning

In this section, we present experiments comparing SLEAHP with state-of-the art structure learning systems on the five datasets described in Section 8.1. We compared SLEAHP with PLP systems such as SLIPCOVER [7] and PROBFOIL+ [47]. We also compared SLEAHP with Statistical Relational Learning methods such as MLN-BC and MLN-BT [26] for learning Markov Logic Networks (MLNs) and with RDN-B [38] for learning Relational Dependency Networks. Note that the dataset PAKDD15 was randomly divided into 10 folds and the same language bias expressed in terms of modes was used in all systems. 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 8.2 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$_B$) uses EMPHIL$_1$ (resp. EMPHIL$_2$ and EMPHIL$_{B_1}$). The average area under the ROC/PR curves and the average time are shown in Tables 5, 6, 7 respectively. The results indicated with - for PROBFOIL+ mean that it was not able to terminate in 24 hours, in Mondial on some folds, in UWCSE and PAKDD15 on all folds. The results with * for MLN-BC indicates that the algorithm did not learn any theory.

Table 5: Average area under the ROC curve for structure learning.

AUCROC	Mutagenesis	Carcinogenesis	Mondial	UWCSE	PAKDD15
SLEAHP$_{G_1}$	0.889676	0.493421	0.483865	0.936160	**0.506252**
SLEAHP$_{G_2}$	0.845452	0.544737	0.472843	0.925436	0.506242
SLEAHP$_{E_1}$	0.878727	0.660526	0.433016	0.907789	0.503251
SLEAHP$_{E_2}$	0.904933	0.414135	0.483798	0.904347	0.505691
SLEAHP$_B$	0.822833	0.618421	0.464058	0.925099	0.504196
SLIPCOVER	0.851000	**0.676000**	0.600000	0.919000	0.500000
PROBFOIL+	0.881255	0.556578	-	-	-
MLN-BC	0.543847	0.561842	0.394252	0.912803	*
MLN-BT	0.841584	0.410526	0.590406	**0.961906**	0.500000
RDN-B	**0.925142**	0.521053	**0.697147**	0.958913	0.500000

Table 6: Average area under the PR curve for structure learning.

AUCPR	Mutagenesis	Carcinogenesis	Mondial	UWCSE	PAKDD15
SLEAHP$_{G_1}$	0.929906	0.498091	0.701244	0.148115	0.223074
SLEAHP$_{G_2}$	0.918519	0.502135	0.690782	0.131750	0.223028
SLEAHP$_{E_1}$	0.948563	0.598095	0.632270	0.059562	0.220226
SLEAHP$_{E_2}$	**0.955678**	0.540510	0.623542	0.069861	**0.225017**
SLEAHP$_B$	0.900300	0.552477	0.623542	0.059655	0.222043
SLIPCOVER	0.885000	**0.600000**	0.733792	0.113000	0.220833
PROBFOIL+	0.937497	0.534393	-	-	-
MLN-BC	0.686894	0.581305	0.631976	0.058467	*
MLN-BT	0.894080	0.506972	0.709318	0.1833378	0.220833
RDN-B	0.953618	0.577169	**0.768584**	**0.266790**	0.220833

Table 7: Average time for structure learning.

Time	Mutagenesis	Carcinogenesis	Mondial	UWCSE	PAKDD15
SLEAHP$_{G_1}$	41.8250	**48.7600**	*59.5054*	*219.6410*	192.4396
SLEAHP$_{G_2}$	47.1344	10524.0900	**14.0470**	*194.9706*	*162.9938*
SLEAHP$_{E_1}$	48.0152	*303.0570*	60.8316	387.6650	151.7217
SLEAHP$_{E_2}$	*45.9245*	92.3820	61.0996	312.2604	68.2432
SLEAHP$_B$	**13.1478**	1399.0090	14.6698	295.6734	**61.5347**
SLIPCOVER	74610.7000	17419.4500	650.3630	**141.3600**	242.7077
PROBFOIL+	1726.6000	15433.0000	-	-	-
MLN-BC	91.7700	59.73500	56.5007	376.2356	*
MLN-BT	360.8563	87.5020	53.5568	891.4226	915.5794
RDN-B	183.7140	61.5000	192.0487	501.1176	793.7661

8.3.1 Discussion of the Results for Structure Learning

In terms of quality of the solutions, SLEAHP outperforms SLIPCOVER in the three datasets Mutagenesis, UWCSE and PAKDD15, and achieves solution of similar quality in the other datasets. Note that SLEAHP$_{E_1}$ and other EM regularizations do not perform well, in terms of AUCPR, on the UWCSE dataset. On the other hand, on the same dataset, SLEAHP$_{G_1}$ and SLEAHP$_{G_2}$ outperform SLIPCOVER in terms of accuracy and time. This motivates 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 7 we also highlight in italic, as done in Table 4, the times associated with the best accuracies of SLEAHP from Tables 5 and 6. Between DPHIL and EMPHIL regularizations, as stated before, DPHIL is often preferred in dataset with large examples (see UWCSE).

With respect to PROBFOIL+, SLEAHP outperforms PROBFOIL+ in terms of solution quality and time in all datasets.

With respect to MLN-BT/BC systems, SLEAHP systems beat them in all datasets except in UWCSE in which they provide similar solutions of quality. In terms of AUCROC, SLEAHP beats RDN-B in the Carcinogenis dataset and provides similar quality in the other datasets. In terms of AUCPR, SLEAHP beats RDN-B in three out of five datasets including Mutagenesis, Carcinogenesis and PAKDD15. In the other datasets it provides similar solutions of quality.

In terms of time, SLEAHP systems clearly outperform the other systems in all datasets except in UWCSE where it provides a learning time similar to that of SLIPCOVER.

We also performed a statistical significance test for structure learning as done for parameter learning. The p-values for each couple of systems are shown in the supplementary material and in the Experiments repository [3]. Also in this case, it can be observed that SLEAHP systems are statistically equivalent both in terms of AUCROC and in terms of AUCPR in almost all datasets except in Mutagenesis in which SLEAHP$_B$ is always significantly worse than SLEAHP$_{E_1}$ and SLEAHP$_{E_2}$ ($p < 0.05$) for both AUCROC and

AUCPR. In UWCSE and Mondial, there is only one significance difference, that of SLEAHP$_{G_1}$ with the other systems. SLIPCOVER and PROBFOIL+ in all datasets, except in UWCSE and Mondial, are almost always statistically equivalent to SLEAHP sytems both in terms of AUCROC and in terms of AUCPR. This situation can be clearly observed in the Mutagenesis dataset. PROBFOIL+, in the Mutagenesis dataset, is almost always statistically equivalent to SLIPCOVER and SLEAHP systems both in terms of AUCROC and in terms of AUCPR. It can be also observed that in all datasets, except in Mutagenesis, MLN-BC/BT and RDN-B systems are not statistically equivalent to PLP systems (SLEAHP, SLIPCOVER and ProbFoil) both in terms of AUCROC and in terms of AUCPR. Overall, SLEAHP systems are almost alway statistically equivalent among themselves and can be used interchangeably. However, stochastic gradient descent-based systems are faster in dataset with a large number of ACs (e.g UWCSE).

To summarize, SLEAHP beats other systems in terms of computation time in almost all datasets and achieves a similar quality of the solution. We believe that as the dimension of data increases the gap between SLEAHP learning time and the other systems learning time would be strongly remarkable. This would make SLEAHP a good compromise between accuracy and learning time.

8.4 Comparing PHIL and SLEAHP

The performance, either in terms of AUCROC, AUCPR and time was recapped for both PHIL and SLEAHP in the same tables in order to highlight a possible benefit of learning the structure of HPLPs instead of manually building them. The different tables are available in the appendix and in the Experiments repository] [3]. From Tables 8 and 9, it can be clearly observed that SLEAHP provides similar and often better performance in terms of AUCROC and AUCPR in all datasets. In the Mondial dataset, in particular, SLEAHP outperforms PHIL in terms of AUCPR. In terms of time (see Table 10) SLEAHP drastically outperforms PHIL in all datasets except in PAKDD15. This is mainly due to the fact that SLEAHP generates the initial hierarchical PLP with a depth that leads to a good compromise between solution quality and learning time. In the PAKDD15 dataset, PHIL learning time outperforms SLEAHP learning time because the programs generated by SLEAHP were remarkably deeper than the one manually built used with PHIL. These observations show the usefulness of performing structure learning with SLEAHP. Having a reasonable tradeoff between solution quality and learning time overcomes the tedious process of manually writing the structure of a program for each dataset.

9 Conclusion

In this work we have presented different algorithms for learning both the structure and the parameters of *hierarchical* PLP from data. We first presented

PHIL, Parameter learning for HIerarchical probabilistic Logic programs, that learns the parameters of *hierarchical* PLP from data. Two versions of PHIL have been presented: DPHIL which learns the parameters by applying gradient descent and EMPHIL which applies Expectation Maximization. Different regularized versions of DPHIL (DPHIL$_1$, DPHIL$_2$) and EMPHIL(EMPHIL$_1$, EMPHIL$_2$, EMPHIL$_B$) have also been proposed. These regularizations favor small parameters during learning. Then, we proposed SLEAHP which learns both the structure and the parameters of *hierarchical* PLP from data. SLEAHP initially generates a large $HPLP$ (containing many clauses) and applies a regularized version of PHIL to perform parameter learning. Clauses with a small parameters are removed from the final program.

Finally we performed experiments comparing, in five real datasets, PHIL with ProbLog2 and EMBLEM and SLEAHP with SLIPCOVER, PROBFOIL+, MLN-BC, MLN-BT and RDN-B. PHIL and SLEAHP achieve similar and often better accuracies in a shorter time. The significance tests performed among the different versions of PHIL and SLEAHP show that these versions are statistically equivalent and can be used interchangeably.

Regarding the restriction imposed on the number of literals in the body of clauses in SLEAHP, we plan to extend this number in order to explore a large space of HPLPs. We also plan to extend HPLPs in domains with continuous random variables in order to apply PHIL and SLEAHP to data such as images.

Acknowledgements This work was partly supported by the "National Group of Computing Science (GNCS-INDAM)".

References

1. Akers, S.B.: Binary decision diagrams. IEEE Trans. Comput. **27**(6), 509–516 (1978)
2. Alberti, M., Bellodi, E., Cota, G., Riguzzi, F., Zese, R.: `cplint` on SWISH: Probabilistic logical inference with a web browser. Intell. Artif. **11**(1), 47–64 (2017). DOI 10.3233/IA-170105
3. Alberti, M., Cota, G., Riguzzi, F., Zese, R.: Probabilistic logical inference on the web. In: G. Adorni, S. Cagnoni, M. Gori, M. Maratea (eds.) AI*IA 2016, *LNCS*, vol. 10037, pp. 351–363. Springer International Publishing (2016). DOI 10.1007/978-3-319-49130-1_26
4. Beerenwinkel, N., Rahnenführer, J., Däumer, M., Hoffmann, D., Kaiser, R., Selbig, J., Lengauer, T.: Learning multiple evolutionary pathways from cross-sectional data. Journal of Computational Biology **12**, 584–598 (2005)
5. Bellodi, E., Riguzzi, F.: Learning the structure of probabilistic logic programs. In: S. Muggleton, A. Tamaddoni-Nezhad, F. Lisi (eds.) 22nd International Conference on Inductive Logic Programming, *LNCS*, vol. 7207, pp. 61–75. Springer Berlin Heidelberg (2012)
6. Bellodi, E., Riguzzi, F.: Expectation maximization over binary decision diagrams for probabilistic logic programs. Intell. Data Anal. **17**(2), 343–363 (2013)
7. Bellodi, E., Riguzzi, F.: Structure learning of probabilistic logic programs by searching the clause space. Theor. Pract. Log. Prog. **15**(2), 169–212 (2015). DOI 10.1017/S1471068413000689
8. Bishop, C.: Pattern Recognition and Machine Learning. Information Science and Statistics. Springer (2016)
9. Clark, K.L.: Negation as failure. In: Logic and data bases, pp. 293–322. Springer (1978)

10. Cox, D.A.: Galois Theory. Pure and Applied Mathematics: A Wiley Series of Texts, Monographs and Tracts. John Wiley & Sons, Ltd. (2012)
11. Cropper, A., Morel, R., Muggleton, S.: Learning higher-order logic programs. Machine Learning **108**(7), 1063–1083 (2019). DOI https://doi.org/10.1007/s10994-019-05862-7. URL `https://doi.org/10.1007/s10994-019-05862-7`
12. Cropper, A., Muggleton, S.H.: Learning efficient logic programs. Machine Learning **108**(7), 1063–1083 (2019). DOI 10.1007/s10994-018-5712-6. URL `https://doi.org/10.1007/s10994-018-5712-6`
13. Darwiche, A.: New advances in compiling CNF into decomposable negation normal form. In: R.L. de Mántaras, L. Saitta (eds.) 16th European Conference on Artificial Intelligence (ECAI 20014), pp. 328–332. IOS Press (2004)
14. Davis, J., Goadrich, M.: The relationship between precision-recall and roc curves. In: Proceedings of the 23rd international conference on Machine learning, pp. 233–240. ACM (2006)
15. De Raedt, L., Kimmig, A., Toivonen, H.: ProbLog: A probabilistic Prolog and its application in link discovery. In: M.M. Veloso (ed.) IJCAI 2007, vol. 7, pp. 2462–2467. AAAI Press/IJCAI (2007)
16. De Raedt, L., Manhaeve, R., Dumancic, S., Demeester, T., Kimmig, A.: Neuro-symbolic= neural+ logical+ probabilistic. In: NeSy'19@ IJCAI, the 14th International Workshop on Neural-Symbolic Learning and Reasoning, pp. 1–4 (2019)
17. Drake, J.D., Worsley, J.C.: Practical PostgreSQL. " O'Reilly Media, Inc." (2002)
18. Džeroski, S.: Handling imperfect data in inductive logic programming. In: 4th Scandinavian Conference on Artificial Intelligence (SCAI 1993), pp. 111–125 (1993)
19. Fierens, D., Van den Broeck, G., Renkens, J., Shterionov, D.S., Gutmann, B., Thon, I., Janssens, G., De Raedt, L.: Inference and learning in probabilistic logic programs using weighted Boolean formulas. Theor. Pract. Log. Prog. **15**(3), 358–401 (2015)
20. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: 5th International Conference and Symposium on Logic Programming (ICLP/SLP 1988), vol. 88, pp. 1070–1080. MIT Press (1988)
21. Gerla, G.: Fuzzy Logic, *Trends in Logic*, vol. 11. Springer (2001). DOI 10.1007/978-94-015-9660-2_8
22. Hájek, P.: Metamathematics of fuzzy logic, vol. 4. Springer (1998)
23. Huynh, T.N., Mooney, R.J.: Online structure learning for markov logic networks. In: Joint European Conference on Machine Learning and Knowledge Discovery in Databases, pp. 81–96. Springer (2011)
24. Kautz, H.A., Selman, B., Jiang, Y.: A general stochastic approach to solving problems with hard and soft constraints. Satisfiability Problem: Theory and Applications **35**, 573–586 (1996)
25. Kazemi, S.M., Poole, D.: Relnn: A deep neural model for relational learning. In: Thirty-Second AAAI Conference on Artificial Intelligence (2018)
26. Khot, T., Natarajan, S., Kersting, K., Shavlik, J.: Learning markov logic networks via functional gradient boosting. In: 2011 IEEE 11th International Conference on Data Mining, pp. 320–329. IEEE (2011)
27. Kingma, D., Ba, J.: Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014)
28. Kok, S., Domingos, P.: Learning the structure of Markov Logic Networks. In: ICML 2005, pp. 441–448. ACM (2005)
29. Law, M., Russo, A., Broda, K.: Iterative learning of answer set programs from context dependent examples. arXiv preprint arXiv:1608.01946 (2016)
30. Lee, S.I., Ganapathi, V., Koller, D.: Efficient structure learning of markov networks using l_1-regularization. In: Advances in neural Information processing systems, pp. 817–824 (2007)
31. Li, H., Zhang, K., Jiang, T.: The regularized em algorithm. In: AAAI, pp. 807–812 (2005)
32. Lowd, D., Domingos, P.: Efficient weight learning for markov logic networks. In: European conference on principles of data mining and knowledge discovery, pp. 200–211. Springer (2007)
33. Lowd, D., Rooshenas, A.: Learning markov networks with arithmetic circuits. In: Artificial Intelligence and Statistics, pp. 406–414 (2013)

34. May, W.: Information extraction and integration: The mondial case study. Tech. rep., Universitat Freiburg, Institut für Informatik (1999)
35. Meert, W., Struyf, J., Blockeel, H.: CP-Logic theory inference with contextual variable elimination and comparison to BDD based inference methods. In: L. De Raedt (ed.) ILP 2009, *LNCS*, vol. 5989, pp. 96–109. Springer (2010). DOI 10.1007/978-3-642-13840-9_10
36. Mørk, S., Holmes, I.: Evaluating bacterial gene-finding hmm structures as probabilistic logic programs. Bioinformatics **28**(5), 636–642 (2012)
37. Muggleton, S.: Inverse entailment and Progol. New Generat. Comput. **13**, 245–286 (1995)
38. Natarajan, S., Khot, T., Kersting, K., Gutmann, B., Shavlik, J.: Gradient-based boosting for statistical relational learning: The relational dependency network case. Machine Learning **86**(1), 25–56 (2012)
39. Nguembang Fadja, A., Lamma, E., Riguzzi, F.: Deep probabilistic logic programming. In: C. Theil Have, R. Zese (eds.) PLP 2017, *CEUR-WS*, vol. 1916, pp. 3–14. Sun SITE Central Europe (2017)
40. Nguembang Fadja, A., Riguzzi, F.: Probabilistic logic programming in action. In: A. Holzinger, R. Goebel, M. Ferri, V. Palade (eds.) Towards Integrative Machine Learning and Knowledge Extraction, *LNCS*, vol. 10344. Springer (2017). DOI 10.1007/978-3-319-69775-8_5
41. Nguembang Fadja, A., Riguzzi, F.: Lifted discriminative learning of probabilistic logic programs. Machine Learning (2018). DOI 10.1007/s10994-018-5750-0. URL `https://doi.org/10.1007/s10994-018-5750-0`
42. Nguembang Fadja, A., Riguzzi, F., Lamma, E.: Expectation maximization in deep probabilistic logic programming. In: International Conference of the Italian Association for Artificial Intelligence, pp. 293–306. Springer (2018)
43. Niu, F., Ré, C., Doan, A., Shavlik, J.: Tuffy: Scaling up statistical inference in markov logic networks using an rdbms. arXiv preprint arXiv:1104.3216 (2011)
44. Pearl, J.: Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference. Morgan Kaufmann (1988)
45. Poole, D.: The Independent Choice Logic for modelling multiple agents under uncertainty. Artif. Intell. **94**, 7–56 (1997)
46. Przymusinski, T.C.: Every logic program has a natural stratification and an iterated least fixed point model. In: 8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS 1989), pp. 11–21. ACM Press (1989)
47. Raedt, L.D., Dries, A., Thon, I., den Broeck, G.V., Verbeke, M.: Inducing probabilistic relational rules from probabilistic examples. In: Q. Yang, M. Wooldridge (eds.) 24th International Joint Conference on Artificial Intelligence (IJCAI 2015), pp. 1835–1843. AAAI Press (2015)
48. Raman, V., Ravikumar, B., Rao, S.S.: A simplified np-complete maxsat problem. Information Processing Letters **65**(1), 1–6 (1998)
49. Riguzzi, F.: Speeding up inference for probabilistic logic programs. Comput. J. **57**(3), 347–363 (2014). DOI 10.1093/comjnl/bxt096
50. Riguzzi, F.: Foundations of Probabilistic Logic Programming. River Publishers, Gistrup, Denmark (2018)
51. Riguzzi, F., Bellodi, E., Lamma, E., Zese, R., Cota, G.: Probabilistic logic programming on the web. Softw.-Pract. Exper. **46**(10), 1381–1396 (2016). DOI 10.1002/spe.2386
52. Riguzzi, F., Lamma, E., Alberti, M., Bellodi, E., Zese, R., Cota, G.: Probabilistic logic programming for natural language processing. In: F. Chesani, P. Mello, M. Milano (eds.) Workshop on Deep Understanding and Reasoning, URANIA 2016, *CEUR Workshop Proceedings*, vol. 1802, pp. 30–37. Sun SITE Central Europe (2017)
53. Riguzzi, F., Swift, T.: Tabling and answer subsumption for reasoning on logic programs with annotated disjunctions. In: ICLP TC 2010, *LIPIcs*, vol. 7, pp. 162–171. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2010). DOI 10.4230/LIPIcs.ICLP.2010.162
54. Riguzzi, F., Swift, T.: The PITA system: Tabling and answer subsumption for reasoning under uncertainty. Theor. Pract. Log. Prog. **11**(4–5), 433–449 (2011). DOI 10.1017/S147106841100010X
55. Rooshenas, A., Lowd, D.: Discriminative structure learning of arithmetic circuits. In: Artificial Intelligence and Statistics, pp. 1506–1514 (2016)

56. Sang, T., Beame, P., Kautz, H.A.: Performing bayesian inference by weighted model counting. In: 20th National Conference on Artificial Intelligence, pp. 475–482. AAAI Press, Palo Alto, California USA (2005)
57. Sato, T.: A statistical learning method for logic programs with distribution semantics. In: L. Sterling (ed.) ICLP 1995, pp. 715–729. MIT Press (1995)
58. Sato, T., Zhou, N.F., Kameya, Y., Izumi, Y., Kubota, K., Kojima, R.: PRISM User's Manual (Version 2.3) (2017). `http://rjida.meijo-u.ac.jp/prism/download/prism23.pdf`, accessed June 8, 2018
59. Sourek, G., Aschenbrenner, V., Zeleztný, F., Kuzelka, O.: Lifted relational neural networks. In: T.R. Besold, A.S. d'Avila Garcez, G.F. Marcus, R. Miikkulainen (eds.) NIPS Workshop on Cognitive Computation 2015, *CEUR Workshop Proceedings*, vol. 1583. CEUR-WS.org (2016)
60. Srinivasan, A., King, R.D., Muggleton, S., Sternberg, M.J.E.: Carcinogenesis predictions using ILP. In: N. Lavrac, S. Džeroski (eds.) ILP 1997, *LNCS*, vol. 1297, pp. 273–287. Springer Berlin Heidelberg (1997)
61. Srinivasan, A., Muggleton, S., Sternberg, M.J.E., King, R.D.: Theories for mutagenicity: A study in first-order and feature-based induction. Artif. Intell. **85**(1-2), 277–299 (1996)
62. Swift, T., Warren, D.S.: XSB: Extending prolog with tabled logic programming. Theor. Pract. Log. Prog. **12**(1-2), 157–187 (2012). DOI 10.1017/S1471068411000500
63. Van Gelder, A., Ross, K.A., Schlipf, J.S.: The well-founded semantics for general logic programs. J. ACM **38**(3), 620–650 (1991)
64. Vennekens, J., Verbaeten, S., Bruynooghe, M.: Logic Programs With Annotated Disjunctions. In: ICLP 2004, *LNCS*, vol. 3132, pp. 431–445. Springer (2004)
65. Wielemaker, J., Schrijvers, T., Triska, M., Lager, T.: Swi-prolog. Theory and Practice of Logic Programming **12**(1-2), 67–96 (2012)
66. Železnỳ, F., Srinivasan, A., Page, C.D.: Randomised restarted search in ilp. Machine Learning **64**(1-3), 183–208 (2006)
67. Železnỳ, F., Srinivasan, A., Page, D.: Lattice-search runtime distributions may be heavy-tailed. In: International Conference on Inductive Logic Programming, pp. 333–345. Springer (2002)
68. Železnỳ, F., Srinivasan, A., Page, D.: A monte carlo study of randomised restarted search in ilp. In: International Conference on Inductive Logic Programming, pp. 341–358. Springer (2004)

10 Appendix

10.1 Gradient Calculation

In this section we describe how to compute the derivative of the error w.r.t each node of the AC.

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 4 becomes $err = -\log(v(r))$. Therefore

$$\frac{\partial err}{\partial v(r)} = -\frac{1}{v(r)} \tag{25}$$

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

$$d(n) = \sum_{pa_n} \frac{\partial v(r)}{\partial v(pa_n)} \frac{\partial v(pa_n)}{\partial v(n)} = \sum_{pa_n} d(pa_n) \frac{\partial v(pa_n)}{\partial v(n)}. \tag{26}$$

If parent pa_n is a × 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

$$\frac{\partial v(pa_n)}{\partial v(n)} = \prod_{n' \neq n} v(n') = \frac{v(pa_n)}{v(n)} \tag{27}$$

if $n = \pi_i$ then

$$\frac{\partial v(pa_n)}{\partial \pi_i} = \prod_{n' \neq \pi_i} v(n') = \frac{v(pa_n)}{\pi_i} \tag{28}$$

The derivative of pa_n with respect to W_i corresponding to π_i is:

$$\begin{aligned}\frac{\partial v(pa_n)}{\partial W_i} &= \frac{\partial v(pa_n)}{\partial \sigma(W_i)} \frac{\partial \sigma(W_i)}{\partial W_i} = \frac{\partial v(pa_n)}{\partial \sigma(W_i)} \sigma(W_i)(1 - \sigma(W_i)) \\ &= \frac{\partial v(pa_n)}{\partial \pi_i} \pi_i(1 - \pi_i) = \frac{v(pa_n)}{\pi_i} \pi_i(1 - \pi_i) \\ &= v(pa_n)(1 - \sigma(W_i))\end{aligned} \tag{29}$$

If parent pa_n is a $\bigoplus$ node with n' indicating its children
$v(pa_n) = \bigoplus_{n'} v(n') = 1 - \prod_{n'}(1 - v(n'))$

$$\frac{\partial v(pa_n)}{\partial v(n)} = \prod_{n' \neq n} (1 - v(n')) = \frac{1 - v(pa_n)}{1 - v(n)} \tag{30}$$

If the unique parent of n is a $not(n)$ $v(pa_n) = 1 - v(n)$ and

$$\frac{\partial v(pa_n)}{\partial v(n)} = -1 \tag{31}$$

Because of the graph construction, $\bigoplus$ and × nodes can only have one × and $\bigoplus$ parent respectively. Leaf nodes can have many × parent nodes. Therefore Equation 26 can be written as

$$d(n) = \begin{cases} d(pa_n) \frac{v(pa_n)}{v(n)} & \text{if n is a } \bigoplus node, \\ d(pa_n) \frac{1 - v(pa_n)}{1 - v(n)} & \text{if n is a } \times \text{ node} \\ \sum_{pa_n} d(pa_n).v(pa_n).(1 - \pi_i) & \text{if n=}\sigma(W_i) \\ -d(pa_n) & pa_n = not(n) \end{cases} \tag{32}$$

Combining equations 25 and 32 we have:

$$\frac{\partial err}{\partial v(n)} = -d(n) \frac{1}{v(r)} \tag{33}$$

10.2 Message Passing

This appendix describes how to perform message passing over the arithmetic circuit.

In order to illustrate the passes, we construct a graphical model associated with the AC and then apply the *belief propagation* (BP) algorithm [44].

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 deterministic OR function, while variables associated with an $\times$ node have a CPT encoding a deterministic AND function. 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 6 is shown in Figure 8.

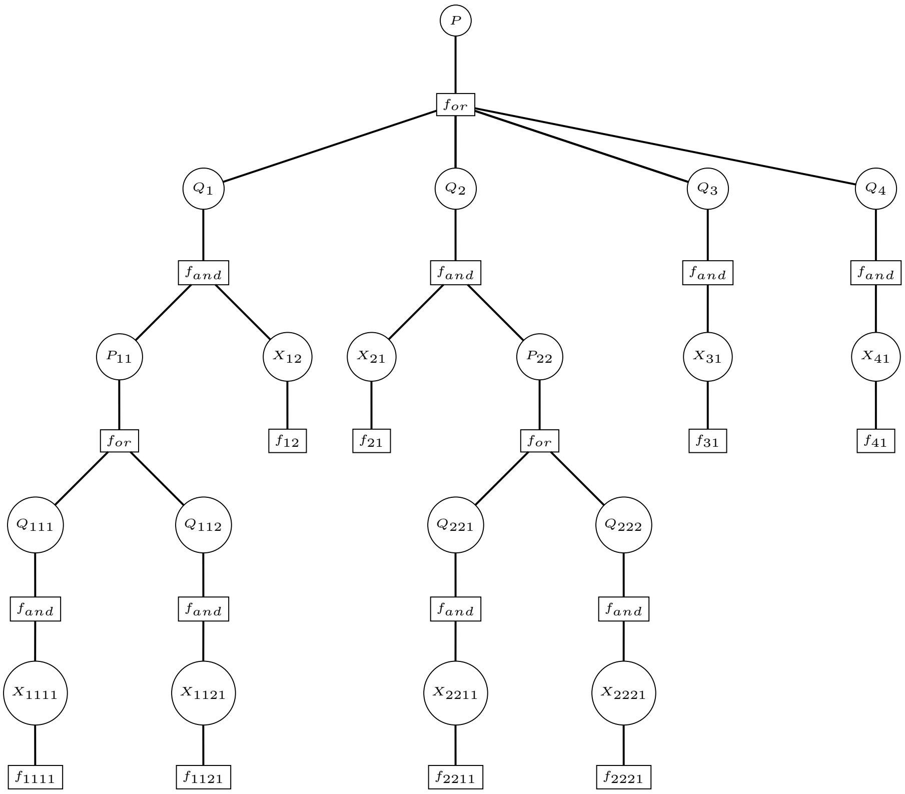

Fig. 8: Factor graph.

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, see [42] and [44]:

$$\mu_{N\to f}(n) = \prod_{h\in nb(N)\backslash f} \mu_{h\to N}(n) \tag{34}$$

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 defined as:

$$\mu_{f\to N}(n) = \sum_{\neg N}(f(n,\mathbf{s}) \prod_{Y\in nb(f)\backslash N} \mu_{Y\to f}(y)) \tag{35}$$

where $nb(f)$ is the set of arguments of f. After convergence, the belief of each variable N is computed as follows:

$$b(n) = \prod_{f\in nb(N)} \mu_{f\to N}(n) \tag{36}$$

that is the product of all incoming messages to the variable. By normalizing b(n) we obtain $P(N = n|e)$ $(n \in \{0, 1\})$. We want to develop an algorithm for computing $b(n)$ over the AC. So we want the AC nodes to send messages. Let $c_N = \mu_{f\to N}(N = 1)$ be the normalized messages in the bottom-up pass and $t_N = \mu_{f\to N}(N = 1)$ the normalized messages in the top-down pass. We proved in [42] that $c_N = v(N)$ and

$$t_N = \begin{cases} \frac{t_P}{t_P + v(P)\ominus v(N)\cdot t_P + (1-v(P)\ominus v(N))\cdot(1-t_p)} & \text{if } P \text{ is a } \oplus \text{ node} \\ \frac{t_P\cdot\frac{v(P)}{v(N)} + (1-t_P)\cdot(1-\frac{v(P)}{v(N)})}{t_P\cdot\frac{v(P)}{v(N)} + (1-t_P)\cdot(1-\frac{v(P)}{v(N)}) + (1-t_P)} & \text{if } P \text{ is a } \times \text{ node} \\ 1 - t_P & \text{if } P \text{ is a } \neg \text{ node} \end{cases} \tag{37}$$

where $v(N)$ is the value of node N, P is its parent and the operator $\ominus$ is defined as

$$v(p) \ominus v(n) = 1 - \frac{1 - v(p)}{1 - v(n)} \tag{38}$$

10.3 Regularization

In this appendix we give the proofs of theorems 1 and 2.

Theorem 3 *The L_1 regularized objective function:*

$$J_1(\theta) = N_1 \log\theta + N_0 \log(1-\theta) - \gamma\theta \tag{39}$$

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

$$J_1(\theta) = \frac{N_1}{\theta} - \frac{N_0}{1-\theta} - \gamma \tag{40}$$

Solving $J_1' = 0$ we have:

$$\begin{aligned} \frac{N_1}{\theta} - \frac{N_0}{1-\theta} - \gamma = 0 \\ N_1(1-\theta) - N_0\theta - \gamma\theta(1-\theta) = 0 \\ N_1 - N_1\theta - N_0\theta - \gamma\theta + \gamma\theta^2 = 0 \\ \gamma\theta^2 - (N_0 + N_1 + \gamma)\theta + N_1 = 0 \end{aligned} \tag{41}$$

Equation 41 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

$$\begin{aligned} \Delta = b^2 - 4ac = (N_0 + N_1)^2 + \gamma^2 + 2(N_0 + N_1)\gamma - 4\gamma N_1 = \\ (N_0 + N_1)^2 + \gamma^2 - 2\gamma N_1 + 2\gamma N_0 \end{aligned}$$

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 4 *The L_2 regularized objective function:*

$$J_2(\theta) = N_1 \log \theta + N_0 \log(1 - \theta) - \frac{\gamma}{2}\theta^2 \tag{42}$$

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

$$J_2'(\theta) = \frac{N_1}{\theta} - \frac{N_0}{1-\theta} - \gamma\theta \tag{43}$$

Solving $J_2'(\theta) = 0$ we have:

$$\begin{aligned}
\frac{N_1}{\theta} - \frac{N_0}{1-\theta} - \gamma\theta &= 0 \\
N_1(1-\theta) - N_0\theta - \gamma\theta^2(1-\theta) &= 0 \\
N_1 - N_1\theta - N_0\theta - \gamma\theta^2 + \gamma\theta^3 &= 0 \\
\gamma\theta^3 - \gamma\theta^2 - (N_0+N_1)\theta + N_1 &= 0 \\
\theta^3 - \theta^2 - \frac{(N_0+N_1)}{\gamma}\theta + \frac{N_1}{\gamma} &= 0 \qquad (44)
\end{aligned}$$

Equation 44 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

$$a\theta^3 + b\theta^2 + c\theta + d = 0 \tag{45}$$

The number of real and complex roots is determined by the discriminant of the cubic equation [10]:

$$\Delta = 18abcd - 4b^3d + b^2c^2 - 4ac^3 - 27a^2d^2$$

In our case we have

$$\Delta = -\frac{27N_1^2}{\gamma^2} + \frac{4N_1}{\gamma} - \frac{18N_1\left(-N_0-N_1\right)}{\gamma^2} + \frac{\left(-N_0-N_1\right)^2}{\gamma^2} - \frac{4\left(-N_0-N_1\right)^3}{\gamma^3} = \\
\frac{4N_0^3}{\gamma^3} + \frac{12N_0^2N_1}{\gamma^3} + \frac{N_0^2}{\gamma^2} + \frac{12N_0N_1^2}{\gamma^3} + \frac{20N_0N_1}{\gamma^2} + \frac{4N_1^3}{\gamma^3} - \frac{8N_1^2}{\gamma^2} + \frac{4N_1}{\gamma}$$

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 $\gamma \geq N_0 + N_1$

$$\begin{aligned}\varDelta = &\tfrac{1}{\gamma^3}\left(4N_1\gamma^2 + \gamma\left(-27N_1^2 + 18N_1\left(N_0+N_1\right) + \left(N_0+N_1\right)^2\right) + \right.\\ &\left. 4\left(N_0+N_1\right)^3\right) \geq \\ &\tfrac{1}{\gamma^3}\left(4N_1\left(N_0+N_1\right)^2 + 4\left(N_0+N_1\right)^3 + \right.\\ &\left.\left(N_0+N_1\right)\left(-27N_1^2 + 18N_1\left(N_0+N_1\right) + \left(N_0+N_1\right)^2\right)\right) = \\ &\tfrac{1}{\gamma^3}N_0\left(5N_0^2 + 37N_0N_1 + 32N_1^2\right) \geq \\ &0\end{aligned}$$

So equation 44 has 3 real roots that can be computed as [10]:

$$\theta_k = 2\sqrt{-\frac{p}{3}}\cos\left(\frac{1}{3}\arccos\left(\frac{3q}{2p}\sqrt{\frac{-3}{p}}\right) - \frac{2\pi k}{3}\right) + \frac{1}{3} \quad \text{for} \quad k = 0,1,2$$

where

$$\begin{aligned}p =& \frac{3ac - b^2}{3a^2},\\ q =& \frac{2b^3 - 9abc + 27a^2d}{27a^3}.\end{aligned}$$

θ_1 is

$$\theta_1 = \frac{2\sqrt{\frac{3N_0+3N_1+\gamma}{\gamma}}\cos\left(\frac{\arccos\left(\frac{\sqrt{\frac{\gamma}{3N_0+3N_1+\gamma}}\left(\frac{9N_0}{2} - 9N_1 + \gamma\right)}{3N_0+3N_1+\gamma}\right)}{3} - \frac{2\pi}{3}\right)}{3} + \frac{1}{3} \quad (46)$$

Since $\lim_{\theta\to 0+} J_2(\theta) = \lim_{\theta\to 1-} J_2(\theta) = -\infty$, 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 46. 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 $\arccos(z)$:

$$\begin{aligned}\arccos(z) ={} & \frac{\pi}{2} - z - \left(\frac{1}{2}\right)\frac{z^3}{3} - \left(\frac{1\cdot 3}{2\cdot 4}\right)\frac{z^5}{5} - \left(\frac{1\cdot 3\cdot 5}{2\cdot 4\cdot 6}\right)\frac{z^7}{7} - \cdots = \\ & \frac{\pi}{2} - z - O(z^3)\end{aligned}$$

so

$$\lim_{\gamma\to 0}\arccos\left(\frac{\sqrt{\frac{\gamma}{3N_0+3N_1+\gamma}}\left(\frac{9N_0}{2} - 9N_1 + \gamma\right)}{3N_0+3N_1+\gamma}\right) = \tag{47}$$

$$\lim_{\gamma\to 0}\arccos(z) = \tag{48}$$

$$\lim_{\gamma\to 0}\frac{\pi}{2} - \sqrt{\frac{\gamma}{(3N_0+3N_1+\gamma)^3}}\left(\frac{9N_0}{2} - 9N_1 + \gamma\right) - O(\gamma^{\frac{3}{2}}) \tag{49}$$

Then

$$\lim_{\gamma\to 0}\cos\left(\frac{\arccos\left(\frac{\sqrt{\frac{\gamma}{3N_0+3N_1+\gamma}}\left(\frac{9N_0}{2}-9N_1+\gamma\right)}{3N_0+3N_1+\gamma}\right)}{3}-\frac{2\pi}{3}\right)= \tag{50}$$

$$\lim_{\gamma\to 0}\cos\left(\frac{\arccos\left(z\right)}{3}-\frac{2\pi}{3}\right)= \tag{51}$$

$$\lim_{\gamma\to 0}\cos\left(\frac{\pi}{6}-\frac{z}{3}-O(\gamma^{\frac{3}{2}})-\frac{2\pi}{3}\right)= \tag{52}$$

$$\lim_{\gamma\to 0}\cos\left(-\frac{z}{3}-O(\gamma^{\frac{3}{2}})-\frac{\pi}{2}\right)= \tag{53}$$

$$\lim_{\gamma\to 0}\cos\left(-\frac{\pi}{2}\right)\cos\left(-\frac{z}{3}-O(\gamma^{\frac{3}{2}})\right)+\sin(-\frac{\pi}{2})\sin\left(\frac{z}{3}-O(\gamma^{\frac{3}{2}}))\right)= \tag{54}$$

$$\lim_{\gamma\to 0}-\sin\left(\frac{z}{3}-O(\gamma^{\frac{3}{2}}))\right) \tag{55}$$

Since the Maclaurin expansion of sin is

$$\sin y = y-\frac{y^3}{3!}+\frac{y^5}{5!}-\frac{y^7}{7!}+\cdots$$

then

$$\lim_{\gamma\to 0}-\sin\left(\frac{z}{3}-O(\gamma^{\frac{3}{2}}))\right)= \tag{56}$$

$$\lim_{\gamma\to 0}-\frac{z}{3}+O(\gamma^{\frac{3}{2}})= \tag{57}$$

$$\lim_{\gamma\to 0}-\frac{1}{3}\sqrt{\frac{\gamma}{(3N_0+3N_1+\gamma)^3}}\left(\frac{9N_0}{2}-9N_1+\gamma\right)+O(\gamma^{\frac{3}{2}}) \tag{58}$$

So

$$\begin{aligned}\lim_{\gamma\to 0}\theta_1 = {} & \lim_{\gamma\to 0}\frac{2\sqrt{\frac{3N_0+3N_1+\gamma}{\gamma}}\left(-\frac{1}{3}\sqrt{\frac{\gamma}{(3N_0+3N_1+\gamma)^3}}\left(\frac{9N_0}{2}-9N_1+\gamma\right)+O(\gamma^{\frac{3}{2}})\right)}{3}+\\ & \frac{1}{3}=\\ & -\lim_{\gamma\to 0}\frac{2\frac{1}{3N_0+3N_1+\gamma}\left(\frac{9N_0}{2}-9N_1+\gamma\right)+2\sqrt{\frac{3N_0+3N_1+\gamma}{\gamma}}O(\gamma^{\frac{3}{2}})}{9}+\frac{1}{3}=\\ & -\frac{2\frac{\frac{9N_0}{2}-9N_1}{3N_0+3N_1}}{9}+\frac{1}{3}=\\ & -2\frac{\frac{N_0}{2}-N_1}{3N_0+3N_1}+\frac{1}{3}=\\ & -\frac{N_0-2N_1}{3N_0+3N_1}+\frac{1}{3}=\\ & \frac{-N_0+2N_1+N_0+N_1}{3N_0+3N_1}=\frac{3N_1}{3N_0+3N_1}\\ & =\frac{N_1}{N_0+N_1}\end{aligned}$$

10.4 Comparison between PHIL and SLEAHP

Table 8: Average area under ROC curve for PHIL and SLEAHP

AUCROC	Mutagenesis	Carcinogenesis	Mondial	UWCSE	PAKDD15
DPHIL$_1$	0.841021	0.571053	0.534817	0.960876	0.504912
SLEAHP$_{G_1}$	0.889676	0.493421	0.483865	0.936160	0.506252
DPHIL$_2$	0.880465	0.618421	0.534563	0.949548	0.514218
SLEAHP$_{G_2}$	0.845452	0.544737	0.472843	0.925436	0.506242
EMPHIL$_1$	0.884016	0.684211	0.536009	0.938121	0.504741
SLEAHP$_{E_1}$	0.878727	0.660526	0.433016	0.907789	0.503251
EMPHIL$_2$	0.885478	0.623684	0.534622	0.969046	0.504742
SLEAHP$_{E_2}$	0.904933	0.414135	0.483798	0.904347	0.505691
EMPHIL$_{B_1}$	0.833539	0.619730	0.536042	0.930243	0.504196
SLEAHP$_B$	0.822833	0.618421	0.464058	0.925099	0.504196

Table 9: Average area under PR curve for PHIL and SLEAHP

AUCPR	Mutagenesis	Carcinogenesis	Mondial	UWCSE	PAKDD15
DPHIL$_1$	0.886598	0.563875	0.142331	0.191302	0.222783
SLEAHP$_{G_1}$	0.929906	0.498091	0.701244	0.148115	0.223074
DPHIL$_2$	0.929244	0.580041	0.147390	0.219806	0.218153
SLEAHP$_{G_2}$	0.918519	0.502135	0.690782	0.131750	0.223028
EMPHIL$_1$	0.944758	0.679712	0.142696	0.275985	0.222609
SLEAHP$_{E_1}$	0.948563	0.598095	0.632270	0.059562	0.220226
EMPHIL$_2$	0.944517	0.655781	0.142066	0.307713	0.222618
SLEAHP$_{E_1}$	0.955678	0.540510	0.623542	0.069861	0.225017
EMPHIL$_{B_1}$	0.880013	0.649090	0.142810	0.261578	0.222043
SLEAHP$_B$	0.900300	0.552477	0.623542	0.059655	0.222043

Table 10: Average time for PHIL and SLEAHP

Time	Mutagenesis	Carcinogenesis	Mondial	UWCSE	PAKDD15
DPHIL$_1$ SLEAHP$_{G_1}$	5.2059 41.8250	177.4270 48.7600	311.3280 59.5054	0.2910 219.6410	20.5704 192.4396
DPHIL$_2$ SLEAHP$_{G_2}$	5.4450 47.1344	88.5100 10524.0900	301.1392 14.0470	0.2214 194.9706	4.9517 162.9938
EMPHIL$_1$ SLEAHP$_{E_1}$	4.8940 48.0152	181.0890 303.0570	317.4202 60.8316	1.0000 387.665	7.0691 151.7217
EMPHIL$_2$ SLEAHP$_{E_2}$	5.0046 45.9245	146.8440 92.3820	245.3830 61.0995	0.8372 312.2604	6.6334 68.2432
EMPHIL$_{B_1}$ SLEAHP$_B$	2.6478 13.1478	85.3210 1399.0090	248.1978 14.6698	0.1572 295.6734	6.0990 61.5347