IMPERIAL COLLEGE LONDON
DEPARTMENT OF COMPUTING

PROBXHAIL: An Abductive-Inductive Algorithm for Probabilistic Inductive Logic Programming

Author:
Stanislav DRAGIEV

Supervision:
Dr. Alessandra RUSSO
Dr. Krysia BRODA
Mark LAW

June 13, 2016

Abstract

This project describes the development of a machine learning model that integrates Bayesian statistics in logic-based learning. This results in a model that is easy to comprehend, a benefit of first-order logic theories, and is able to express uncertainty like statistical models. We do this by defining a programming language for probabilistic logic programming, and then developing algorithms that learn the statistical parameters of the program and the structure of its first-order logic theory. The probabilistic programming language has certain interesting properties when compared to state-of-the-art probabilistic logic programs, including the use of stable set semantics and Bayesian priors. In addition, the algorithm introduces the concept of abductive-inductive learning in probabilistic inductive logic programming (PILP), i.e. learning first-order theories by combining abduction and induction.

First, we define annotated literal programs, a type of probabilistic logic program in which every literal is annotated with a parameter for a Bernoulli or Beta distribution. The associated probabilistic model, which is defined by a generative story for a normal logic program, uses independent Bernoulli trials to determine whether each literal should be included in the theory. We outline parameter learning algorithms for the program based on statistical abduction. We define PROBXHAIL (PROBabilistic eXtended Hybrid Abductive-Inductive Learning), an inductive-abductive algorithm for structural learning that is a generalization of XHAIL[26]. The properties of annotated literal programs and the corresponding machine learning algorithms are evaluated in light of state-of-the-art probabilistic logic programming languages.

Acknowledgements

I am very grateful for the time that Dr. Alessandra Russo and Dr. Krysia Broda dedicated to my project and for the guidance I was given to take the project in an original direction. Mark Law also provided me with a lot of insight on various topics in logic programming, and his knowledge of how to optimize Answer Set Programs was particularly helpful to me. I am also very grateful for Calin Rares Turliuc's help in understanding statistical abduction and probabilistic logic programming. Last, but not least, I would like to thank my parents for their support during my studies.

Contents

List of Figures

List of Abbreviations

ALP	**A**nnotated **L**iteral **P**rogram(ming)
ASP	**A**nswer **S**et **P**rogram(ming)
BDD	**B**inary **D**ecision **D**iagram
EM	**E**xpectation **M**aximization
HAIL	**H**ybrid **A**bductive **I**nductive **L**earning
LDA	**L**atent **D**irichlet **A**llocation
LPAD	**L**ogic **P**rogram with **A**nnotated **D**isjunctions
MCMC	**M**arkov **C**hain **M**onte **C**arlo
MSE	**M**ean **S**quare **E**rror
PILP	**P**robabilistic **I**nductive **L**ogic **P**rogramming
PLP	**P**robabilistic **L**ogic **P**rogram(ming)
PROBXHAIL	**PROB**abilistic E**X**tended **H**ybrid **A**bductive **I**nductive **L**earning
ROBDD	**R**educed **O**rdered **B**inary **D**ecision **D**iagram
SLDNF	**S**elective **L**inear **D**efinite clause resolution with **N**egation as **F**ailure
XHAIL	E**X**tended **H**ybrid **A**bductive **I**nductive **L**earning

Chapter 1

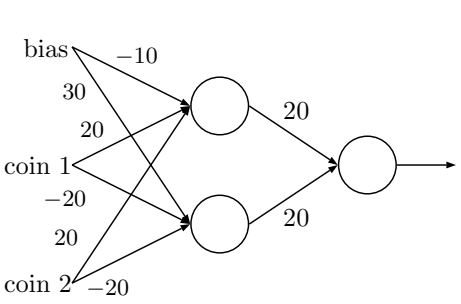
Introduction

Logic-based learning is concerned with learning explicit rule sets that explain data, often in the form of first-order logic theories. The greatest benefit of logic-based models is that rule sets are very easy to interpret in a natural language, which can be very useful in data mining. Moreover, extending the models with prior knowledge is very straightforward. Modifying the model involves simply adding intuitive rules or constraints. A disadvantage of logic-based models is that they do not typically handle noisy and uncertain data well.

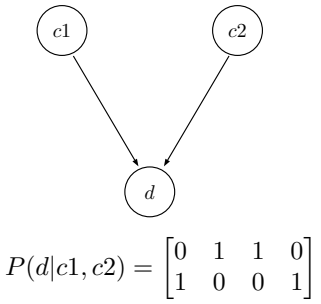
A commonly used method in machine learning is Artificial Neural Networks. With this approach, any function can be approximated to an arbitrary degree by building a network of computational units called neurons. Neurons output a single value, which is based on the a weighed input from other neurons. Learning involves choosing the weights that minimize the error between the output and the expected result. Artificial Neural Networks are designed to approximate real-valued functions and because of that they are, in a way, a complete opposite of logic-based models: they are much more scalable, distributable and capable of handling noisy and redundant data (e.g. images). On the other hand, the connectivity weights are not easily interpreted by humans. This means that it is very hard to comprehend the significance of the features in the learned model.

Another common set of methods in machine learning rely on Bayesian statistics. The learning task is to find parameters for statistical models that maximize the likelihood of the data given the parameters and the model. Inference on these models can be very efficient and can be used on large data sets. Statistical models are typically easier to interpret than Artificial Neural Networks, though understanding what has been learned still involves interpreting the meaning of parameters of statistical distributions. Another difficulty lies in coming up with a correct probabilistic model, which is able to capture the conditional dependencies of the data.

An example that illustrates the advantages and disadvantages of different machine learning models is shown in Figure 1.1, where three different machine learning models are proposed for the same task. The goal is to learn when 2 different coin tosses have different results. Since the neural network and the probabilistic models do not directly work with a symbolic representation, heads and tails are encoded as 1 and 0, respectively. The goal is thus to learn the XOR function. The neural network requires a hidden layer and a bias node to learn a non-linearly separable function, which



(A) Neural Network



(B) Probabilistic Model

```
different(C1, C2) :- heads(C1), not heads(C2).
different(C1, C2) :- not heads(C1), heads(C2).
```

(C) Logic Program

FIGURE 1.1: A comparison of 3 different models of the coin toss problem

results in a complex architecture. A more elegant conditional probability model can be constructed to learn the same task. In a logic program, XOR has a trivial expression. However, introducing noise on the observation is something that cannot be elegantly handled in a deterministic logic program.

Combining logic-based learning and Bayesian statistics results in symbolic models that can cope with uncertainty, benefiting from both higher accuracy and easy interpretation. Such models are generally encoded as probabilistic logic programs, which, similarly to logic programs, define a set of clauses. However, some of their clauses or literals are linked to probability distributions, which allows them to express a probability distribution over interpretations. Logic-based learning models augmented with probabilistic elements are better at handling noisy and uncertain data than deterministic logic-based models. They are able to express common probabilistic models such as Bayesian Networks or Markov Random Fields, which means their expressive power is comparable to probabilistic models with discrete random variables. In addition, they are able to include prior knowledge in the form of first-order logic rules. As a result of this, Probabilistic Logic-based learning models can, in theory, achieve high accuracy on datasets while still being easy to interpret.

In this project, we aim to develop a new PILP method by combining together two different areas of research: abduction-driven logic based learning and statistical abduction. The ultimate goal is to define a new notion of Probabilistic Hybrid Abductive-Inductive Learning, that is able to learn a probability distribution of hypotheses and allows users to comprehend the significance of specific clauses in logic-based models.

This report is split in two parts. First, we define a Probabilistic Logic Programming Language, which we call Annotated Literal Programs and

in which every literal is associated with a probability. We argue that this language is equally expressive to other contemporary PLP languages. We then show that learning the best parameters of a given program from interpretations can be expressed as a statistical abduction task. For this reason, we consider input programs for Peircebayes[38], a language for statistical abduction, and the performance implications of switching from Prolog to an Answer Set Program representation. Whilst Prolog programs and Answer Set programs use similar syntax to describe logic programs, the inherent difference between them lies in the fact that Prolog uses top-down SLDNF resolution to perform deduction/abduction, whereas answer set solvers (e.g. clingo) use a bottom-up approach in search for stable sets. Some problems that are inefficiently expressed in Prolog can be more naturally and efficiently described in a bottom-up manner. We are able to show a performance improvement in some tasks, particularly in seeded LDA. As a more practical consequence, Peircebayes parameter learning can be integrated with XHAIL[26], an ILP method whose implementation is based on ASP and abduction-driven search.

As a second main contribution to this project, we investigate how to use statistical abduction to produce an algorithm for learning annotated literal programs. The resulting algorithm, named PROBXHAIL, is a modified version of XHAIL in which the hypothesis compression abductive task is generalized to statistical abduction. Whilst previous PILP algorithms such as [3] have used most specific clauses for examples to restrict the hypothesis search space, PROBXHAIL takes this one step further by using the most specific hypothesis as a basis for the probabilistic model itself. The most-specific hypothesis is rewritten in the PLP language described in the first part and a parameter learning algorithm is used to learn the parameters associated with each literal. In XHAIL, abduction is used to search for a most compressed hypothesis, whereas PROBXHAIL uses statistical abduction to weigh the significance of clauses. In this report, we describe the PILP task that PROBXHAIL attempts to solve and argue that the proposed algorithm and its implementation correctly computes it. Then, we evaluate the model runtime and error on a synthetic dataset.

Chapter 2

Background

2.1 Clause Theory Semantics

The report assumes that the reader is familiar with first-order logic and Prolog's SLDNF procedure.

Literals are atoms or negated atoms. A clause is a disjunction of literals. A Horn clause consists at most one positive literal. A definite clause consists of exactly one positive literal, called the head, and 0 or more literals, called body literals. A definite logic program is a set of definite clauses, whereas a normal program is a set of rules of the form $A \leftarrow B_1, B_2 \ldots ...B_n$ where B_k is either an atom or an atom preceded by not, which denotes Negation as Failure.[35]

The set of ground atoms of the language of a logic program P is called the *Herbrand base* of P. A Herbrand interpretation maps the Herbrand base to truth values. It is also commonly written as the subset of the Herbrand base that is true according to the interpretation. A Herbrand interpretation of a definite/normal logic program is *supported* if, for every true ground atom in the Herbrand base, there is a clause whose body predicates are all true. A Herbrand model is a Herbrand interpretation in which every formula of P is true. A formula α is entailed by a program P when α is true in every model of P. This is denoted by $P \vDash \alpha$. A Herbrand model is minimal if it is a subset of all Herbrand models of a program P. A minimal Herbrand model is least if it is a strict subset of all other Hebrand models of P. For definite logic programs, the least Herbrand model exists and is always supported. The least Herbrand model of a program, if it exists, is denoted by $M(P)$. A logic program's semantics can be defined by its minimal supported Herbrand models.

The immediate consequence operator T_P of a program P maps between sets of ground atoms of P. It is defined as the set of heads of all clauses in P whose bodies are a subset of the given set. Hence, for a Herbrand interpretation X_P of definite program P, X_P is a Herbrand model of P iff $T_P(X_P) \subseteq X_P$, and is supported iff $X_P \subseteq T_P(X_P)$. So, the supported Hebrand models are fixpoints of T_P and the least fixpoint of T_P is the least Herbrand model of P. By Knaster-Tarski lemma, the least Herbrand model of a definite logic program can be computed by fixpoint iteration on the empty set[35].

Stable models provide an intuitive semantics for normal logic programs. A reduct P_X of a normal logic program P for a given set X removes all clauses from the program that contain body literals `not B` such that $B \in X$. It also removes all other negated literals from the bodies of the other clauses. Note that the reduct of a normal logic program is a definite logic program. A set is said to be a stable model (or an answer set) of a normal logic program P iff $X = M(P_X)$[36].

2.2 BDDs

As explained in [27], Binary Decision Diagram is a rooted directed acyclic graph that is used to represent a boolean function. It consists of nodes representing boolean variables/constants and edges which represent assignments.

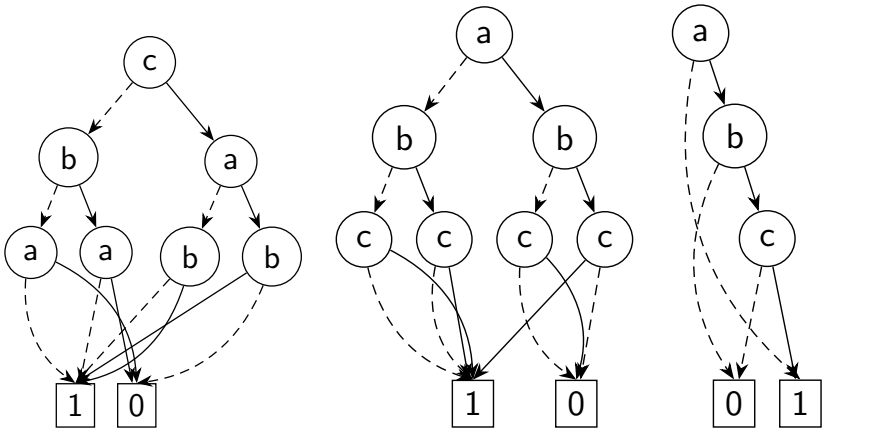
- An **terminal node** represents either truth or falsity. It does not have any outgoing edges.
- A **variable node** represents a boolean variable. There is always a path from a variable node to a terminal node. An outgoing edge could be labelled with either 0 or 1. A 0/1 edge represents an assignment of the variable to false/true, respectively.

The boolean function represented by the BDD can be evaluated for a given assignment by starting at the root node, looking up the assignment of the variable of the current node and following the corresponding edge until a terminal node is reached. The value of the terminal node is the value of the function for the given assignment.

Any propositional formula can be represented as a BDD. The way the BDD can be built for a formula F is by the following procedure.

1. If the formula consists of only $\top$ and $\bot$, then evaluate it and return a corresponding terminal node. This is the root of the BDD.
2. Otherwise, choose an atom b from F and construct a variable node with b as a label. This is the root of the BDD.
3. Obtain F_+ and F_- by substituting b in F with $\top$ and $\bot$, respectively.
4. Construct the BDDs of F_+ and F_- and connect the variable node to the root of the BDDs of F_+ and F_- with a 1-edge and a 0-edge, respectively.

If the order of atom choices is fixed for the BDD, it is said to be ordered. Note that the size of the BDD depends on the ordering of the variables. The size of the BDD constructed above can be minimized by having nodes share subtrees and removing nodes whose 0 and 1 edge point to the same node. BDDs minimized in such a way with a fixed ordering are known as Reduced Ordered BDDs (ROBDDs). For example, three different BDDs for $(a \rightarrow b) \wedge (a \rightarrow c)$ are shown in 2.1. Solid arrows represent 1-edges and dotted arrows represent 0-edges. The ordering is is $a < b < c$. Only 5 nodes are needed to represent the ROBDD, whereas the original BDD requires 9.



(A) Unordered BDD (B) Ordered BDD (C) ROBDD

FIGURE 2.1: A BDD for $(a \rightarrow b) \wedge (a \rightarrow c)$ and its ordered and reduced equivalent

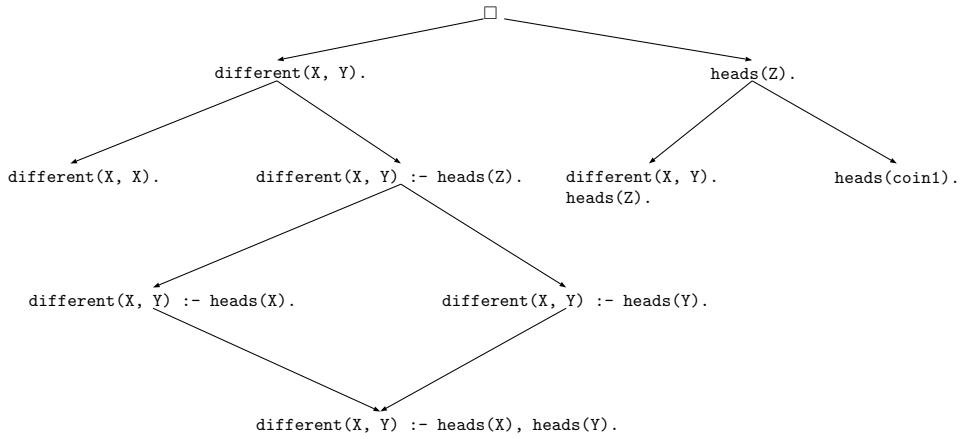


FIGURE 2.2: A partial expansion of a search lattice on Shapiro's[37] ρ refinement operator for the machine learning task depicted in Figure 1.1c

Note that ROBDDs are canonical, in a sense that equivalent propositional formulas translate to the same ROBDDs (for a fixed order of the variables). The rest of the report assumes that all BDDs are reduced and ordered.

2.3 Inductive Logic Programming

Inductive Logic Programming, as defined in [33], is a machine learning task commonly used for logic-based learning. Given a set of positive and negative examples, we are able to evaluate whether a given hypothesis (typically expressed in first-order logic) implies as many of the positive examples and as few of the negative examples as possible. Formally, given observations in a language L_E, partitioned in a set of positive examples E^+ and a set of negative examples E^-, background knowledge B, a hypothesis language L_H and a coverage criterion $covers(B, H, e)$, the goal of an ILP task is to find a theory H in L_H that satisfies both $\forall e \in E^+.covers(B, H, e)$ and $\forall e \in E^-.\neg covers(B, H, e)$. In Learning from Entailment, the coverage criterion is the classical logic entailment and the task is to find a theory H in L_H

such that $\forall e \in E^+.B, H \models e$ and $\forall e \in E^-.B, H \not\models e$. Note that in Learning from Entailment, the generality relation can be exploited to define refinement operators (as in FOIL[28], MIS[37]) and structure the search space as a lattice such as the one shown in Figure 2.2. The search could then proceed top-down or bottom-up using a preferred search algorithm and termination conditions.

Note that the search space described by such a refinement operator is infinite and grows combinatorially with the number of possible literals and so it is necessary to prune/prioritize the search based on some bias. This can be done in an intuitive manner using mode declarations, which determine what atoms and type of terms can appear in the head and the body of clauses and how many times they can be used.

Example: The lattice structure described in Figure 2.2 could have been generated using the following bias:

```
modeh(different(coin, coin)).
modeh(heads(-coin)).
modeh(heads(\#coin)).
modeb(heads(+coin).
modeb(heads(-coin)).
determination(different/2, heads/1).
```

Note that the syntax in this example is representative of real ILP algorithms and is based on the syntax defined in HYPER[6]. `modeh` and `modeb` statements are used to declare that the specified predicate can be used in the head/body of a clause, respectively. The declarations include type information for the terms associated with each predicate. The +, − and # signs prepended to the term type specify that the term must be an input variable (matched with a previous variable), output variable (newly declared), and a constant, respectively. A determination specifies that the second predicate can be used in the body of a clause which has the first predicate as its head.

2.3.1 Inverse entailment and Bottom Generalization

An advantage of such a bias definition is that, given the examples and the background knowledge, a compact most specific hypothesis can be computed and used to bound the search lattice from the bottom. In order to formalize this, Muggleton[25] defines the concept of Inverse Entailment. The task of searching for a hypothesis in a search space, typically expressed as finding H such that $B \cup H \models E$, can be redefined equivalently as finding H for which $B \cup \bar{E} \models \bar{H}$ where $\bar{H}$ and $\bar{E}$ are the first-order logic complements of H and E, respectively. Reformulating the task in this way is useful because it suggests a deductive method for computing a most specific hypothesis. If we know for a specific hypothesis H_{bot} that $B \cup \bar{E} \models \overline{H_{bot}} \models \bar{H}$ for every H that is consistent with B and E, then we can conclude that $H \models H_{bot}$, hence that H_{bot} is at least as specific as any other compatible hypothesis. This means that $\overline{H_{bot}}$ can be computed by deducing every possible ground literal from $B \cup \bar{e}$ for every example and then taking the complement. A generalization of this hypothesis can then be chosen as the actual model. This procedure is known as Bottom Generalization and is first used

in Progol[25].

2.3.2 (Extended) Hybrid Abductive Inductive Learning

Bottom Generalization can be extended to a procedure named Kernel Set Subsumption[29]. This involves searching for hypotheses that are more general than a so called Kernel Set $\mathcal{K}$. This is defined for any example e and background B representable as a Horn clause and a Horn theory, respectively:

$$\mathcal{K}(B,e) = \left\{ \begin{array}{c} \alpha_1 \leftarrow \delta_1^1, \ldots, \delta_1^{m(1)} \\ \vdots \\ \alpha_i \leftarrow \delta_i^1, \ldots, \delta_i^{m(i)} \\ \vdots \\ \alpha_n \leftarrow \delta_n^1, \ldots, \delta_n^{m(n)} \end{array} \right\}$$

where $B \cup \bigcup_i \alpha_{i\in[1,n]} \models e$ and $B \models \bigcup_{i\in[1,n],j\in[1,m(i)]} \delta_i^j$. Thus we can compute the Kernel set for any example in the following way:

1. Find all α values, the heads of the Kernel Set, by abduction with goal e and background B. Use the bias to determine the abducibles: all the predicates in head mode declarations should be considered.

2. Compute the body of the Kernel Set. For every clause in the Kernel Set, deduce everything from B that agrees with the body mode declarations and the corresponding head predicate α.

Then, similarly to Bottom Generalization, a search for a hypothesis that subsumes the Kernel Set can proceed. This is exactly how HAIL[29] works: While there are uncovered examples, the algorithm picks an example, computes the Kernel Set and picks the smallest hypothesis from the ones that subsume it and cover the maximum number of other examples. Then the computed hypothesis is added to B and the process is repeated for the uncovered examples.
Note that the iterative procedure in HAIL only works with monotonic theories. XHAIL[26] rectifies this by attempting to generate a Kernel Set[1] for an appropriately clustered set of examples. The algorithm also solves the inductive task, searching for the most appropriate hypothesis that subsumes the Kernel set, using abduction. The way the task is defined is by associating an abducible with every literal in the Kernel Set, which is called `use_clause_literal(i,j)` for β_i^j and `use_clause_literal(i,0)` for α_i. Background knowledge for the abductive task is generated from the Kernel Set. For each clause i in the Kernel Set, a rule is constructed starting which has the same head and cointains `use_clause_literal(i,0)` in the body. In addition, the rule contains a `try(i,j,X1,...XN)` for every literal β_j^i in the original clause, where each `X1,...XN` are the variables in the original literal. Moreover, the background knowledge contains two clauses are added for each body literal β_j^i:

[1]The concept of Kernel Set in XHAIL is slightly modified from the original definition.

```
try(i,j,X1,...XN) :- not use_clause_literal(i, j).
try(i,j,X1,...XN) :- beta, use_clause_literal(i, j),
        t1(X1)...tn(XN).
```

where *beta* is β_j^i and `ti` is the type predicate for variable `Xi`. The body literal is appended to the body of the second clause.
Example: Given bias declarations that associate each variable with an appropriate type and given a Kernel Set:

```
different(X,Y) :- heads(X), heads(Y).
different(X,Y) :- different(X, mycoin),
        different(mycoin, Y).
```

The following abductive task can be constructed (assuming clause numbering starts with 0):

```
different(X, Y) :- use_clause_literal(0,0),
  try(0,1,X), try(0,2,Y).
try(0,1,X) :-
  not use_clause_literal(0,1).
try(0,1,X) :-
  use_clause_literal(0,1), heads(X), coin(X).
try(0,2,Y) :-
  not use_clause_literal(0,2).
try(0,2,Y) :-
  use_clause_literal(0,2), heads(Y), coin(Y).

different(X, Y) :- use_clause_literal(1,0),
  try(1,1,X), try(1,2,Y).
try(1,1,X) :-
  not use_clause_literal(1,1).
try(1,1,X) :-
  use_clause_literal(1,1), different(X,mycoin), coin(X).
try(1,2,Y) :-
  not use_clause_literal(1,2).
try(1,2,Y) :-
  use_clause_literal(1,2), different(mycoin,Y), coin(Y).
```

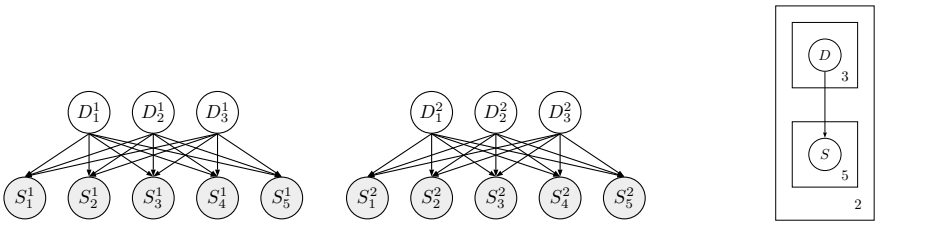
2.4 Probability Theory

2.4.1 Bayes Theorem

Bayes' Theorem is used for inference in probabilistic models. Given random variables A and B, a conditional probability $P(A|B)$, and a probability distribution $P(B)$, the conditional probability $P(B|A)$ can be computed:

$$P(B|A) = \frac{P(A|B)P(B)}{P(A)} \text{ where } P(A) = \int P(A|B)P(B)dB$$

This formula is called Bayes' Theorem[15]. Note that the computation of $P(A)$ can be avoided if $P(A|B)P(B)$ can be normalized. In the formula above, $P(B)$ is known as the prior, $P(B|A)$ is known as the posterior and $P(A|B)$ is known as the likelihood. Whenever the distribution family of $P(B|A)$ ends up being the same as the distribution family of $P(B)$, the distribution of the prior $P(B)$ is said to be a *conjugate prior* for the distribution of the likelihood $P(A|B)$.



(A) Complete Model

(B) Plate Notation

FIGURE 2.3: Basic Generative Probabilistic Model

2.4.2 Categorical Distribution

$$P(x = i|\boldsymbol{p}) = \boldsymbol{p}_i \text{ where } \boldsymbol{p} \in \mathbb{R}^{\mathbb{C}} \text{ and } |\boldsymbol{p}| = 1$$

The categorical distribution can be seen as a probability distribution on the outcomes of an event with C possible outcomes. A categorical distribution with 2 possible outcomes is also known as a Bernoulli distribution.

2.4.3 Dirichlet Distribution

$$P(\boldsymbol{p}; \boldsymbol{\alpha}) = \frac{1}{B(\boldsymbol{\alpha})} \prod_{i=1}^{K} \boldsymbol{p}_i^{\boldsymbol{\alpha}_i - 1} \text{ with } B(\boldsymbol{\alpha}) = \frac{\prod_{i=1}^{K} \Gamma(\boldsymbol{\alpha}_i)}{\Gamma(\sum_{i=1}^{K} \boldsymbol{\alpha}_i)} \text{ and } \Gamma(n) = (n-1)!$$

A Dirichlet distribution[17] can be seen as a distribution of distributions. An intuitive way to think of the Dirichlet distribution is that we have sampled several times from a categorical distribution and recorded the number of occurrences of each class in $\boldsymbol{\alpha}$. The Dirichlet distribution establishes which are the likely probabilities, $\boldsymbol{p}$, of the categorical distribution that we sampled from. A useful property of the Dirichlet distribution is that it is the conjugate prior for the categorical distribution. A Dirichlet distribution for 2 classes is also called a Beta distribution and the associated $\boldsymbol{\alpha}$ vector is denoted as a pair (α, β).

2.4.4 Generative Probabilistic Models and Plate notation

Generative Probabilistic Models are models that describe the generation of observable variables. In this report we consider Bayesian Networks[4], Probabilistic Models in which the relationship between the variables is described using conditional probabilities.
Example: The following example describes a basic generative model for diagnosing illnesses. A graphical representation of the model is presented in Figure 2.3a. Labelled circles represent random variables and coloured labelled circles represent observed variables. An arrow from variable B to A implies that variable A is defined by the conditional probability $P(A|B)$.
The model considers 2 people, 3 diseases and 5 symptoms.

1. For each person, the occurrence of each disease is represented by a binary variable $D \sim Bernoulli(p)$ for some constant p, suggesting that the occurrence of each disease is equally likely. It is also independent

from the occurrence of the other diseases and from the disease patterns of the other person.

2. The occurrence of symptoms is represented by conditional variables S that depend on D. We use a matrix to describe the probability distribution of the binary variable conditioned on each joint state of the diseases. We again assume that Symptoms are i.i.d.

$$P(S_i|D_1\&D_2\&D_3) = \begin{bmatrix} 0.4 & 0.6 & 0.2 & 0.5 & 0.8 & 0.2 & 0.1 & 0.1 \\ 0.6 & 0.4 & 0.8 & 0.5 & 0.2 & 0.8 & 0.9 & 0.9 \end{bmatrix}$$

The value of these models stems from the existence of latent (hidden) variables. Given a large number of observations, methods such as Bayes' theorem can be used to infer the the probability distributions of the latent variables. If the underlying model is an appropriate approximation of the real world problem, these values can be used to accurately predict the future observables. In the example provided, we can compute the posterior distribution of latent variables D, the probabilities of various illnesses given the data. The inferred probabilities can be used to predict symptoms in new observations, in this case new people.

It is important to note that models often make simplifying independence assumptions in order to make computations on the model tractable. Even when incorrect, some simplifications can still produce a model with useful predictive power, which should be verified empirically. In the disease occurrence model, it is assumed that people have independent disease patterns, which is clearly incorrect for transmittable diseases.

Plate notation is a way to represent repetition in graphical descriptions of generative probabilistic models. Plate notation can be used to produce clear and concise models when used judiciously. Uncircled symbols represent hyperparameters, circled symbols represent random variables, coloured circles represent observed variables, arrows between variables represent a conditional dependency from one variable to the other. Rectangles drawn around sets of random variables represent repetition, with the number of repetitions written in the lower right corner. As an example, the model described in this section is depicted in Figure 2.3b.

Note that the dependence between random variables in these models can be expressed in first-order logic as a set of clauses, resulting in an intuitive representation of the model. This justifies augmenting logic theories with probabilities to produce Probabilistic Logic Programming Languages. **Example:** A deterministic logic program that could represent the generative story of the illness occurrence model is:

```
nausea :- flu.
nausea :- poisoning.

runny_nose :- flu.
runny_nose :- allergy.
```

Note that this deterministic logic-based model does not possess any notion of randomness. In order for it to be useful, it needs to be somehow associated with the statistical model described in this section. This motivates the use of Probabilistic Logic Programs (see Section 2.5).

2.4.5 LDA

Latent Dirichlet Allocation[5] is a generative probabilistic model, a popular application of which is the inference of topic-document and word-topic categorical distributions with Dirichlet priors. This report uses a simplified version of the model in the original paper. An intuitive way to understand this model is by the following generative story, repeating it for each of the N words of a given document.

1. We sample a topic t from the topic-document μ_d distribution of the current document, d.
2. We sample a word w from the word-topic distribution ϕ_t of the sampled topic.

There are variations to this model that incorporate additional prior knowledge, such as seeded LDA.

2.4.6 Expectation-Maximization

As described in [8], common problem in statistics is inferring the parameters θ of a probability distribution $P(X|\theta)$ given samples X. A standard way to solve this problem is to use the Maximum Likelihood Estimate (MLE), i.e. to pick the parameters $\hat{\theta}$ which best explain the samples:

$$\hat{\theta} = \arg\max_{\theta} P(X|\theta)$$

This can then be solved with an optimization method.

Consider a more general probabilistic model $P(X, Y|\theta)$, which consists of observed variables X and latent variables Y. We can still attempt to use MLE to infer the parameters. Since we do not know the values of the latent variables, we have to marginalize over them:

$$\hat{\theta} = \arg\max_{\theta} P(X|\theta) = \arg\max_{\theta} \int P(X, Y|\theta) dY$$

A computational problem arises: the size of the domain of Y could be large which renders marginalization intractable. The Expectation-Maximization Algorithm (ME) can be used to approximate the MLE in this case.

The algorithm consists of two phases:

- In the **Expectation (E) step**, the expectation of the log likelihood function is computed with respect to the latent variables for a fixed estimate of θ:

$$Q(\theta) := E_Y[\log P(X, Y|\theta)]$$

- In the **Maximization (M) step**, the θ value is computed that maximizes the previously estimated Log Likelihood function:

$$\theta := \arg\max_{\theta} Q(\theta)$$

Note that the algorithm does not guarantee convergence towards the maximum likelihood estimate in the general case. Thus, additional domain-specific measures may need to be taken to ensure that the computation does not get stuck in a local maximum.

2.4.7 Markov Chain Monte Carlo (MCMC) Methods

In this report, we use MCMC methods to address the problem of estimating intractable probability distributions. Consider calculating the joint probability in a generative probabilistic model with latent variables $\boldsymbol{Y}$ and $\boldsymbol{Z}$ and observations $\boldsymbol{X}$.

$$P(\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{Z}) = \iint P(\boldsymbol{X}|\boldsymbol{Y}, \boldsymbol{Z})P(\boldsymbol{Y}|\boldsymbol{Z})P(\boldsymbol{Z})d\boldsymbol{Y}\,d\boldsymbol{Z}$$

Note that this computation can be intractable whenever the domain of $\boldsymbol{Y}$ or $\boldsymbol{Z}$ is large. So an empirical approximation can be constructed using sampling. Random Walk Markov Chain Monte Carlo methods perform sampling by building a Markov Chain on the sample space and then traversing non-deterministic paths in it, sampling the last element of each path. When appropriate transition probabilities are chosen, it can be guaranteed that the probability of sampling a specific element from the Markov chain converges to the target probability as the length of the path tends to infinity[4].

One MCMC method tailored for approximating joint probabilities is Gibbs Sampling. It assumes that sampling from the conditional probabilities $P(\boldsymbol{X}|\boldsymbol{Y}, \boldsymbol{Z})$, $P(\boldsymbol{Y}|\boldsymbol{X}, \boldsymbol{Z})$ and $P(\boldsymbol{Z}|\boldsymbol{X}, \boldsymbol{Y})$ is efficient. The algorithm starts out with an arbitrary sample $(\boldsymbol{X}_{(0)}, \boldsymbol{Y}_{(0)}, \boldsymbol{Z}_{(0)})$. Then each variable is sampled in a fixed order using the appropriate conditional probability, given the latest sampled values for the other variables. So, in this example, we sample $\boldsymbol{X}_{(1)}$ with probability $P(\boldsymbol{X}_{(1)}|\boldsymbol{Y}_{(0)}, \boldsymbol{Z}_{(0)})$, then $\boldsymbol{Y}_{(1)}$ with probability $P(\boldsymbol{Y}_{(1)}|\boldsymbol{X}_{(1)}, \boldsymbol{Z}_{(0)})$ and $\boldsymbol{Z}_{(1)}$ with probability $P(\boldsymbol{Z}_{(1)}|\boldsymbol{X}_{(1)}, \boldsymbol{Y}_{(1)})$. This procedure is repeated until enough examples are obtained to produce a sufficiently representative empirical distribution.[4]

Several things should be noted about this procedure. First, it is common to disregard some of the initial samples obtained as they are less likely to be representative of the target probability distribution. Moreover, the accuracy of the method could be increased by only considering every Nth sample for the empirical distribution and discarding all others. This is similar to increasing the length of the path that is traversed in the underlying Markov Chain, which would reduce the dependency between the samples and potentially increase the sampling accuracy.

Gibbs sampling can also be collapsed: rather than conditioning on all the sampled variables using $P(\boldsymbol{X}|\boldsymbol{Y}, \boldsymbol{Z})$, the conditional probability $P(\boldsymbol{X}|\boldsymbol{Z})$

can be used by marginalizing on the given variable (if tractable).

2.5 Probabilistic Logic Programs

Probabilistic logic programming languages are used to describe rule sets with uncertainty. Sato[34] points out that augmenting symbolic computation with probabilistic elements results in useful models, such as Hidden Markov Models or Bayesian Networks. This section introduces a semantics for Probabilistic Logic Programs and describes programming languages that are based on it.

2.5.1 Distribution Semantics

Sato's distribution semantics[34] provides a definition of Probabilistic Logic Programs. Programs consist of a set of ground probabilistic facts F and a set of ground definite clauses R whose heads are not predicates in F.

A probability measure on the interpretations of F can be extended to produce a probability measure on the least models of $F' \cup R$ for any $F' \subseteq F$. For any interpretation of F, a least model of $F' \cup R$ can be derived (where F' is the subset of all true facts in the interpretation) by iteratively applying the immediate consequence operator $T_{F' \cup R}$ on F'. This works because $F' \cup R$ is a definite clause theory. Then, the probability of least model M can be defined as the sum of the probabilities of all interpretations whose derived least model is M.

More formally, let Ω_F be the set of all interpretations of F. For an interpretation of F, ω, define $F_\omega \subseteq F$ as the subset of all true facts in F according to ω and M_ω as the unique minimal model of $F_\omega \cup R$ derivable from ω. Then, define for any possible subset of literals in $F \cup R$, $[\ell_1, \ell_2 \ldots \ell_n]_F = \{\omega \in \Omega_F | M_\omega \vDash \ell_1 \wedge \ell_2 \wedge \ldots \wedge \ell_n\}$. Given a probability measure on the set of all interpretations of F, P_F, we can define $P_{R \cup F}(M) = \{\sum_{\omega \in [\{a | a \in head(R) \wedge M \vDash a\}]_F} P_F(\omega)\}$.

Example:

$$
\begin{aligned}
F_1 &= \{weather(cloudy), forecast(rain)\} \\
R_1 &= \{bring(umbrella) \leftarrow weather(cloudy), \\
&\qquad bring(umbrella) \leftarrow forecast(rain)\}
\end{aligned}
$$

Provided in Figure 2.4a is a probability distribution on the interpretations in F_1. Based on that, the probability distribution of least models based on R_1 and subsets of F_1 is calculated in Figure 2.4b.

Note that the original definition of distribution semantics describes a distribution on definite programs and least models. The PLP languages described in this section are based on a notion similar to distribution semantics (as argued in [32]), with the caveat that they generalize the concept to normal logic programs. Later works attempt to generalize the original

Interpretation of F	Probability
$\emptyset$	0.4
$\{weather(cloudy)\}$	0.3
$\{forecast(rain)\}$	0.1
F	0.2

(A) P_ω on interpretations of F

Least Models of $\forall F' \subseteq F.P \cup F'$	Probability
$\emptyset$	0.4
$\{weather(cloudy), bring(umbrella)\}$	0.3
$\{forecast(rain), bring(umbrella)\}$	0.1
$F \cup \{bring(umbrella)\}$	0.2

(B) P_M on least models

FIGURE 2.4: Probability distributions on the program

formal definition of distribution semantics, culminating in [30], which attempts to redefine distribution semantics for arbitrary normal programs.

Sato also considers the problem of learning the program's probability distributions based on observations of clause heads of R. Sato proposes using an EM algorithm to find the pair of distributions that maximizes the likelihood of the observations given the distributions.

2.5.2 ProbLog

ProbLog [10] is a probabilistic logic programming language which is based on Prolog's syntax and SLDNF semantics. ProbLog programs consists of probability-clause pairs $\{p_1 : C_1, p_2 : C_2 \ldots p_n : C_n\}$. This induces a distribution on deterministic logic programs whose clauses are subsets of the clauses of the ProbLog program. The deterministic program is generated by considering each clause C_i independently and including it in the program with probability p_i.

The notion of query success in Prolog is generalized in ProbLog. Here we consider the probability that a query is successful. That is simply the sum of the probabilities of all deterministic programs in which the query succeeds. Query success in deterministic programs is determined in the same way as in Prolog: using the SLDNF proof procedure.

Example:

```
0.3::weather(cloudy).
0.1::forecast(rain).
0.9::bring(umbrella) :- weather(cloudy).
bring(umbrella) :- forecast(rain).
```

Note that when a probability of a clause is not included, it is assumed to be 1. In this example, a query `?- bring(umbrella).` would result in a probability of 0.343, since there are 5 deterministic programs with non-zero

Program	Probability
`forecast(rainy).` `bring(umbrella) :- forecast(rainy).`	$0.7 \times 0.1 \times 0.1 \times 1.0 = 0.007$

`weather(cloudy).`
`forecast(rainy).`
`bring(umbrella) :- forecast(rainy).`
	$0.3 \times 0.1 \times 0.1 \times 1.0 = 0.003$

`forecast(rainy).`
`bring(umbrella) :- weather(cloudy).`
`bring(umbrella) :- forecast(rainy).`
	$0.7 \times 0.1 \times 0.9 \times 1.0 = 0.063$

`weather(cloudy).`
`bring(umbrella) :- weather(cloudy).`
`bring(umbrella) :- forecast(rainy).`
	$0.3 \times 0.9 \times 0.9 \times 1.0 = 0.243$

`weather(cloudy).`
`forecast(rainy).`
`bring(umbrella) :- weather(cloudy).`
`bring(umbrella) :- forecast(rainy).`
	$0.3 \times 0.1 \times 0.9 \times 1.0 = 0.027$

Total:	0.343

FIGURE 2.5: The probability of query $bring(umbrella)$.

probability in which this query succeeds (as shown in Figure 2.5).

Note that the naive approach of computing query probability by considering every possible deterministic program separately is exponential in time complexity on the number of clauses, which is not practically efficient. In [10], a more efficient implementation is described which relies on the fact that each possible way to prove a query uses a set of clauses and the product of the probabilities of these clauses is equal to the sum of the probabilities of every deterministic program that includes these clauses. Query success can be expressed as a DNF formula on atoms b_i where b_i is true if and only if the clause C_i needs to be in the deterministic program for the query to succeed. The formula is a disjunction of all possible SLDNF proofs of the query, and each proof can be expressed as a conjunction of b_i atoms for every clause C_i necessary for the proof to succeed. Given the probabilities of b_i, the probability of the DNF formula being true can be efficiently computed by mapping the formula to a Binary Decision Diagram. A probability function can be recursively defined where the positive and negative terminal node have probabilities 1 and 0, respectively. The backward probability of internal node b_i can be computed using the recursive relation:

$$probability(b_i) = p_i * probability(b_{i+}) + (1 - p_i) * probability(b_{i-})$$

where p_i is the probability of including clause C_i in the program, and b_{i+} and b_{i-} denote the positive and negative child of node b_i, respectively.

Instead of doing the BDD computation after the SLDNF procedure has completed, an upper and a lower boundary on the probability can be computed by performing iterative deepening on SLDNF resolution. This means

that a reasonable approximation of the query probability can be computed without building the complete BDD. The approximation relies on the monotonicity of DNF formulas.

2.5.3 Logic Programs with Annotated Disjunctions and EMBLEM

Logic Programs with Annotated Disjunctions[39] (LPADs) are a probabilistic generalization of disjunctive logic programs. Clauses have deterministic bodies and a head which is a disjunction of atoms with a probability, all summing to one[2]. A deterministic normal logic program can be produced by grounding the clauses and independently choosing an atom for every head according to the probability of the heads. The probabilities of the heads induce a probability distribution on the deterministic programs that can be generated by the LPAD. The probability of a query is the sum of the probabilities of the deterministic programs whose total well-founded model entail the query.

Example:

```
1.0::bring(umbrella).
0.3::weather(cloudy), 0.1::forecast(rain) :- bring(umbrella).
```

The probability of the query `?- weather(cloudy).` is $1.0 \times 0.3 = 0.3$ since the only deterministic program whose well-founded model implies the query is:

```
bring(umbrella).
weather(cloudy) :- bring(umbrella).
```

Note that this is subtly different from the example in Section 2.5.2, since `weather(cloudy)` and `forecast(rain)` are both (mutually exclusive) possible heads of the second clause.

The EMBLEM[1] algorithm performs parametric learning - it is able to infer the probabilities of the heads of the clauses given observations by expressing all possible explanations as a boolean formula and converting it to a BDD (more precisely, to an Multiple Decision Tree, which can be redefined as a Binary Decision Tree) and then performing EM using the BDD to find the values of the parameters that maximize the likelihood of the data.

2.6 Statistical Abduction

Abduction is the task of inferring one or more explanations given background knowledge and some observations. In a deterministic logic programming paradigm, the Background Knowledge (BK) and Observations (O) are typically represented as clausal theories, whereas the explanations ϵ are represented as a subset of a set E of ground atoms, which are called *abducibles*. The computational task is to find one (or all) $\epsilon \subseteq E$ such that

[2]Riguzzi also considers heads whose probabilities sum to less than one. In that case, an implicit null head is inserted in the distribution. Intuitively speaking, choosing the null head means that the clause is discarded.

$B \cup \epsilon \models O$.

Statistical abduction is a probabilistic generalization of the abduction task which introduces reasoning about the likelihood of explanations. Sato and Ishihata[18] provide a detailed description of the computational task in the context of PLP and an efficient MCMC inference methodology. Statistical abduction becomes the task of inferring a probability distribution on explanations.

2.6.1 Peircebayes and Latent Dirichlet Analysis

Peircebayes[38] extends the probabilistic abductive procedure in [18] by supporting overlapping abductive explanations when assuming that all possible explanations can be enumerated. It is based on the following generative probabilistic model:

1. We start with a vector $\boldsymbol{\alpha}$ of Dirichlet priors to categorical distributions. Each element in this vector, $\boldsymbol{\alpha}_i$, is a vector of a_i values.

2. For each Dirichlet prior, we sample I_a many $\boldsymbol{\theta}_{ai}$ values, the probabilities of a categorical distribution.

$$\boldsymbol{\theta}_{ai}|\boldsymbol{\alpha}_a \sim Dirichlet(\boldsymbol{\alpha}_a)$$

3. For each of the N observations, we sample a from the categorical distributions.

$$\boldsymbol{x}_{nai}|\boldsymbol{\theta}_{ai} \sim Categorical(\boldsymbol{\theta}_{ai})$$

4. We can represent the value of the latent variables x by a set of conditionally independent boolean variables $\boldsymbol{v_{naij}}$.

5. For the current observation, we take a boolean function f of all the bits of the categorical distributions of that observation. And we say that this explains our observation so, the resulting f_n is always observed as a 1.

Peircebayes programs are based on the aforementioned semantics and extend Asystem abductive logic programs in the following way:

1. All categorical distributions and their corresponding Dirichlet priors are described using the fact `pb_dirichlet(Alpha, PredicateName, L, A).` This implies the existence of `A` categorical distributions with `L` categories named `PredicateName`, sampled from a Dirichlet distribution with parameters `Alpha`. The predicate `PredicateName(Category, DistributionNumber)` is true iff sample with `Category` was drawn from the categorical distribution identified by `PredicateName` and `DistributionNumber`.

2. A `pb_plate(OuterQuery,Count, InnerQuery).` predicate which iterates through the observations using the list `OuterQuery` as a SLDNF goal, and using `InnerQuery` as a goal of an abductive task, which represents the "generative story" of the given model.

```
generate(Word, Document) :-
        Topic in 1..10, mu(Topic, Document), phi(Word, Topic).

pb_dirichlet(1.0, mu, 10, 5).
pb_dirichlet(1.0, phi, 4, 10).

pb_plate([observe(word(Word), document(Document), count(Count))]),
        Count, [generate(Word, Document)]).

% Example observation:
observe(word(8), document(9), count(3)).
```

FIGURE 2.6: Latent Dirichlet Allocation in *Peircebayes*

Peircebayes allows us to write generative stories in an intuitive way. For instance, we can describe all the explanations of observing a word under an LDA model with 5 documents, 10 topics and a 4 word vocabulary using the Peircebayes program in Figure 2.6. The program assumes a symmetric Dirichlet prior of 1 for all categorical distributions.

The main task of Peircebayes is to determine a distribution of most likely explanations, i.e. to infer $P(\boldsymbol{\theta}|\boldsymbol{\alpha})$. Since the prior distribution is conveniently chosen to be a Dirichlet distribution, the prior conjugate of the categorical distribution, this task is equivalent to finding posterior values of $\boldsymbol{\alpha}$. [38] describes using Gibbs sampling along $\boldsymbol{\theta}$ and $\boldsymbol{x}$ to achieve that. The Peircebayes implementation can also use collapsed Gibbs sampling $P(\boldsymbol{x}_n|\boldsymbol{\alpha}, \boldsymbol{x} - \{\boldsymbol{x}_n\})$ by integrating out $\boldsymbol{\theta}$ for efficiency reasons. Sampling from the conditional probabilities used by Gibbs sampling is done by converting the observations to DNF formulas of conditionally independent boolean variables and then sampling from the resulting Binary Decision Diagram.

Chapter 3

Annotated Literal Programs

In this chapter, we define Annotated Literal Programs (ALP), a new type of probabilistic logic programs in which every literal of a normal logic program is associated with a Bernoulli or a Beta distribution. The reason why we defined this new language is that it is very intuitive to extend the XHAIL algorithm to perform structural and parameter learning on it. However, it turns out that an extra benefit of Annotated Logic Programs is that they have certain properties that could conceivably make them preferable to other state-of-the-art PLP languages:

- ALP uses stable set semantics which, compared to SLDNF semantics, allows for more expressive programs which can handle a large number of constraints.
- ALP can define more expressive probability distributions than a large number of existing PLP languages with the use of a Beta distribution.

In this chapter, we provide a formal definition of Annotated Literal Programs. First, we describe the syntax of the language. Then, we present two probabilistic models that define the probability of a query and thus represent the formal semantics of the language. We compare the expressiveness of this new language to the languages defined in Section 2.5.

3.1 Syntax

The PLP language has a syntax similar to ASP. A program $T =< T_B, T_{PROB} >$ consists of two parts: Background Knowledge T_B and probabilistic theory T_{PROB}. Background knowledge uses the input syntax of clingo 3[14]. We propose two types of programs depending on the probability distribution in T_{PROB}.

3.1.1 Unpriored Model

Informally, if T_{PROB} is defined without a prior, it can be written as a sequence of clauses whose literals are annotated with probabilities. Each probability is a parameter for a Bernoulli trial:

$$A : \theta_0 \leftarrow B_1 : \theta_1, B_2 : \theta_2 \ldots B_n : \theta_n$$

where A is a positive literal, $\{B\}$ are literals, and $\{\theta\}$ are rational values between 0 and 1.

Example: An example PLP model for the machine learning task in Figure 1.1 is the program $T^1 =< T^1_B, T^1_{PROB} >$ such that:

$$T^1_B = \begin{Bmatrix} 1\{heads(X), tails(X)\}1 \leftarrow coin(X). \\ coin(realcoin). \\ coin(biasedcoin). \end{Bmatrix}$$

$$T^1_{PROB} = \begin{Bmatrix} heads(realcoin) : 0.5 \\ tails(biasedcoin) : 0.9 \\ different(X,Y) \leftarrow heads(X), tails(Y). \\ different(X,Y) \leftarrow tails(X), heads(Y). \end{Bmatrix}$$

Missing probabilities in T_{PROB} are assumed to equal 1.0.

3.1.2 Priored Model

Alternatively, every literal can be annotated with a pair of positive rational numbers that represent a Beta Distribution :

$$A : (\alpha_0, \beta_0) \leftarrow B_1 : (\alpha_1, \beta_1), B_2 : (\alpha_2, \beta_2) \ldots B_n : (\alpha_n, \beta_n)$$

where A is a positive literal, $\{B\}$ are literals, and $\{(\alpha, \beta)\}$ are pairs of non-negative rational values.
Example: An example PLP model for the machine learning task in Figure 1.1 is the program $T^2 =< T^2_B, T^2_{PROB} >$ such that:

$$T^2_B = \begin{Bmatrix} 1\{heads(X), tails(X)\}1 \leftarrow coin(X). \\ coin(realcoin). \\ coin(biasedcoin). \end{Bmatrix}$$

$$T^2_{PROB} = \begin{Bmatrix} heads(realcoin) : (5,5) \\ tails(biasedcoin) : (90,10) \\ different(X,Y) : (101,1) \leftarrow heads(X) : (101,1), tails(Y) : (101,1). \\ different(X,Y) : (101,1) \leftarrow tails(X) : (101,1), heads(Y) : (101,1). \end{Bmatrix}$$

3.2 Formal Definition

Formally, we define the concept of clause sequences and probabilistic annotations. An *ordered clause* is defined as an extended definite clause, in which the body is a sequence, rather than a set, of literals. If S is an ordered clause sequence, S^i is defined as the ith clause in the sequence and S^i_j is defined as the jth body literal in the ith clause. S^i_0 is defined as the head of the ith clause. Let $head(c)$ be the head of ordered clause c, and let $body(c)$ be the sequence of body literals of clause c. $|S|$ is the number of clauses in ordered clause sequence S and $|body(c)|$ is the number of body literals in clause c.

Probabilistic annotations map literals to parameters of probability distributions. We define two types of probabilistic annotations: Bernoulli annotations and Beta annotations. A Bernoulli annotation $\boldsymbol{\theta}$ is a binary relation $(\mathcal{N} \times \mathcal{N}, \mathcal{Q})$ that maps pairs of integers to rational numbers in the range

$[0, 1]$. A Beta annotation $\boldsymbol{\tau}$ is a binary relation $(\mathcal{N} \times \mathcal{N}, (\mathcal{Q} \times \mathcal{Q}))$ that maps pairs of integers to pairs of positive real numbers.

A probabilistic annotation $\boldsymbol{\omega}$ is said to be *compatible* with a clause sequence S if and only if the domain of $\boldsymbol{\omega}$ is equal to $\{(i, j)|i \in [1, |S|], j \in [0, |body(S^i)|]\}$ and the mapping is one-to-one, i.e. every literal in S is associated with exactly one set of parameters.

Formally, an Annotated Literal Program T_{PROB} can be defined as a pair $< S, \boldsymbol{\omega} >$ where S is an ordered clause sequence, and $\boldsymbol{\omega}$ is a probabilistic annotation that is compatible with S. The unpriored model uses a Bernoulli annotation, whereas the priored model uses a Beta annotation.

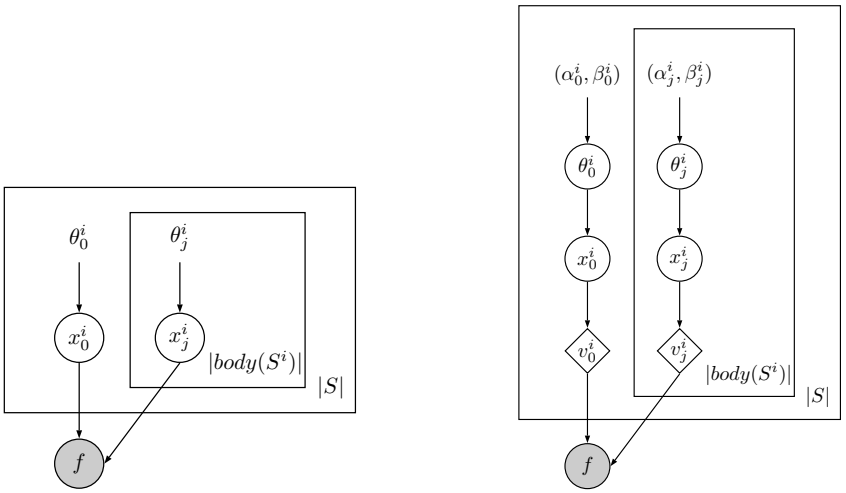
Example: The formal way to write programs T^1_{PROB} and T^2_{PROB} is:

$$
\begin{aligned}
T^1_{PROB} &= < S, \boldsymbol{\theta} > \\
T^2_{PROB} &= < S, \boldsymbol{\tau} > \\
S^1 &= heads(realcoin) \\
S^2 &= tails(biasedcoin) \\
S^3 &= different(X, Y) \leftarrow heads(X), tails(Y) \\
S^4 &= different(X, Y) \leftarrow tails(X), heads(Y) \\
\boldsymbol{\theta} &= \{((0,0), 0.5), \\
&\quad ((1,0), 0.9), \\
&\quad ((2,0), 1), ((2,1), 1), ((2,2), 1), \\
&\quad ((3,0), 1), ((3,1), 1), ((3,2), 1), \} \\
\boldsymbol{\tau} &= \{((0,0), (5,5)), \\
&\quad ((1,0), (90,10)), \\
&\quad ((2,0), (101,1)), ((2,1), (101,1)), ((2,2), (101,1)), \\
&\quad ((3,0), (101,1)), ((3,1), (101,1)), ((3,2), (101,1)), \}
\end{aligned}
$$

3.3 Semantics

Annotated logic programs describe a probability distribution of answer set programs. In order to make it easier to contrast ALP to languages with SLDNF semantics, the language doesn't define a probability distribution on answer sets. Brave entailment is used for query semantics, meaning that, for a given answer set program, a query is successful if it is true in at least one answer set of a program.

In order to formally define the semantics of the program, some additional notation is needed. We consider sequences with empty elements $\emptyset$. An ordered clause with empty elements can be interpreted as an extended definite clause whose body consists of all non-empty elements. Similarly, a clause sequence can be interpreted as a program by ignoring empty elements.



(A) Unpriored model (B) Priored model

FIGURE 3.1: Generative model for P_T

Formally, a program T defines a probability distribution over clause sequences $L \in \mathcal{L}_T$. The probability of a query q is the sum of the probabilities of all clause sequences that bravely entail it together with the background knowledge T_B:

$$P(q|T) = \sum_{L \in \mathcal{L}_T} p(q, L|T) = \sum_{L \in \mathcal{L}_T} P(q|L, T)P(L|T)$$

$$p(q|L, T) = \begin{cases} 1 & \text{if } L \cup T_B \models_b q \\ 0 & \text{otherwise} \end{cases}$$

where $\models_b$ denotes brave entailment.

We propose two different probabilistic models for the distribution of answer set programs based on T_{PROB}, $P(L|T) = P(L|T_{PROB})$.

3.3.1 Unpriored Model

Intuitively, each ALP defines a distribution over clause sequences with empty elements. The generative story for a single clause sequence given the program is described by the following sequence of steps:

1. Include each clause with probability θ_0^i, the probability of the head literal.
2. In each included clause, the head is always included. If the clause is included, consider the sequence of body literals independently and include each with its corresponding probability θ_j^i.
3. The generative story outputs a boolean variable which is true if the generated hypothesis is equal to the expected clause sequence.

The purpose for generating a boolean variable in the end is to make the model comparable to the priored model, which needs to be compatible with

peircebayes for parameter learning. Formally[1], we define the probabilistic model for a program $< S, \boldsymbol{\theta} >$:

1. For each $i \in [1, |S|]$ and each $j \in [0, |body(S)|]$, sample $x_j^i \sim Bernoulli(\theta_j^i)$.
2. Then, we sample a clause sequence with empty elements:

$$P(f|x) = \begin{cases} 1 & \text{if } f_S(X) = L \\ 0 & \text{otherwise} \end{cases}$$

where:

$$f_S(X) = \{L^i\}_{i=1}^{|S|}$$

$$L^i = \begin{cases} \emptyset & \text{if } x_0^i \\ clause < head(S^i), \{B_j^i\}_{j=1}^{|body(S)|} > & \text{if } \neg x_0^i \end{cases}$$

$$B_j^i = \begin{cases} \emptyset & \text{if } x_j^i \\ S_j^i & \text{if } \neg x_j^i \end{cases}$$

This is equivalent to:

$$P(L|T_{PROB} =< S, \boldsymbol{\theta} >) = \prod_{i=1}^{|S|} P(L^i| < S, \boldsymbol{\theta} >)$$

$$P(L^i| < S, \boldsymbol{\theta} >) = \begin{cases} 1 - \theta_0^i & \text{if } L^i = \emptyset \\ \theta_0^i \times \prod_{j=1}^{|body(L^j)|} P(L_j^i| < S, \boldsymbol{\theta} >) & if head(L^i) = head(S^i) \\ 0 & \text{otherwise} \end{cases}$$

$$P(L_j^i| < S, \boldsymbol{\theta} >) = \begin{cases} \theta_j^i & \text{if } L_j^i = S_j^i \\ 1 - \theta_j^i & \text{if } L_j^i = \emptyset \\ 0 & \text{otherwise} \end{cases}$$

The probability distribution is depicted in Plate notation in Figure 3.1a.

3.3.2 Priored Model

The priored model is similar to the unpriored one, except θ values are not explicitly used as input, but are sampled from a Beta distribution with parameters (α, β):

1. For every literal, sample a Bernoulli parameter $\theta_i^j \sim Beta(\alpha_j^i, \beta_j^i)$
2. Include each clause with probability θ_0^i, the probability of the head literal.
3. For each included clause the head is always included. Consider the sequence of body literals independently and include each with its corresponding probability θ_j^i.
4. The generative story outputs a boolean variable which is true if the generated hypothesis is equal to the expected clause sequence.

[1] We use θ_j^i to refer to $\theta(i, j)$

The purpose of generating a boolean variable in the end of the generative story is to make the model compatible with peircebayes, a statistical abduction model where the boolean variable determines whether the observation has been explained by the abducibles. We later show that modifying the boolean variable to return true whenever a partial interpretation is explained by the hypothesis corresponding to the $\boldsymbol{x}$ values will result in a statistical abduction model that can be used for parameter learning.
Formally:

1. For each $i \in [1, |S|]$ and each $j \in [0, |body(S)|]$, sample $\theta_j^i \sim Beta(\alpha_j^i, \beta_j^i)$.
2. For each $i \in [1, |S|]$ and each $j \in [0, |body(S)|]$, sample $x_j^i \sim Bernoulli(\theta_j^i)$.
3. Then, we sample a clause sequence with empty elements:

$$P(f|x) = \begin{cases} 1 & \text{if } f_S(X) = L \\ 0 & \text{otherwise} \end{cases}$$

where:

$$f_S(X) = \{L^i\}_{i=1}^{|S|}$$

$$L^i = \begin{cases} \emptyset & \text{if } x_0^i \\ clause < head(S^i), \{B_j^i\}_{j=1}^{|body(S)|} > & \text{if } \neg x_0^i \end{cases}$$

$$B_j^i = \begin{cases} \emptyset & \text{if } x_j^i \\ S_j^i & \text{if } \neg x_j^i \end{cases}$$

Note that Figure 3.1b has a set of v variables between x and f. The value sampled from v is always the same as the value sampled from x. The purpose of this redundant variable is to make the generative model compatible with peircebayes, where v is necessary to represent the x variable as a set of independent binary variables, as x might have more than 2 states in the general case. However, to keep probability calculations simpler, this variable is ignored in the rest of the report.

3.4 Comparison to Distribution Semantics

Comparing ALP to the languages described in Section 2.5 is difficult for several reasons. A notable difference between Annotated Literal programs and other PLPs is that entailment is not based on SLD semantics. ALP is based on Stable Set Semantics. Moreover, the original paper on distribution semantics[34] considers definite programs, whereas modern PLP languages define probability distributions on normal logic programs instead. For this reason, rather than relating the expressive power of annotated logic programs to any definition of distribution semantics, we compare it to ProbLog, a PLP language that has been proven equivalent to others[32].

3.4.1 Unpriored Model

For the unpriored model, we attempt to prove that annotated literal programs are equivalent to ProbLog programs for which stable semantics agrees

with SLDNF resolution. We do this by describing methods to translate between both languages.

Translation from ProbLog

A program is translated from Problog to annotated literal programs by simply using the clause probability as the head probability and setting all body probabilities to 1. Consider a ProbLog program $T_{Problog} = \{c_1 : \theta^1, c_2 : \theta^2 \dots c_n : \theta^n\} = \{(\theta_i, A^i \leftarrow B^i_1, B^i_2 \dots B^i_{n(i)}) | i \in [1..n]\}$ where $n(i)$ is the number of body literals in clause i and n are the number of clauses. Then, consider the program T with $T_B = \emptyset$ and $T_{PROB} = \{A^c : \theta^c \leftarrow B^c_1 : 1.0, B^c_2 : 1.0 \dots B^c_{n(c)} | c \in T_{Problog}\}$. These two programs describe the same distributions of hypotheses. ALP considers a subset of the full clauses (all the body literal probabilities are 1.0). Moreover, the probabilities of all deterministic programs are the same for both models: In T, the probability of each program can also be computed by considering each clause independently and adding it to the deterministic program with probability.
Example: The Problog program described in Section 2.5.2 is equivalent to the program (almost no transfomation is necessary):

$$T_B = \emptyset$$

$$T_{PROB} = \left\{ \begin{array}{l} weather(cloudy) : 0.3. \\ forecast(rain) : 0.1. \\ bring(umbrella) : 0.9 \leftarrow weather(cloudy) : 1.0. \\ bring(umbrella) : 1.0 \leftarrow forecast(rain) : 1.0. \end{array} \right\}$$

Theorem: Given a clause sequence S and a sequence of probabilites $\{p^i\}_{i=1}^{|S|}$, and a ProbLog program $T_{Problog} = \{S^i : p^i\}_{i=1}^{|S|}$, we can construct an annotated literal program T such that $P(q|T) = P(q|T_{Problog})$ for any query q.
Proof: We consider the probability of a query in the ProbLog program[10], which by definition is:

$$\begin{aligned} P(q|T_{Problog}) &= \sum_{L \subseteq T_{Problog}} p(q, L|T_{Problog}) \\ &= \sum_{L \subseteq T_{Problog}} P(q|L, T_{Problog}) P(L|T_{Problog}) \\ P(q|L, T_{Problog}) &= \begin{cases} 1 & \text{if } L \models q \\ 0 & \text{otherwise} \end{cases} \\ P(L|T_{Problog}) &= \prod_{c \in L} P(c) \prod_{c \in S - L} (1 - P(c)) \end{aligned}$$

where L is a set of clauses. If we redefine L as a clause sequence with empty elements, the expression for $P(L|T_{Problog})$ can be rewritten as:

$$P(L|T_{Problog}) = \prod_{i=1}^{|S|} p(L^i)$$

$$p(L^i) = \begin{cases} \theta^i & \text{if } L^i = S^i \\ 1 - \theta^i & \text{if } L^i = \emptyset \\ 0 & \text{otherwise} \end{cases}$$

Now consider a program $T =< T_B, T_{PROB} >$ with $T_B = \emptyset$ and $T_{PROB} =< S, \boldsymbol{\theta} >$ such that $\forall i \in [1, |S|].\theta_0^i = p^i$ and $\forall i \in [1, |S|].\forall j \in [1, |body(S^i)|]\theta_j^i = 1$. $\boldsymbol{\theta}$ is compatible with S which means that T is a valid ALP program. We see that $P(q|T_{PROB}) = P(q|T_{Problog})$ and $P(q|L, T_{PROB}) = P(q|L, T_{Problog})$ if we assume that we only consider programs where brave entailment and SLDNF resolution produce are equivalent (e.g. in definite programs, stratified normal programs). We need to show that $P(L|T_{PROB}) = P(L|T_{Problog})$. We have that:

$$P(L|T_{PROB} =< S, \boldsymbol{\theta} >) = \prod_{i=1}^{|S|} P(L^i| < S, \boldsymbol{\theta} >)$$

$$P(L^i| < S, \boldsymbol{\theta} >) = \begin{cases} 1 - \theta_0^i & \text{if } L^i = \emptyset \\ \theta_0^i & \text{if } head(L^i) = head(S^i) \wedge body(L^i) = body(S^i) \\ 0 & \text{otherwise} \end{cases}$$

Thus, $P(L|T_{PROB}) = P(L|T_{Problog})$ and so $P(q|T) = P(q|T_{Problog})$.
ProbLog and this PLP language use a different notion of entailment. ProbLog entailment is defined operationally by the SLDNF procedure, whereas ALP semantics is based on brave entailment of answer set programs. In order to prove that both probabilistic languages are equivalent, we have to restrict ourselves to PLPs which describe a distribution over programs for which $T \models_b q \iff T \models q$. In the general case, query success is not equivalent in SLDNF and stable set semantics, which means that the languages have a different expressive power.

Translation to ProbLog

Now, we show how to use ProbLog to define the same probability distributions on queries as annotated literal programs. The way to generate a ProbLog program $T_{Problog}$ is to encode the clauses of T_{PROB} in the same way as the Kernel Set in XHAIL during the inductive stage, as described in Section 2.3.2. The clauses of the XHAIL task should be deterministically entailed by $T_{Problog}$, i.e. each of the clauses should have probability 1. Then, each `use_clause_literal(i,j)` atom should be a separate fact in the program with probability θ_j^i. This probabilistic set of clauses is essentially a way to represent the $\boldsymbol{x}$ variables in the probabilistic model of annotated literal programs. This should result in the same probability distribution for queries as the one described by the unpriored model.

Example: The example program in Section 3.1.1 can be rewritten in ProbLog as:

```
%%%%%%%%%%%%%%%%%%%%%%%%
% Probabilistic part %
%%%%%%%%%%%%%%%%%%%%%%%%

% head(realcoin):0.5.
0.5::use_clause_literal(0,0).

heads(realcoin) :- use_clause_literal(0,0).

% tails(biasedcoin):0.9.
0.9::use_clause_literal(1,0).

tails(biasedcoin) :- use_clause_literal(1,0).

% different(X, Y) :- heads(X), tails(Y).
1.0::use_clause_literal(2, 0).
1.0::use_clause_literal(2, 1) :- use_clause_literal(2, 0).
1.0::use_clause_literal(2, 2) :- use_clause_literal(2, 0).

different(X, Y) :-
        use_clause_literal(2, 0),
        try(2, 1, X),
        try(2, 2, Y).

try(2, 1, X) :- \+ use_clause_literal(2, 1).
try(2, 1, X) :- coin(X), heads(X), use_clause_literal(2, 1).

try(2, 2, Y) :- \+ use_clause_literal(2, 2).
try(2, 2, Y) :- coin(Y), tails(Y), use_clause_literal(2, 2).


% different(X, Y) :- tails(X), tails(Y).
1.0::use_clause_literal(3, 0).
1.0::use_clause_literal(3, 1) :- use_clause_literal(3, 0).
1.0::use_clause_literal(3, 2) :- use_clause_literal(3, 0).

different(X, Y) :-
        use_clause_literal(3, 0),
        try(3, 1, X),
        try(3, 2, Y).

try(3, 1, X) :- \+ use_clause_literal(3, 1).
try(3, 1, X) :- coin(X), tails(X), use_clause_literal(3, 1).

try(3, 2, Y) :- \+ use_clause_literal(3, 2).
try(3, 2, Y) :- coin(Y), heads(Y), use_clause_literal(3, 2).

%%%%%%%%%%%%%%%
% Background %
%%%%%%%%%%%%%%%

% We need to redefine the choice rule in a Prolog-friendly
% way. And we need to be careful to avoid infinite loops.
% Which is why we need to ground them.
head(biasedcoin) :- \+ tails(biasedcoin).
tails(realcoin) :- \+ heads(realcoin).

coin(realcoin).
coin(biasedcoin).
```

Note that some changes were made to the original program since ProbLog can't do ASP-style choice rules. Overall, annotated literal programs appears to be better at dealing at dealing with constraints than SLDNF-based languages. If we restrict ourselves to programs with a single finite model, we still obtain equivalent programs.

An informal proof of this equivalence is that the ProbLog program defines a probability distribution on the `use_clause_literal` predicate in this program and this distribution is the same as the one defined by the priored model.

Theorem: Given an ALP Program $T =< T_B, T_{PROB} >$ with $T_B = \emptyset$ and $T_{PROB} =< S, \boldsymbol{\theta} >$, we can construct a ProbLog program $T_{Problog}$ such that $P(q|T) = P(q|T_{Problog})$.
Proof: We assume the existence of predicates `ucl/2` (short for *use clause literal*) and `try/varlength` that are not used in any clauses in S to define the following program:

$$T_{Problog} = T_{UCL} \cup T_{XHAIL}$$
$$T_{UCL} = \{ucl(i,j) : \theta_j^i\}_{i=1,j=0}^{i\leq|S|,j\leq|body(S^i)|}$$
$$T_{XHAIL} = T_{clause} \cup T_{try}$$
$$T_{clause} = \left\{ S_0^i \leftarrow ucl(i,0), \{try(i,j,vars(S_j^i)) : 1\}_{j=1}^{|body(S^i)|} \right\}_{i=1}^{|S|}$$
$$T_{try} = \left\{ try(i,j,vars(S_j^i)) \leftarrow / + ucl(i,j). : 1 \right\}_{i=1,j=1}^{i\leq|S|,j\leq|body(S^i)|}$$
$$\cup \left\{ try(i,j,vars(S_j^i)) \leftarrow S_j^i, ucl(i,j). : 1 \right\}_{i=1,j=1}^{i\leq|S|,j\leq|body(S^i)|}$$

where $const(L)$ denotes the constants of a literal L. Informally, the way to prove that this program is equivalent to T_{PROB} is to realize that the `ucl` predicates and its clause probabilities in ProbLog have an equivalent function to the random variables x and θ in ALP. So we can associate each configuration of $\boldsymbol{x}$ with a subsequence of T_{UCL} such that $x_j^i \iff ucl(i,j)$. Then, in both cases, the probability of drawing a hypothesis from the model is the probability that each literal in the hypothesis is included.

3.4.2 Priored model

The priored model is more general than PLPs based on distribution semantics. By associating a Dirichlet prior with each literal, the priored model is essentially defining a probability distribution on unpriored models with an identical structure and varying parameters. Equivalently, the priored model is defining a probability distribution on ProbLog programs which have the same clauses but different parameters.

Chapter 4

ASP input programs in Peircebayes

It can be shown that parameter learning on annotated literal programs can be done using Peircebayes. However, the syntax of Peircebayes is based on Prolog, and hence Peircebayes is restricted to performing statistical abduction on Prolog programs. Annotated literal programs are based on stable set semantics and are evaluated with clingo. For this reason, we modify Peircebayes to accept answer set input programs.

We note that we gain an extra benefit of defining ASP programs for Peircebayes. Constraints can be efficiently expressed in answer set programs, as opposed to programs using SLDNF semantics. For this reason, we redefine other Peircebayes tasks in ASP and compare their runtime to their original Prolog-style definitions.

4.1 Brave abduction in ASP

In order to use ASP programs with Peircebayes, we need to modify the logical stage of the program. For SLDNF programs, top-down abduction is executed during that step in order to enumerate all possible ways in which Peircebayes' probabilistic predicates can explain the observations. Thus, we need a way to compute the abductive stage of Peircebayes in ASP. We can express an abductive task in ASP in the following way[22]:

- The set of abducibles is put in a choice rule with no body and no restriction on count.
- The knowledge base and integrity constraints are used as is.
- The goal is represented by a clause with head `goal` and contains the set of observed ground instances in its body. In addition, an extra constraint `:- not goal.` is added.

This is basically a generate-and-test approach to abduction, however ASP solvers are often able to optimize the typically intractable search for solutions implicitly.

Example:

$$KB = \left\{ \begin{array}{c} wobblyWheel \leftarrow brokenSpokes. \\ wobblyWheel \leftarrow flatTyre. \\ flatTyre \leftarrow leakyValve. \\ flatTyre \leftarrow puncturedTube. \leftarrow puncturedTube, leakyValve. \end{array} \right\}$$

$$O = \{wobblyWheel\}$$

$$Ab = \{brokenSpokes, puncturedTube, leakyValve\}$$

The task can be translated as:

$$P_{ASP} = KB \cup \left\{ \begin{array}{c} 0\{brokenSpokes, puncturedTube, leakyValve\}3. \\ goal \leftarrow wobblyWheel. \\ \leftarrow not\ goal. \end{array} \right\}$$

4.2 Automatic translation of Peircebayes input programs

In order to compare the runtime of various Peircebayes tasks in Prolog and ASP, we need to rewrite them. This needs to be done carefully, as certain Prolog programs that rely on the top-down SLDNF are very inefficient to compute bottom-up since they result in very large groundings. In this section, we consider several steps that can be taken to automatically convert a Peircebayes input program to ASP. For every grounding, we generate one ASP program that contains a single grounding of `outer_query`. For each outer query grounding, we have a single ASP whose answer sets represent different solutions to the abductive task inside. Then, we construct the rest of the ASP program as an abductive task with `inner_query` as the goal and each Dirichlet predicate as the abducibles.

1. Copy in the background knowledge and the grounding of a single outer query.
2. Copy in the inner query from the Peircebayes input program and convert it to a clause with head `inner_query` and an unground literal `outer_query` in the body.
3. Add the inner query as a goal:

```
goal :- inner_query.
:- not goal.
```

4. For every distribution, generate:

```
0 {distribution_name(K, I) : distribution_name_category(K) } 1
        :- distribution_name_sample_number(I).
```

   where `distribution_name` is the name of the probabilistic literal.
5. Generate a minimization statement that weighs any distribution predicate equally.

As previously noted, the transformation above may often result in huge groundings as the original Prolog programs are optimized for top-down computation.

Task	Prolog	ASP-Parallel	ASP-Serial
fast LDA	1.152s	0.495s	0.942s
slow LDA		0.588s	1.207s
seeded LDA	2057.388s	307.474s	454.830s
RIM	9.830s	3.051s	6.982s

FIGURE 4.1: Comparison of Peircebayes tasks in Prolog and ASP

Note that these steps only work with programs which are purely declarative. Prolog programs have procedural semantics, which cannot be automatically captured in ASP. For example, cutting is a meaningless operation in bottom-up semantics where the order of body literals is irrelevant. Moreover, Prolog programs have richer library support than modern ASP solvers. To complicate the problem further, the lack of support for some libraries in ASP can be justified by the lack of efficient ways to implement them. For example, implementing Prolog's list interface (`append/3`, `member/2`, etc) would require an intractably large grounding in clingo.

It is, however, possible to modify part of the query to avoid redundant groundings of the abducibles. Currently, there are no restrictions on abducibles, so any subset of them is eligible for generate-and-test (subject to ASP solver optimizations). The order of body predicates in Prolog is significant and body predicates used before abducibles in bodies of Prolog clauses can be used to reduce the grounding in bottom up computation. For example, a clause $a(X) \leftarrow b(X,Y), c(Y,Z), abducible(Z)$ can be rephrased during abduction with goal a in bottom-up as:

$$\{abducible(Z) : typeOfZ(Z)\} \leftarrow a(X), b(X,Y), c(Y,Z).$$

An example of these techniques can be seen in the reimplementation of the LDA algorithm in Section 4.3.1.

4.3 Evaluation

In this section, we consider implementations of several Peircebayes problems in ASP, compare logical inference runtime to Prolog programs. The tests are executed on a 4GB machine with an Intel(R) Core(TM) i7-4980HQ CPU @ 2.80GHz processor. The execution time reported is an average of 10 executions times for each task. We use datasets that Peircebayes has previously been evaluated on. The results are summarized in Figure 4.1 and are explained in the rest of this section.

It is important to note that, for abduction in ASP, we split the observations into several parts and run these on different processes. Answer set program execution time does not scale linearly with the number of observations so the observations have to be split into several answer set programs. We haven't split the observations in such a way for the Prolog implementation. So we show the runtime for Peircebayes, for serial execution of the

partitioned abductive task, and for a parallel implementation that uses a process pool.

4.3.1 LDA and CLDA

We attempted to automatically translate LDA using the techniques described in Section 4.2. The original program is:

```
pb_dirichlet(1.0, mu, 10, 1000).
pb_dirichlet(1.0, phi, 25, 10).

generate(Doc, Token) :-
    Topic in 1..10,
    mu(Topic,Doc),
    phi(Token,Topic).

pb_plate(
    [observe(doc(Doc), TokenList),
         member((word(Token), count(Count)), TokenList)],
    Count,
    [generate(Doc, Token)]
).

observe(doc(1), [(word(1), count(1)),(word(3), count(5)),...).
...
```

The `pb_dirichlet` facts identify the abducible predicates as `mu` and `phi`. The `pb_plate` facts define the abductive goals of the program. First, the query in the first argument (known as the outer query) is executed and the variable grounding is used to define an observation. This grounding method is a feature of Peircebayes that allows for several observations to share the same BDD. For each grounding of the variables in the outer query, top-down minimizing abduction is executed, in which the goal is to explain the observation, the third argument of `pb_plate` grounded with the given outer query results.

An initial hurdle is circumventing the use of Prolog lists. Unfortunately, this cannot be done automatically, however it is straightforward to circumvent manually. Each `observe` fact can be transformed from a fact that maps a document to the list of all word-frequency pairs associated with it to a set of facts that contain a document-word-frequency) tuple.

Then, we can consider automatic translation techniques. First, we use the background in order to ground the outer query:

```
% Clingo directives to output only outer_query groundings.
#hide.
#show outer_query/3.

% Outer query.
outer_query(Doc, Token, Count) :-
        observe(doc(Doc),
        word(Token),
        count(Count)).

% Background knowelge.
```

```
topic(1..10).
generate(Doc, Token) :-
    topic(Topic),
    mu(Topic,Doc),
    phi(Token,Topic).


observe(doc(1), word(1), count(1)).
observe(doc(1), word(3), count(5)).
```

Each grounding of the outer query is used to ground the inner query. So we repeatedly assert each grounding and perform the abductive task:

```
%Asserting a single grounding of the outer query.
outer_query(1, 1, 1).

%Abductive task.
%%%%%%%%%%%%%%%%

% Whitelist predicates we output in answer set.
#hide.

%Abducibles
#show mu/2.
mu_sample_number(1..1000).
mu_category(1..10).
0 {mu(K, I) : mu_category(K) } 1 :- mu_sample_number(I).

#show phi/2.
phi_sample_number(1..10).
phi_category(1..25).
0 {phi(K, I) : phi_category(K) } 1 :- phi_sample_number(I).

% Abductive goal is to explain inner_query.
inner_query :- outer_query(Doc, Token, Count), generate(Doc, Token).
goal :- inner_query.
:- not goal.

% We want minimizing abduction, i.e. to minimize the number of
% abducibles in each explanation.
#minimize{ mu(K, I) : mu_category(K) : mu_sample_number(I),
phi(K, I) : phi_category(K) : phi_sample_number(I)}.

% Background knowledge
topic(1..10).

generate(Doc, Token) :-
    topic(Topic),
    mu(Topic,Doc),
    phi(Token,Topic).

observe(doc(1), word(1), count(1)).
observe(doc(1), word(3), count(5)).
```

Note that this generates a very big grounding and a very slow ASP optimization task. The number of abducibles generated is very large which results in a exceedingly large grounding. However, we can reduce the number of abducibles by inverting the inner query, as described in Section 4.2.

```
%Asserting a single grounding of the outer query.
outer_query(1, 1, 1).

%Abductive task.
%%%%%%%%%%%%%%%%

% Whitelist predicates we output in answer set.
#hide.

%Abducibles
#show mu/2.
mu_sample_number(1..1000).
mu_category(1..10).
0 {mu(Topic, Doc) : mu_category(Topic) } 1 :-
        mu_sample_number(Doc),
        outer_query(Doc, Token, Count),
        topic(Topic).

#show phi/2.
phi_sample_number(1..10).
phi_category(1..25).
0 {phi(Token, Topic) : phi_category(Token) } 1 :-
        phi_sample_number(Topic),
        outer_query(Doc, Token, Count),
        topic(Topic),
        mu(Topic, Doc).

% Abductive goal is to explain inner_query.
inner_query :-
        outer_query(Doc, Token, Count),
        generate(Doc, Token).
goal :- inner_query.
:- not goal.

% We want minimizing abduction, i.e. to minimize the number of
% abducibles in each explanation.
#minimize{ mu(K, I) : mu_category(K) : mu_sample_number(I),
phi(K, I) : phi_category(K) : phi_sample_number(I)}.

% Background knowledge
topic(1..10).

generate(Doc, Token) :-
    topic(Topic),
    mu(Topic,Doc),
    phi(Token,Topic).

observe(doc(1), word(1), count(1)).
observe(doc(1), word(3), count(5)).
```

This results in a significantly faster implementation that takes roughly 0.20 seconds to execute for the two observations given. Note that all predicates in the inner query that precede the abducibles in the body of the Prolog clause are used to restrict their grounding. Doing this in an automatic manner, however, does not seem achievable in non-trivial situations.

We compare the performance of LDA logic inference on a synthetic dataset with 100 documents, 10 topics and a vocabulary of 25 words (the

data has been generated according to the generative story described in Section 2.4.5). For the ASP representation, we split the observations in 75 partitions. Then we record average runtime over 10 executions for both Prolog and ASP tasks. We notice that the execution time for both tasks is comparable when ASP is executed serially. Unsurprisingly, the parallel ASP implementation is faster than the serial one and, hence, than the Prolog implementation.

In Appendix A we show a manually written Answer Set program that produces equivalent answer sets to the original one, however it does not explicitly define a generate-and-test minimal abductive task. In fact, the answer set program does not feature any minimization statements. After measuring the runtime on the same dataset, it can be concluded that the serially executed manual implementation is roughly 15% faster than the Prolog implementation for the given input size. In addition, the parallel manual implementation is faster than the parallel abductive one.

An extension of this task, called seeded LDA[19] (also shown in Appendix A), applies lexical priors to the models. This is done by a `seed` predicate which associates words with specific topics. Our evaluation on part of the 20 newsgroups dataset[21] suggests that a manually written ASP implementation is significantly faster than a Prolog implementation. We have not identified why that is the case.

4.3.2 RIM

RIM[13] is a probabilistic model for ranking preferences. This task had to be completely redesigned in order to benefit from bottom-up computation. We also show a significant speed up of the computation. The task is executed on the Sushi[20] dataset. The code for the implementation is available in Appendix A. We note an improvement in performance for the ASP implementation, as opposed to the Prolog one.

Chapter 5

Parameter Learning

In this chapter, we discuss the procedure used to do parameter learning in Annotated Literal Programs. The way this is achieved is by transforming ALP programs in a different PLP language and using previously developed BDD algorithms to learn the parameters for APL programs.

5.1 Partial Interpretations

First, we formalize the concept of partial interpretations as in [23]:
A partial interpretation $E =< E^+, E^- >$ consists of two sets of ground atoms, such that all atoms in E^+ are true in the interpretation, and all atoms in E^- are false. An interpretation I is said to extend a partial interpretation E if $E^+ \subseteq I$ and $E^- \cap I = \emptyset$.

5.2 Priored Model

The parameter learning task that we aim at solve is to evaluate the posterior distribution of θ. In particular, given a set of observations $\mathcal{O}$, which are represented by partial interpretations, we want to find the posterior expectation of the Bernoulli parameters $E_{P(\boldsymbol{\theta}|\mathcal{O},\boldsymbol{\tau})}\{\theta\}$ and the Beta parameters $E_{P(\boldsymbol{\alpha}|\mathcal{O},\boldsymbol{\tau})}\{\alpha\}$ and $E_{P(\boldsymbol{\beta}|\mathcal{O},\boldsymbol{\tau})}\{\beta\}$.

If we observe the probabilistic model defined by the priored ALP (shown in Figure 3.1b), we can see that it is expressible in Peircebayes. Thus, we can compute the posterior expectation of the parameters of an annotated literal program by using the statistical abduction procedure in Peircebayes.

We can adapt the ProbLog transformation described in Section 3.4.1 in order to learn a probability distribution on the abducibles. We can write an ASP abductive task in Peircebayes as described in Section 4.1.

In the parameter learning program, we associate each `ucl` with a categorical variable with a Dirichlet prior with the specified hyperparameters and encode the abductive solutions in a format understandable by Peircebayes.
Formally, for a program $T =< T_B, < S, \boldsymbol{\theta} >>$, the statistical abduction task can be specified as:

- Background T:

$$
\begin{aligned}
T =& T_{clause} \cup T_{try} \\
T_{clause} =& \left\{ S_0^i \leftarrow ucl(i,0), \{try(i,j,vars(S_j^i)) : 1\}_{j=1}^{|body(S^i)|} \right\}_{i=1}^{|S|} \\
T_{try} =& \left\{ try(i,j,vars(S_j^i)) \leftarrow notucl(i,j). \right\}_{i=1,j=1}^{i \leq |S|, j \leq |body(S^i)|} \\
& \cup \left\{ try(i,j,vars(S_j^i)) \leftarrow S_j^i, ucl(i,j). \right\}_{i=1,j=1}^{i \leq |S|, j \leq |body(S^i)|}
\end{aligned}
$$

  where $vars(L)$ represents all the variables in literal L

- Each observation $O \in \mathcal{O}$, one per abductive task.

$$
\begin{aligned}
& \leftarrow goal. \\
& goal \leftarrow O_1^+, ...O_n^+, notO_1^-, ...notO_m^-. \\
& \text{where} \\
& < \{O_1^+, ...O_n^+\}, \{O_1^-, ...O_m^-\} > \in \mathcal{O}
\end{aligned}
$$

- Abducibles `ucl` for every literal in the clause sequence.

Example: The clause sequence in Section 3.1.1 can be rewritten as[1]:

```
goal :- heads(biasedcoin), heads(realcoin),
        not different(realcoin, biasedcoin).
:- goal.

{ucl(i,j)} :- literal(i, j).
literal(0,0).
literal(1,0).
literal(2,0).
literal(2,1).
literal(2,2).
literal(3,0).
literal(3,1).
literal(3,2).

heads(realcoin) :- ucl(0,0).

tails(biasedcoin) :- ucl(1,0).

different(X,Y) :- ucl(2,0), try(2,1,X), try(2,2,Y).
try(2,1,X) :- not ucl(2,1), coin(X).
try(2,1,X) :- heads(X), ucl(2,1), coin(X).
try(2,2,Y) :- not ucl(2,2), coin(Y).
try(2,2,Y) :- tails(Y), ucl(2,2), coin(Y).

different(X,Y) :- ucl(3,0), try(3,1,X), try(3,2,Y).
try(3,1,X) :- not ucl(3,1), coin(X).
try(3,1,X) :- heads(X), ucl(3,1), coin(X).
try(3,2,Y) :- not ucl(3,2), coin(Y).
try(3,2,Y) :- tails(Y), ucl(3,2), coin(Y).
```

[1]We use a slightly more complicated abductive task that does type matching

Note that we can reduce the grounding of the ASP program by only adding variables to `try` predicates that are used by more than 1 literal. This has been implemented in our parameter learning program.

Here we note an additional advantage of using ASP for abduction is that we are able to perform complete abduction to compute any hypothesis that is consistent with the observations, rather than minimal abduction which only computes the minimal hypotheses that do so, a property of Prolog style abduction.

5.3 Unpriored model

We use EMBLEM[1] in order to perform parameter learning on the unpriored model. Note that EMBLEM works on LPADs rather than on programs in our PLP. In order to avoid the problem of supporting LPADs in ASP, we perform deterministic abduction of the ASP program in Section 3.4.1 and then use the abductive solutions to construct an LPAD, for which the parameters learned will be equivalent to the MLE parameters of a program based on the unpriored model.
In order to do this, we do the following:

- Define an abductive task on the background program for each observation, as defined in 3.4.1, with the observation as a goal and all ground atoms `use_clause_literal` as abducibles.
- A separate LPAD clause is generated for each `use_clause_literal` with some initial θ probability.
- In the background, each observation is associated with a set of clauses, one for each abductive solution. In every clause, the set of abducibles used are put in the body predicate. The head is an atom named `observation(id).`
- Each query/interpretation is a single literal, `observation(id).`

Example: For some sequence S and observations $\mathcal{O}$, we have run abduction in ASP and we have figured out that the first observation can be explained by 1 hypothesis, and the second observation can be explained by 2 hypotheses. We produce the following program.

```
:- begin_in.
ucl(0,0):0.5.
ucl(0,1):0.5.
ucl(1,0):0.5.
ucl(1,1):0.5.
:- end_in.

% Ways to generate observation
observation(1) :- ucl(0,0), ucl(0,1), ucl(1,0), ucl(1,1).

observation(2) :- ucl(0,0).
observation(2) :- ucl(1,0).

%observations
begin(model(m1)).
```

```
observation(1).
end(model(m1)).

begin(model(m2)).
observation(2).
end(model(m2)).
```

Chapter 6

PILP with PROBXHAIL

6.1 PILP Learning task

6.1.1 Unpriored model

For the unpriored model, we attempt to find a hypothesis that maximizes the likelihood of the problem given the data. Formally, we are given:

- A bag $\mathcal{E}$ of partial interpretations E.
- A bias M, consisting of mode declarations and determinations.
- Background knowledge B in ASP

The goal is to compute a hypothesis H which abides to bias M such that:

$$H = \arg\max_{H} p(\mathcal{E}|H)$$

6.1.2 Priored model

Point Estimate

We are given:

- A bag $\mathcal{E}$ of partial interpretations E.
- A bias M, consisting of mode declarations and determinations.
- Background knowledge B in ASP

We assume that the probability distribution is generated from the unprior model and we want to find the values of θ that were used in a process.The goal is to compute a hypothesis H which abides to bias M such that:

$$< S^*, \boldsymbol{\theta}^* >= \arg\max_{<S^*,\boldsymbol{\theta}^*>} \sum (\theta_j^i - \theta_j^{*i})^2.$$

Note that the Bayesian estimator of that value for a compatible sequence S is the posterior expectation $E_{p(\theta|\mathcal{E},S)}\{\theta\}$[7].

Bayesian Method

A more Bayesian approach to consider is to obtain the posterior expectation of the Beta parameters, $E_{p(\alpha|\mathcal{E},S)}\{\alpha\}$ and $E_{p(\beta|\mathcal{E},S)}\{\beta\}$. Then we can use this to evaluate the probability of a new example given previous data:

$$p(E^{new}|\mathcal{E}) = \int P(E^{new}|\boldsymbol{\theta})P(\boldsymbol{\theta}|\boldsymbol{\alpha},\boldsymbol{\beta})P(\boldsymbol{\alpha},\boldsymbol{\beta}|\mathcal{E})d\theta$$

However, note that we don't actually do PILP with PROBXHAIL based on this technique since cannot justify the use of a most specific hypothesis in this method.

6.2 Algorithm

6.2.1 Input

The input of the program is:

- A bag $\mathcal{E}$ of partial interpretations E.
- A bias M, consisting of mode declarations, determinations, and clause and literal repetition count limits
- Background knowledge B, written in ASP.
- A list of input and output predicates. Output predicates can be used in the constructed ALP and are abductive goals for parameter learning. Input predicates are not used to construct the ALP. Ground input facts are added in $\mathcal{E}$ to assign types to constant terms.

6.2.2 Description

The algorithm consists of two stages: structural learning and parameter learning. During the structural learning stage, the algorithm computes a clause sequence that is able to entail any possible hypothesis (subject to the mode declarations). In the parameter learning stage, a set of parameters is learned that attempt to complete one of the learning tasks defined in Section 6.1.

6.2.3 Structure Learning

In PROBXHAIL, the structural learning stage is guided solely by the bias and the mode declarations. Note that in Chapter 9, we discuss how to use a HAIL style Kernel Set for structural learning, so that the data can be used to construct a smaller most-specific hypothesis.

The algorithm first uses the mode declarations to construct a hypothesis consisting of ground literals. Then, the hypothesis is "lifted", i.e. constant terms are replaced with variables to match the requirements of the mode declarations.

1. Using all predicates in the head mode, we build a bag Θ of ground atoms of the predicates which, in order to be grounded with the right types of terms, use the type information in $\mathcal{E}$. Note that the type information is represented by atoms of input predicates and it is per observation. While the type information could have equivalently been specified separately and not per-observation, we use this so that the algorithm can be more easily extended.
2. For every atom $\alpha \in \Theta$, we consider every predicate in a body mode declaration that is allowed by the determination statements, and use

them, along with the type information in $\mathcal{E}$, in order to construct a bag of all ground atoms Δ that match the bias of the body. This results in a clause $\alpha \leftarrow \Delta$, and all such rules are joined in the clause sequence $H_\perp$ in some arbitrary order.

3. "Lift" the hypothesis $H_\perp$ and produce $H_\top$. For every literal, consider a mode declaration that is compatible with it and substitute terms with variables in accordance to the mode declaration.

4. Remove repetition of clauses/literals in a mode-compliant way

Example: Given:

- Example Set

$$\mathcal{E} = \Big\{ < \{heads(biasedcoin), tails(realcoin), coin(biasedcoin), coin(realcoin), different(realcoin, biasedcoin)\}, \{\} >;$$

$$< \{heads(biasedcoin), tails(realcoin), coin(biasedcoin), coin(realcoin)\}, \{different(realcoin, biasedcoin)\} > \Big\}$$

- Bias

$$M = \big\{ modeh(*, different(+coin, +coin)),$$
$$modeh(*, heads(\#coin)),$$
$$modeb(*, heads(+coin)),$$
$$modeb(*, tails(+coin)),$$
$$determination(different/2, heads/1),$$
$$determination(different/2, tails/1),$$
$$\texttt{Max repetition of body literals : 2}$$
$$\texttt{Max repetition of clauses : 2} \big\}$$

- Empty background B
- Input predicate `coin` and output predicates `heads`, `tails` and `different`

We proceed with the algorithm:

1. We obtain the set of clause heads:

$$\Theta = \{heads(realcoin), heads(biasedcoin), different(realcoin, biasedcoin), different(biasedcoin, realcoin)\}$$

2. We obtain the set of body literals and construct a ground hypothesis $H_\perp$:

```
heads(biasedcoin).
heads(realcoin).
different(realcoin, biasedcoin) :-
        heads(realcoin), tails(realcoin),
```

```
        heads(fakecoin), tails(fakecoin).
different(biasedcoin, realcoin) :-
        heads(realcoin), tails(realcoin),
        heads(fakecoin), tails(fakecoin).
```

3. We unground the hypothesis in order to produce $H_\top$:

```
heads(biasedcoin).
heads(realcoin).
different(X, Y) :-
        heads(X), tails(X),
        heads(Y), tails(Y).
different(X, Y) :-
        heads(X), tails(X),
        heads(Y), tails(Y).
```

4. Clause and literal repetition is already compliant with mode declaration, so the result of structure learning is $H_\top$.

6.2.4 Parameter Learning

Parameter Learning for the unpriored and priored model then occurs as in Section 5.3 and 5.2, respectively. Note that the parameter learning tasks optimizes the objective function of the corresponding PILP task for a fixed hypothesis. To be more specific, given a clause sequence S, unpriored parameter learning computes a Bernoulli annotation such that:

$$\boldsymbol{\theta} = \arg\max_{\boldsymbol{\theta}} p(\mathcal{E}| < S, \boldsymbol{\theta} >)$$

In addition, given a priored annotated logic program $< S, \boldsymbol{\tau} >$, priored parameter learning computes a Bernoulli annotation which is a Bayes Estimator for the least squared error.

6.3 Proof of correctness

6.3.1 Unpriored model

The way to prove correctness is to prove that the most specific PLP program is able to express the probability distribution of every most specific hypothesis and hence we can find a probabilistic annotation θ that subsumes it.

We define the subset relation for two clause sequences: $S_1 \subseteq S_2$ if and only if every clause in $S1$ is a subsequence of the corresponding clause in $S2$. An ordered clause $c1$ is a subsequence of $c2$ if and only if they have the same head and the body of $c1$ is a subsequence of the body in $c2$.

First, we consider the following lemma.

Lemma: For every pair of clause sequences $S_1 \subseteq S_2$ and probabilistic annotation $\boldsymbol{\theta}_1$ compatible with S_1, there exists a probabilistic annotation $\boldsymbol{\theta}_2$ compatible with S_2 such that for any query q, $P(q| < S_1, \boldsymbol{\theta}_1 >) = P(q| < S_2, \boldsymbol{\theta}_2 >)$.

Proof: Choose $\boldsymbol{\theta}_2$ such that $\forall(i,j) \in dom(S_1).\boldsymbol{\theta}_2(i,j) = \boldsymbol{\theta}_1(i,j)$ and $\forall(i,j) \in dom(S_2) - dom(S_1).\boldsymbol{\theta}_2(i,j) = 0$. This is basically sets the probabilities of all literals that are in S_2 but not in S_1 to 0, which means that S_2

induces the exact same probability distribution as S_1.

Now, note that the most-specific sequence S_{bot} constructed in the second part of PROBXHAIL is a supersequence of every other sequence that can be defined given the bias. Hence, the most-specific sequence can describe any probability distribution of hypothesis that the bias allows. Hence, there exists a set of parameters $\boldsymbol{\theta}_{bot}$ that are compatible with S_{bot} such that:

$$< S_{bot}, \boldsymbol{\theta}_{bot} >= \arg\max_{H} p(\mathcal{E}|H)$$

6.3.2 Priored model

Similarly to the unpriored model, we can argue that the most specific clause sequence is able to induce any probability distribution that a subsequence can induce. Hence, after parameter learning we have a clause sequence with a probabilistic annotation that is optimized for the learning task.

Chapter 7

PILP Evaluation

7.1 Synthetic data experiment - Ground Kernel

To assess scalability, we measure the parameter learning error and run time for a very basic example. The example consists of a single clause and a set of facts. The body of the clause contains all of the literals that are ascertained by the facts. The head of the clause is a unique literal. As an example, 4 body literals result in a theory of:

```
h :- p1, p2, p3, p4.
p1.
p2.
p3.
p4.
```

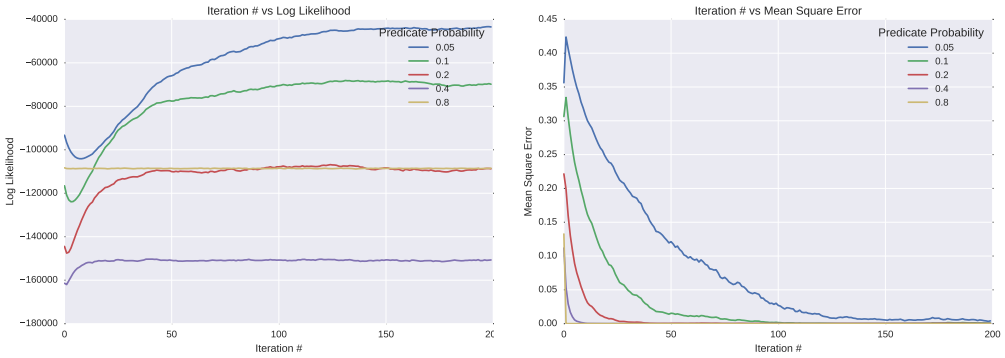
We associate a probability with each literal in the theory and generate hypotheses with accordance to the probabilistic model. This means that we generate each fact independently with the supplied probabilities. Then, we consider the head probability of the clause and start generating the clause with that probability. If we decide to generate the clause, we generate each body predicate independently with the provided probability. Then, we deduce the unique Herbrand interpretation of the hypothesis. In this case, the interpretation contains all of the generated facts. In addition, it contains the head of the clause iff every body predicate has been generated together with the corresponding fact. We then plot the results for varying probabilities, number of BDD Samples and number of interpretation samples, we also consider uncollapsed and collapsed Gibbs sampling.

7.1.1 EMBLEM

We measure the runtime for increasingly larger sizes of the program. The results are shown in Figure 7.1. We run this program with 10 random

Number of Facts	EMBLEM runtime	Full runtime	MSE	Score
1	0.95	4.61	0.0035	-1.68
3	3.17	15.49	0.0630	-3.01
4	5.74	29.23	0.119	-3.70
5	17.37	58.30	0.09	-4.66
6	22.58	58.09	0.09	-5.56

FIGURE 7.1: EMBLEM runtime and score



(A) Log Likelihood (B) Mean Square Error

FIGURE 7.2: Convergence rate of parameter learning for a fixed program with varying probability distribution

restarts, $\epsilon = 0.00001$[1] and an initial state of $\theta_j^i = 0.5$ for all literal indices. Then we compare probabilistic inference execution time, execution time of logic and probabilistic inference, mean square error and score (which is based on log likelihood). The error of learned θ values increases quickly and in a hypothesis with 5 facts it is already the case that the EM computation gets stuck in the initial state. The method is overall not particularly scalable.

7.1.2 Peircebayes

Log Likelihood and Mean Square Error against probabilities

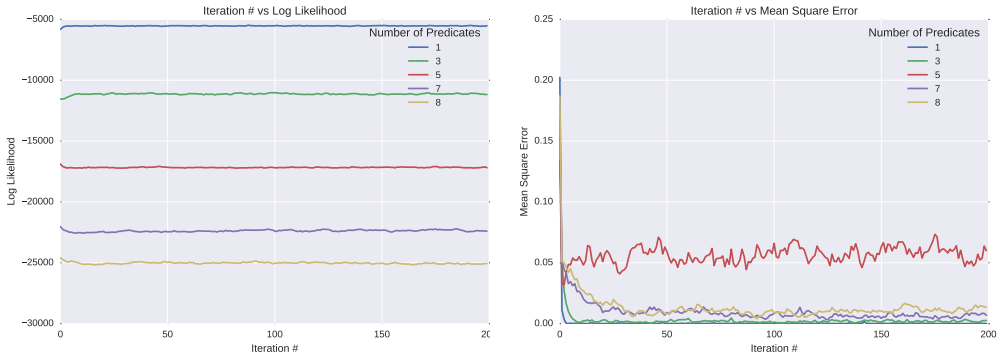
For the first experiment, we create a Kernel Set with 1 fact. This means that we need to specify three probabilities: the probability of the head and the body literal of the clause, and the probability of the fact. Using uncollapsed Gibbs sampling, 100000 observation samples from the kernel set and 200 BDD samples, we choose a value and set all that probabilities to that given value. For this experiment, we consider 0.05, 0.1, 0.2, 0.4 and 0.8 as literal probabilities, respectively. The Log Likelihood and Mean Root Error of the parameters are depicted in Figure 7.2a and 7.2b, respectively.

A conclusion we can draw from this is that the choice of probabilities affects the number of BDD samples necessary for convergence. In this situation, lowering the probabilities results in certain observations becoming highly unlikely. Accurately learning hypothesis distributions from unlikely observations requires a larger number of samples.

Log Likelihood and Mean Square Error against number of literals

In this experiment, we consider a kernel set in which all the probabilities have a value of 0.8. Using uncollapsed Gibbs sampling, 100 BDD samples and 100000 observation samples from the kernel set, we attempt varying the number of facts in the kernel set. For every added fact, it is necessary to learn an extra pair of probabilities, the probability of the fact itself and

[1] δ is a parameter used in the termination condition



(A) Log Likelihood (B) Mean Square Error

FIGURE 7.3: Convergence rate of parameter learning for a program with increasing number of literals

Number of Predicates	Avg Runtime (secs)
1	1.74
3	3.89
5	9.79
7	38.022
8	81.35

FIGURE 7.4: Average runtime vs number of predicates

the probability of the predicate contributing to the clause as a body literal. The log Likelihood and the mean square error of the parameters is shown in Figure 7.3a and Figure 7.3b, respectively.

From these figures, we can draw the conclusion that larger programs generally take slightly longer to converge (for these probabilities). It is curious to notice that the program with 5 facts does not have parameter error converging to 0 like the others.

Note that the runtime (shown in Figure 7.4) increases exponentially with the number of predicates. The main reason for this is the fact that the number of possible observations doubles for every added head fact (every clause in the general case) and the majority of the runtime is consumed by logical abduction for this input size.

Overall, when compared to EMBLEM parameter learning, Peircebayes parameter learning results in faster computation and more accurate paramaters.

7.2 Synthetic data - Ambiguous dataset

In this section, we consider learning a program (along with a mode declaration that allows PROBXHAIL to learn this structure):

```
a:0.8 :- b:0.9.
b:0.7 :- a:0.6.
```

Note that there are several deterministic answer set programs which produce the exact same answer sets $\{a, b\}$ that can be generated from it: $H_1 = \{a \leftarrow b; b \leftarrow a\}$, $H_2 = \{a \leftarrow b; b\}$, $H_3 = \{a; b \leftarrow a\}$ and $H_4 = \{a; b\}$. We can observe several possible hypothesis for $\emptyset$. In fact, if we attempt to analytically calculate the probability of each predicate from the observation probabilities given the hypothesis, we will find that the system of equations has an infinite number of solutions.

7.2.1 EMBLEM

EMBLEM doesn't converge to a specific value but converges to a different set of parameters every time. The parameters always induce a probability distribution on the observations that is close to the expected value. This is expected of EM, which will converge to a random (potentially local) maximum, depending on the initial state of the algorithm.

7.2.2 Peircebayes

Peircebayes isn't able to learn this probability distribution exactly, outputing the following hypothesis for a 1.0 symmetric Dirichlet Prior, 10000 BDD Samples and 10000000 kernel set samples:

```
a:0.9 :- b:0.8.
b:0.87 :- a:0.35.
```

Peircebayes calculates the expectation of the posterior value of θ, a value that can be uniquely inferred from the observation distribution, which means that the result doesn't change with different executions of the program.

Chapter 8

Related Work

PILP introduces the concept of rule learning in a probabilistic setting. In this chapter, we consider other PILP algorithms and how they relate to PROBXHAIL. The choice of languages in this paper is largely based on previous overviews of Probabilistic Logic Programming[31][9].
A common pattern in the following approaches is the iteration of structure and parameter learning, whereby one continuously repeats fixing the structure of the program and optimizes the probabilistic parameters to match the example set, followed by a choice of structure change. This is where PROBXHAIL differs from other languages, as parameter learning is completely separated procedure that is executed after structure learning is completed.

8.1 ProbFOIL and ProbFOIL+

ProbFOIL[11] and ProbFOIL+[12] are PILP algorithms that attempt to learn ProbLog programs.

8.1.1 PILP task

In the PILP task for ProbFOIL and ProbFOIL+, each example is a ground literal labelled with a probability. The goal is to find a ProbLog hypothesis which induces a probability distribution on the examples that is as close as possible to the target distribution (as described by the example probabilities). This is formalized by the concept of a loss function which is used to quantify the discrepancy between the hypothesis distribution and the target one. For ProbFOIL and ProbFOIL+, the loss function chosen is the sum of the absolute differences between query and label probability of each example.

8.1.2 Algorithm

ProbFOIL and ProbFOIL+ are a generalization FOIL[28], a deterministic ILP algorithm for Prolog programs that performs a top-down heuristic greedy search on a refinement tree similar to Figure 2.2.

ProbFOIL is initialized with the target predicate in the head and it greedily adds body literals and clauses based on a monotonic refinement operator. The progress of the algorithm is ensured by the fact that each clause added to the hypothesis increases the "number of true positives", a notion generalized from FOIL: Hypotheses can cover a fraction of a probabilistic example.

Note that ProbFOIL only learns deterministic rules. ProbFOIL+[12] improves on ProbFOIL by learning probabilistic rules. It is able to learn the probabilities of the candidate clauses by assuming the hypothesis consists of the given clause, and then varies the clauses parameter in order to optimize a local scoring function. After each possible refinement of the clause is considered, the clause with the highest possible scoring function is chosen. In addition, the ProbFOIL+ algorithm uses beam search rather than the simplified greedy search approach of ProbFOIL.

8.2 SLIPCASE and SLIPCOVER

SLIPCASE[2] and SLIPCOVER[3] are a family of PILP algorithms for LPADs.

8.2.1 PILP task

SLIPCASE and SLIPCOVER use an EM algorithm to do parameter learning. Similarly to the unpriored PROBXHAIL learning algorithm, the task is to choose a compact hypothesis for which the likelihood of the data given the hypothesis is as high as possible.

8.2.2 Algorithm

SLIPCASE learns LPADs and uses a beam search on a refinement operator to find the optimal hypothesis. For each candidate hypothesis, EMBLEM[1] is used to calculate the parameters of the theory. Then, the beam search is restricted based on the log likelihood of the hypotheses. SLIPCOVER is an efficiency improvement on SLIPCASE since it uses Muggleton's inverse entailment[25] to guide the beam search. SLIPCOVER uses the bottom clause as the initial hypothesis and then attempts to generalize it.
Unlike SLIPCOVER, PROBXHAIL doesn't use a most-specific clause to bind the search space, the most-specific clause itself is the basis of the algorithm. In this respect, PROBXHAIL is a less scalable algorithm since it produces a larger logic program.

8.3 Sem CP-Logic

SEM CP logic[24] converts CP theories (which are very similar to LPADs) to Bayesian Networks and then uses Bayesian Network Learning techniques to learn new hypotheses. The Bayesian Network consists of atom variables and clause variables. Atom variables represent an assignment and clause variables represent a probability distribution of the heads of the given rule. Parameter learning is done by instantiating the atoms in the Bayesian network to the values of the example and then using the Bayesian Network to infer the head probabilities of rules. Then, structure learning can be done by adding variables to the Bayesian Network using a refinement operator in a way that ensures that the resulting Bayesian Network represents a valid CP-Theory. Parameter learning is done on the resulting theories. Then, theories with high log likelihood of the data are considered.
Treating the probabilistic rule set as a Bayesian network inspires an extension to PROBXHAIL. The probabilistic language is currently the most basic

dependency model between the literals. Introducing an extra literal in the network can be beneficial.

Chapter 9

Conclusion and Future Work

9.1 Achievements

Overall, the outcomes of this project include:

- The implementation of answer set program input in Peircebayes
- The implementation of answer set programs for several statistical abduction tasks which are faster than their Prolog counterparts
- The definition of annotated literal programs, a new type of probabilistic logic program that is arguably more expressive than some modern PLP languages, due to it utilizing both Beta priors and stable set semantics
- The definition and implementation of several parameter learning tasks for annotated literal programs: a Peircebayes statistical abduction task and an LPAD parameter learning task
- The development of PROBXHAIL, a structural learning algorithm for annotated literal programs and a probabilistic generalization of the XHAIL algorithm
- The introduction of inductive-abductive learning in the context of PILP

9.2 Potential Extensions

9.2.1 Data-driven most specific hypothesis

One of the main issues of PROBXHAIL is its low scalability. For this reason, it is important to keep the most specific hypothesis as small as possible. Doing that in a bias-driven approach to structure learning requires very specific mode declarations and, hence, a lot of user interaction. A significant improvement of this algorithm would be to use the data to generate the most specific hypothesis in a manner similar to algorithms based on Inverse Entailment.
XHAIL already features a modification of Kernel Set Subsumption (as defined in HAIL) to perform data-driven computation of a most-specific hypothesis. However, XHAIL is restricted to learning data from a single interpretation, whereas the PILP tasks defined in this report require the construction of a probabilistic hypothesis which is able to explain several potentially mutually inconsistent interpretations.

In order to tackle this problem, we propose a different data-driven approach that generalizes the structural learning procedure in XHAIL:

1. For each interpretation construct the ground kernel set as in XHAIL, with one major difference: whenever there are multiple solutions to the abductive stage of XHAIL, construct a separate kernel set from every possible set of heads, rather than choosing an arbitrary solution to the abductive task and discarding others.

2. "Merge" the ground kernel sets: group all clauses in the kernel sets by their ground head and merge each group in a single clause by preserving the head and taking the union of all body literals.

3. Lift the kernel and trim the kernel set according to the bias, as described in Chapter 6.

This approach should result in a probabilistic hypothesis that is able to explain every interpretation, while remaining more compact than the most specific hypothesis generated from a broad bias. Whether this algorithm targets the PILP task defined in this report is an open question.

9.2.2 Probabilistic model extension

A potential way to produce more accurate programs is to modify the probabilistic model. Techniques borrowed from Bayesian Network structural learning could be used to construct a more realistic probabilistic model[16].

An example of this would be to remove the independency assumption between certain body literals. Currently, body literals in each clause of an annotated literal program are considered independently for inclusion. However, certain body literals may occur much more often together than they would separately. For example, the predicate $teaches$ might almost never occur without the predicate $teacher$ in the same clause of a hypothesis, as the concept of teaching is associated with a teacher. In this situation, using an independent inclusion variable per literal in the probabilistic model is inadequate, as the inclusion of one literal depends on the inclusion of the other.
The way to implement this is to generalize the PLP model to associate groups of body literals with a single categorical variable that differentiates between all joint states.

Example: Parameter learning for such a program can be done using Peircebayes. We associate a group of N variables with a single `ucl` predicate that can have 2^N values. So the clause:

$$advisedby(stan, T) \leftarrow teacher(T), teaches(T, C), level(C, graduate)$$

can be rewritten in the abductive program as (assuming a model with a joint random variable for the literals $teacher(T)$ and $teaches(T, C)$):

```
advisedby(stan,T) :- ucl(0,0), try_multi(0,1,T,C), try(2,0,C).

try_multi(0,1,T,C) :- not teacher(T), not teaches(T,C),
```

```
        ucl_multi(0,1,none).
try_multi(0,1,T,C) :- teacher(T), not teaches(T,C),
        ucl_multi(0,1,onlyfirst).
try_multi(0,1,T,C) :- not teacher(T), teaches(T,C),
        ucl_multi(0,1,onlysecond).
try_multi(0,1,T,C) :- teacher(T), teaches(T,C),
        ucl_multi(0,1,both).

try(0,2,C) :- not ucl(0,2).
try(0,2,C) :- level(C, graduate), ucl(0,2).
```

Similarly, in EMBLEM we can have an LPAD that learns a distribution on the joint states of the literals:

```
ucl_multi(0,1,both):0.25, ucl_multi(0,1,onlyfirst):0.25,
        ucl_multi(0,1,onlysecond):0.25.
```

Note that the joint exclusion of literals is modelled implicitly in the LPAD by the empty head probability.

A different change to the probability distribution is to attempt omitting body literals in the most-specific hypothesis. This could potentially increase the accuracy of an overfit model. This can be done randomly, or, similarly to SLIPCOVER, with respect to a scoring function.

An additional topic for research would be to investigate the use of a noise variable in the probabilistic model. The current PLP language is not able to handle inconsistencies within the interpretations themselves, even though it is able to handle uncertainty of the occurrence of a specific interpretation.

9.2.3 Potential applications

Another important task to research is to evaluate the practical value of this algorithm. This algorithm has only been evaluated on synthetic datasets that aim to show that PROBXHAIL correctly optimizes the parameters to complete the learning task. We've made an unsuccessful attempt on getting the algorithm to work on a large dataset.

Using Beta priors is a mathematically well-defined way of incorporating prior knowledge in a model. It can be considered a more elegant approach of initializing a model than using an initial state in an EM algorithm. In EM, the initial parameters determine whether the MLE computation would get stuck in a local maximum or not. However, we haven't investigated how practically useful it is to incorporate the Beta priors in the bias. Perhaps it would be meaningful to annotate each mode declaration with the number of times the literal is expected to appear in a hypothesis and the number of times it wouldn't.

9.3 Concluding Remarks

As large relational datasets become more prevalent, extracting patterns from data in the form of human comprehensible rule sets becomes increasingly valuable. Introducing uncertainty in logic-based models extends the their usefulness in domains for which modelling error and randomness is a core concept, such as computational biology. This project is novel in its attempt to introduce abductive-inductive learning in the context of PILP. In addition, it introduces a set of expressive probabilistic logic programming languages that uses both stable set semantics and a prior distribution for parameter learning. The most apparent issue that impacts the practical usability of the model is its scalability. Indeed, combining logic-based learning and statistical distributions in the same model introduces two sources of intractable problems. Logic satisfiability and the evaluation of probabilistic models with intractably large latent spaces are both problems that need to be solved in probabilistic logic programs. For this reason, improving scalability and parallelism in such systems could produce machine learning models of great value.

Appendix A

Peircebayes task implementations

A.1 RIM

A.1.1 Prolog implementation

This Prolog implementation was originally defined in [38]

```
% prob distribs
pb_dirichlet(8.33333333333, pi, 6, 1).
pb_dirichlet(0.1, p2, 2, 6).
pb_dirichlet(0.1, p3, 3, 6).
pb_dirichlet(0.1, p4, 4, 6).
pb_dirichlet(0.1, p5, 5, 6).
pb_dirichlet(0.1, p6, 6, 6).
pb_dirichlet(0.1, p7, 7, 6).
pb_dirichlet(0.1, p8, 8, 6).
pb_dirichlet(0.1, p9, 9, 6).
pb_dirichlet(0.1, p10, 10, 6).

% plate
pb_plate(
    [observe(Sample)],
    1,
    [generate([0,1,2,3,4,5,6,7,8,9], Sample)]
).

insert_rim([], ToIns, Ins,
    Pos, Ins1) :-
    append(Ins, [ToIns], Ins1),
    length(Ins1, Pos).
insert_rim([H|_T], ToIns, Ins,
    Pos, Ins1) :-
    nth1(Pos, Ins, H),
    nth1(Pos, Ins1, ToIns, Ins).
insert_rim([H|T] , ToIns, Ins,
    Pos, Ins1) :-
    \+member(H, Ins),
    insert_rim(T, ToIns, Ins,
        Pos, Ins1).

generate([H|T], Sample):-
    K in 1..6,
    pi(K,1),
    generate(T, Sample, [H], 2, K).

generate([], Sample, Sample, _Idx, _K).
generate([ToIns|T], Sample, Ins, Idx, K) :-
```

```
    % insert next element at Pos
    % yielding a new list Ins1
    append(_, [ToIns|Rest], Sample),
    insert_rim(Rest, ToIns, Ins,
        Pos, Ins1),
    % build prob predicate in Pred
    number_chars(Idx, LIdx),
    append(['p'], LIdx, LF),
    atom_chars(F, LF),
    Pred =.. [F, Pos, K],
    % call prob predicate
    pb_call(Pred),
    Idx1 is Idx+1,
    % recurse
    generate(T, Sample, Ins1, Idx1, K).
```

A.1.2 ASP implementation

Outer Query Grounding

```
% NUMIDS is the number of possible examples.
1 { outer_query(1..NUMIDS) } 1.

% An example observation:
% Ranking consists of several steps of inserting elements into
% the model. Each ranking(Step, Item, Position) atom signifies
% that that in step Step, item Item is in position Position.
10 { ranking(9, 5, 1),
     ranking(9, 0, 2),
     ranking(9, 3, 3),
     ranking(9, 4, 4),
     ranking(9, 6, 5),
     ranking(9, 9, 6),
     ranking(9, 8, 7),
     ranking(9, 1, 8),
     ranking(9, 7, 9),
     ranking(9, 2, 10)  } 10 :- outer_query(1).
```

Abductive Task

```
% Type information.
#hide step/1.
#hide elem/1.
#hide pos/1.
step(0..9).
elem(0..9).
pos(1..10).

% Each pi represents a choice of a different ranking function.
1 {pi(1..6)} 1.
#hide ranking/3.
%#hide outer_query/1.

% Generates previous step of RIM by "removing" an element
% out of its position.
ranking(I, Elem, Pos) :-
  step(I), elem(Elem), pos(Pos),
  ranking(I+1, I+1, PosNextElem),
  ranking(I+1, Elem, Pos),
```

```
  PosNextElem > Pos.

ranking(I, Elem, Pos) :-
  step(I), elem(Elem), pos(Pos),
  ranking(I+1, I+1, PosNextElem),
  ranking(I+1, Elem, Pos+1),
  PosNextElem < Pos + 1.

% At step i we use pi to choose a new position to
% insert the new element.
p2(Pos, K) :- pi(K), ranking(1, 1, Pos).
p3(Pos, K) :- pi(K), ranking(2, 2, Pos).
p4(Pos, K) :- pi(K), ranking(3, 3, Pos).
p5(Pos, K) :- pi(K), ranking(4, 4, Pos).
p6(Pos, K) :- pi(K), ranking(5, 5, Pos).
p7(Pos, K) :- pi(K), ranking(6, 6, Pos).
p8(Pos, K) :- pi(K), ranking(7, 7, Pos).
p9(Pos, K) :- pi(K), ranking(8, 8, Pos).
p10(Pos, K) :- pi(K), ranking(9, 9, Pos).
```

A.2 Seeded LDA

A.2.1 Prolog

This Prolog implementation was originally defined in [38]

```
seed_naf(Token) :- seed(Token, _).

seed(9503, [1]).
seed(13296, [1]).

% prob distribs
pb_dirichlet(2.5, theta, 20, 4767).
pb_dirichlet(0.01, phi, 27485, 20).

% plate
pb_plate(
    [observe(d(Doc), TokenList),
     member((w(Token), Count), TokenList),
     \+ seed_naf(Token)],
    Count,
    [Topic in 1..20,
     theta(Topic,Doc),
     phi(Token,Topic)]
).

pb_plate(
    [observe(d(Doc), TokenList),
     member((w(Token), Count), TokenList),
     seed_naf(Token)],
    Count,
    [seed(Token, TopicList),
     member(Topic, TopicList),
     theta(Topic,Doc),
     phi(Token,Topic)]
).

%example observations
observe(d(1), [(w(20791),1),(w(4045),1),(w(20022),1)].
```

A.2.2 ASP representation

This model consists of 2 queries, which we consider as 2 separate abductive tasks.

Outer Queries

First query:

```
#hide.
#show outer_query/3.

1 { outer_query(Doc, Token, Count) :
                observe(d(Doc), (w(Token), Count)) } 1.

:- outer_query(_, Token, _), seed_naf(Token).


count(Count) :- outer_query(_, _, Count).

%example observation
observe(d(1), (w(20601),1)).

%example lexical prior
seed_naf(Token) :- seed(Token, _).
seed(9399, 1).
seed(13168, 1).
```

Second query:

```
#hide.
#show outer_query/3.

1 { outer_query(Doc, Token, Count)
                : outer_q(Doc, Token, Count)} 1.

outer_q(Doc, Token, Count) :-
        observe(d(Doc), (w(Token), Count)), seed_naf(Token).

count(Count) :- outer_query(_, _, Count).

%example observation
observe(d(1), (w(20601),1)).

%example lexical prior
seed_naf(Token) :- seed(Token, _).
seed(9399, 1).
seed(13168, 1).
```

Inner Queries

First query:

```
#show theta/2.
#show phi/2.

inner_query(Doc, Token, Count) :- outer_query(Doc, Token, Count).

1{ topic(1..20) } 1 :- inner_query(_, _, _).
```

```
1{ theta(Topic, Doc) } 1 :- topic(Topic),
        inner_query(Doc, Token, Count).
1{ phi(Token, Topic) } 1 :- topic(Topic),
        theta(Topic, Doc),
        inner_query(Doc, Token, _).
```

Second query:

```
#show theta/2.
#show phi/2.

inner_query(Doc, Token, Count) :- outer_query(Doc, Token, Count).

1{ theta(Topic, Doc) } 1 :-
        seed(Token, Topic),
        inner_query(Doc, Token, _).
1{ phi(Token, Topic) } 1 :-
        seed(Token, Topic),
        theta(Topic, Doc),
        inner_query(Doc, Token, _).
```

A.3 Fast LDA

This describes a manual implementation of LDA that is faster than the original definitions in Chapter 4.

A.3.1 Outer Query

```
#hide.
#show outer_query/3.
1 { outer_query(Doc, Token, Count)
        : observe(d(Doc), (w(Token), Count)) } 1.

count(Count) :- outer_query(_, _, Count).

%example observation
observe(d(1), (w(3), 3)).
```

A.3.2 Inner Query

```
#show theta/2.
#show phi/2.

inner_query(Doc, Token, Count) :- outer_query(Doc, Token, Count).

1{ topic(1..10) } 1 :- inner_query(_, _, _).
1{ theta(Topic, Doc) } 1 :-
        topic(Topic),
        inner_query(Doc, Token, Count).
1{ phi(Token, Topic) } 1 :-
        topic(Topic),
        theta(Topic, Doc),
        inner_query(Doc, Token, _).
```

Bibliography

[1] Elena Bellodi and Fabrizio Riguzzi. "Expectation Maximization over binary decision diagrams for probabilistic logic programs". In: *Intelligent Data Analysis* 17.2 (2013), pp. 343–363.

[2] Elena Bellodi and Fabrizio Riguzzi. "Inductive Logic Programming: 21st International Conference, ILP 2011, Windsor Great Park, UK, July 31 – August 3, 2011, Revised Selected Papers". In: ed. by Stephen H. Muggleton, Alireza Tamaddoni-Nezhad, and Francesca A. Lisi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. Chap. Learning the Structure of Probabilistic Logic Programs, pp. 61–75. ISBN: 978-3-642-31951-8. DOI: 10.1007/978-3-642-31951-8_10. URL: http://dx.doi.org/10.1007/978-3-642-31951-8_10.

[3] Elena Bellodi and Fabrizio Riguzzi. "Structure learning of probabilistic logic programs by searching the clause space". In: *Theory and Practice of Logic Programming* 15 (Special Issue 02 Mar. 2015), pp. 169–212. ISSN: 1475-3081. DOI: 10.1017/S1471068413000689. URL: http://journals.cambridge.org/article_S1471068413000689.

[4] Christopher M Bishop. "Pattern Recognition and Machine Learning". In: (2006).

[5] David M Blei, Andrew Y Ng, and Michael I Jordan. "Latent dirichlet allocation". In: *the Journal of machine Learning research* 3 (2003), pp. 993–1022.

[6] Ivan Bratko. *Refining complete hypotheses in ILP*. Springer, 1999.

[7] Jenny Brynjarsdottir. *Lecture slides on Estimation*. URL: https://www2.stat.duke.edu/courses/Fall12/sta611/Lecture12.pdf (visited on 01/29/2016).

[8] Yihua Chen and Maya R. Gupta. *EM Demystified: An Expectation-Maximization Tutorial*. URL: https://www.ee.washington.edu/techsite/papers/documents/UWEETR-2010-0002.pdf (visited on 01/29/2016).

[9] Luc De Raedt and Kristian Kersting. "Probabilistic logic learning". In: *SIGKDD Explorations: newsletter of the Special Interest Group (SIG) on Knowledge Discovery & Data Mining* 5.1 (2003), pp. 31–48.

[10] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. "ProbLog: A Probabilistic Prolog and Its Application in Link Discovery". In: *Proceedings of the 20th International Joint Conference on Artifical Intelligence*. IJCAI'07. Hyderabad, India: Morgan Kaufmann Publishers Inc., 2007, pp. 2468–2473. URL: http://dl.acm.org/citation.cfm?id=1625275.1625673.

[11] Luc De Raedt and Ingo Thon. "Probabilistic rule learning". In: *Inductive Logic Programming*. Springer, 2011, pp. 47–58.

[12] Luc De Raedt et al. “Inducing probabilistic relational rules from probabilistic examples”. In: *International Joint Conference on Artificial Intelligence (IJCAI)*. 2015.

[13] Jean-Paul Doignon, Aleksandar Pekeč, and Michel Regenwetter. “The repeated insertion model for rankings: Missing link between two subset choice models”. In: *Psychometrika* 69.1 (2004), pp. 33–54.

[14] Martin Gebser et al. “A user’s guide to gringo, clasp, clingo, and iclingo”. In: (Oct. 4, 2010). URL: https://www.cs.utexas.edu/users/vl/teaching/lbai/clingo_guide.pdf (visited on 05/29/2016).

[15] Duncan Gilies. *Lecture notes on Bayes Theorem and Bayesian Inference*. URL: https://www.doc.ic.ac.uk/~dfg/ProbabilisticInference/IDAPILecture01.pdf (visited on 01/29/2016).

[16] Duncan Gilies. *Lecture slides on Approximate Inference*. URL: https://www.doc.ic.ac.uk/~dfg/ProbabilisticInference/IDAPISlides08.pdf (visited on 01/29/2016).

[17] Leon Gu. *Lecture Slides on Dirichlet Distribution, Dirichlet Process and Dirichlet Process Mixture*. URL: http://www.cs.cmu.edu/~epxing/Class/10701-08s/recitation/dirichlet.pdf (visited on 01/29/2016).

[18] Masakazu Ishihata and Taisuke Sato. “Bayesian inference for statistical abduction using Markov chain Monte Carlo”. In: *ACML*. 2011, pp. 81–96.

[19] Jagadeesh Jagarlamudi, Hal Daumé III, and Raghavendra Udupa. “Incorporating lexical priors into topic models”. In: *Proceedings of the 13th Conference of the European Chapter of the Association for Computational Linguistics*. Association for Computational Linguistics. 2012, pp. 204–213.

[20] T. Kamishima, H. Kazawa, and S. Akaho. “Supervised ordering - an empirical survey”. In: *Fifth IEEE International Conference on Data Mining (ICDM’05)*. 2005, 4 pp. DOI: 10.1109/ICDM.2005.138.

[21] Ken Lang. “NewsWeeder: Learning to Filter Netnews”. In: *in Proceedings of the 12th International Machine Learning Conference (ML95*. 1995.

[22] Mark Law. *Lecture notes in Answer Set Programming*. URL: https://www.doc.ic.ac.uk/~ml1909/teaching/Unit7.pdf (visited on 06/05/2016).

[23] Mark Law, Alessandra Russo, and Krysia Broda. “Inductive learning of answer set programs”. In: *Logics in Artificial Intelligence*. Springer, 2014, pp. 311–325.

[24] Wannes Meert, Jan Struyf, and Hendrik Blockeel. “Learning ground CP-Logic theories by leveraging Bayesian network learning techniques”. In: *Fundamenta Informaticae* 89.1 (2008), pp. 131–160.

[25] Stephen Muggleton. “Inverse entailment and Progol”. In: *New Generation Computing* 13.3 (), pp. 245–286. DOI: 10.1007/BF03037227. URL: http://dx.doi.org/10.1007/BF03037227.

[26] "Nonmonotonic abductive inductive learning". In: *Journal of Applied Logic* 7.3 (2009), pp. 329–340. URL: http://www.sciencedirect.com/science/article/pii/S1570868308000682.

[27] Frank Pfenning. *Lecture notes on Binary Decision Diagrams*. Oct. 28, 2010. URL: https://www.cs.cmu.edu/~fp/courses/15122-f10/lectures/19-bdds.pdf (visited on 01/25/2016).

[28] J.R. Quinlan. "Learning Logical Definitions from Relations". In: *Machine Learning* 5.3 (1990), pp. 239–266. ISSN: 1573-0565. DOI: 10.1023/A:1022699322624. URL: http://dx.doi.org/10.1023/A:1022699322624.

[29] Oliver Ray, Krysia Broda, and Alessandra Russo. "Hybrid abductive inductive learning: a generalisation of Progol". In: *Inductive Logic Programming*. Springer, 2003, pp. 311–328.

[30] Fabrizio Riguzzi. "The Distribution Semantics Is Well-Defined for All Normal Programs". In: *PLP 2015 Probabilistic Logic Programming* (), p. 69.

[31] Fabrizio Riguzzi, Elena Bellodi, and Riccardo Zese. "A History of Probabilistic Inductive Logic Programming". In: *Frontiers in Robotics and AI* 1 (2014), p. 6.

[32] Fabrizio Riguzzi and Terrance Swift. "Probabilistic Logic Programming Under the Distribution Semantics". In: ().

[33] Alessandra Russo. *Lecture notes on Inductive Logic Programming*. 2015.

[34] Taisuke Sato. "A Statistical Learning Method for Logic Programs with Distribution Semantics". In: *Proceedings of the 12th International Conference on Logic Programming*. MIT Press, 1995, pp. 715–729.

[35] Marek Sergot. *Lecture notes on Minimal models and fixpoint semantics for definite logic program*. Jan. 2005. URL: http://www.doc.ic.ac.uk/~mjs/teaching/KnowledgeRep491/Fixpoint_Definite_491-2x1.pdf (visited on 01/29/2016).

[36] Marek Sergot. *Lecture notes on Stable Models (Answer Sets)*. Feb. 2006. URL: https://www.doc.ic.ac.uk/~mjs/teaching/KnowledgeRep491/StableModels-2x1.pdf (visited on 01/29/2016).

[37] Ehud Y Shapiro. *Inductive inference of theories from facts*. 1981.

[38] Calin Rares Turliuc et al. "Probabilistic Abductive Logic Programming using Dirichlet Priors". In: *PLP 2015 Probabilistic Logic Programming* (2015), p. 85.

[39] Joost Vennekens, Sofie Verbaeten, and Maurice Bruynooghe. "Logic Programs with Annotated Disjunctions". English. In: *Logic Programming*. Ed. by Bart Demoen and Vladimir Lifschitz. Vol. 3132. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, pp. 431–445. ISBN: 978-3-540-22671-0. DOI: 10.1007/978-3-540-27775-0_30. URL: http://dx.doi.org/10.1007/978-3-540-27775-0_30.