Probabilistic Inductive Logic Programming

Luc De Raedt and Kristian Kersting

Institute for Computer Science, Machine Learning Lab
Albert-Ludwigs-University, Georges-Köhler-Allee, Gebäude 079,
D-79110 Freiburg i. Brg., Germany
{deraedt,kersting}@informatik.uni-freiburg.de

Abstract. Probabilistic inductive logic programming, sometimes also called statistical relational learning, addresses one of the central questions of artificial intelligence: the integration of probabilistic reasoning with first order logic representations and machine learning. A rich variety of different formalisms and learning techniques have been developed. In the present paper, we start from inductive logic programming and sketch how it can be extended with probabilistic methods.
More precisely, we outline three classical settings for inductive logic programming, namely *learning from entailment*, *learning from interpretations*, and *learning from proofs or traces*, and show how they can be used to learn different types of probabilistic representations.

1 Introduction

In the past few years there has been a lot of work lying at the intersection of probability theory, logic programming and machine learning [40, 15, 42, 31, 35, 18, 25, 21, 2, 24]. This work is known under the names of statistical relational learning [14, 11], probabilistic logic learning [9], or probabilistic inductive logic programming. Whereas most of the existing works have started from a probabilistic learning perspective and extended probabilistic formalisms with relational aspects, we will take a different perspective, in which we will start from inductive logic programming and study how inductive logic programming formalisms, settings and techniques can be extended to deal with probabilistic issues. This tradition has already contributed a rich variety of valuable formalisms and techniques, including probabilistic Horn abduction by David Poole, PRISMs by Sato, stochastic logic programs by Eisele [12], Muggleton [31] and Cussens [4], Bayesian logic programs [22, 20] by Kersting and De Raedt, and Logical Hidden Markov Models [24].

The main contribution of this paper is the introduction of three probabilistic inductive logic programming settings which are derived from the learning from entailment, from interpretations and from proofs settings of the field of inductive logic programming [6]. Each of these settings contributes different notions of probabilistic logic representations, examples and probability distributions. The first setting, probabilistic learning from entailment, combines key principles of the well-known inductive logic programming system FOIL [41] with the naïve Bayes' assumption; the second setting, probabilistic learning from interpretations, incorporated in Bayesian logic programs [22, 20], integrates Bayesian networks with logic programming; and the third setting, learning from proofs, incorporated in stochastic logic programs [12, 31, 4], upgrades stochastic context free grammars to logic programs. The sketched settings (and their instances presented) are by no means the only possible settings for probabilistic inductive logic programming. Nevertheless, two of the settings have – to the authors' knowledge – not been introduced before. Even though it is not our aim to provide a complete survey on probabilistic inductive logic programming (for such a survey, see [9]), we hope that the settings will contribute to a better understanding of probabilistic extensions to inductive logic programming and will also clarify some of the logical issues about probabilistic learning.

This paper is structured as follows: in Section 2, we present the three inductive logic programming settings, in Section 3, we extend these in a probabilistic framework, in Section 4, we discuss how to learn probabilistic logics in these three settings, and finally, in Section 5, we conclude. The Appendix contains a short introduction to some logic programming concepts and terminology that are used in this paper.

2 Inductive Logic Programming Settings

Inductive logic programming is concerned with finding a hypothesis H (a logic program, i.e. a definite clause program) from a set of positive and negative examples P and N. More specifically, it is required that the hypothesis H *covers* all positive examples in P and none of the negative examples in N. The representation language chosen for representing the examples together with the *covers* relation determines the inductive logic programming setting [6]. Various settings have been considered in the literature, most notably learning from *entailment* [39] and learning from *interpretations* [8, 17], which we formalize below. We also introduce an intermediate setting inspired on the seminal work by Ehud Shapiro [43], which we call learning from *proofs*.

Before formalizing these settings, let us however also discuss how background knowledge is employed within inductive logic programming. For the purposes of the present paper[1], it will be convenient to view the background knowledge B as a logic program (i.e. a definite clause program) that is provided to the inductive logic programming system and fixed during the learning process. Under the presence of background knowledge, the hypothesis H together with the background theory B should cover all positive and none of the negative examples. The ability to provide declarative background knowledge to the learning engine is viewed as one of the strengths of inductive logic programming.

[1] For the learning from interpretations setting, we slightly deviate from the standard definition in the literature for didactic purposes.

2.1 Learning from Entailment

Learning from entailment is by far the most popular inductive logic programming systems and it is addressed by a wide variety of well-known inductive logic programming systems such as FOIL [41], Progol [30], and Aleph [44].

Definition 1. *When learning from entailment, the examples are definite clauses and a hypothesis H covers an example e w.r.t. the background theory B if and only if $B \cup H \models e$.*

In many well-known systems, such as FOIL, one requires that the examples are ground facts, a special case of definite clauses. To illustrate the above setting, consider the following example inspired on the well-known mutagenicity application [45].

Example 1. Consider the following facts in the background theory B. They describe part of molecule 225.

```
molecule(225).                              bond(225,f1_1,f1_2,7).
logmutag(225,0.64).                         bond(225,f1_2,f1_3,7).
lumo(225,-1.785).                           bond(225,f1_3,f1_4,7).
logp(225,1.01).                             bond(225,f1_4,f1_5,7).
nitro(225,[f1_4,f1_8,f1_10,f1_9]).          bond(225,f1_5,f1_1,7).
atom(225,f1_1,c,21,0.187).                  bond(225,f1_8,f1_9,2).
atom(225,f1_2,c,21,-0.143).                 bond(225,f1_8,f1_10,2).
atom(225,f1_3,c,21,-0.143).                 bond(225,f1_1,f1_11,1).
atom(225,f1_4,c,21,-0.013).                 bond(225,f1_11,f1_12,2).
atom(225,f1_5,o,52,-0.043).                 bond(225,f1_11,f1_13,1).
...
ring_size_5(225,[f1_5,f1_1,f1_2,f1_3,f1_4]).
hetero_aromatic_5_ring(225,[f1_5,f1_1,f1_2,f1_3,f1_4]).
...
```

Consider now the example mutagenic(225). It is covered by the following clause

```
mutagenic(M) :- nitro(M,R1), logp(M,C), C > 1 .
```

Inductive logic programming systems that learn from entailment often employ a typical separate-and-conquer rule-learning strategy [13]. In an outer loop of the algorithm, they follow a set-covering approach [29] in which they repeatedly search for a rule covering many positive examples and none of the negative examples. They then delete the positive examples covered by the current clause and repeat this process until all positive examples have been covered. In the inner loop of the algorithm, they typically perform a general-to-specific heuristic search employing a refinement operator under θ-subsumption [39].

2.2 Learning from Interpretations

The learning from interpretations settings [8] has been inspired on the work on boolean concept-learning in computational learning theory [47].

Definition 2. *When learning from interpretations, the examples are Herbrand interpretations and a hypothesis H covers an example e w.r.t. the background theory B if and only if e is a model of $B \cup H$.*

Herbrand interpretations are sets of true ground facts and they completely describe a possible situation.

Example 2. Consider the interpretation I which is the union of B

```
{father(henry,bill),   father(alan,betsy),  father(alan,benny),
father(brian,bonnie), father(bill,carl),   father(benny,cecily),
father(carl,dennis),  mother(ann,bill),    mother(ann,betsy),
mother(ann,bonnie),   mother(alice,benny), mother(betsy,carl),
mother(bonnie,cecily), mother(cecily,dennis), founder(henry).
founder(alan). founder(an). founder(brian). founder(alice).
```

and $C = \{\texttt{carrier(alan)}, \texttt{carrier(ann)}, \texttt{carrier(betsy)}\}$. It is covered by the clause c

```
carrier(X) :- mother(M,X),carrier(M),father(F,X),carrier(F).
```

This clause covers the interpretation I because for all substitutions θ such that $body(c)\theta \subseteq I$ holds, it also holds that $head(c)\theta \in I$.

The key difference between learning from interpretations and learning from entailment is that interpretations carry much more – even complete – information. Indeed, when learning from entailment, an example can consist of a *single* fact, whereas when learning from interpretations, all facts that hold in the example are known. Therefore, learning from interpretations is typically easier and computationally more tractable than learning from entailment, cf. [6].

Systems that learn from interpretations work in a similar fashion as those that learn from entailment. There is however one crucial difference and it concerns the generality relationship. A hypothesis G *is more general than* a hypothesis S if all examples covered by S are also covered by G. When learning from entailment, G is more general than S if and only if $G \models S$, whereas when learning from interpretations, when $S \models G$. Another difference is that learning from interpretations is well suited for learning from positive examples only. For this case, a complete search of the space ordered by θ-subsumption is performed until all clauses cover all examples [7].

2.3 Learning from Proofs

Because learning from entailment (with ground facts as examples) and interpretations occupy extreme positions w.r.t. the information the examples carry, it is interesting to investigate intermediate positions. Ehud Shapiro's Model Inference System (MIS) [43] fits nicely within the learning from entailment setting where examples are facts. However, to deal with missing information, Shapiro employs a clever strategy: MIS queries the user for missing information by asking her for the truth-value of facts. The answers to these queries allow MIS to reconstruct the trace or the proof of the positive examples. Inspired by Shapiro, we define the learning from proofs setting.

Definition 3. *When learning from proofs, the examples are ground proof-trees and an example e is covered by a hypothesis H w.r.t. the background theory B if and only if e is a proof-tree for $H \cup B$.*

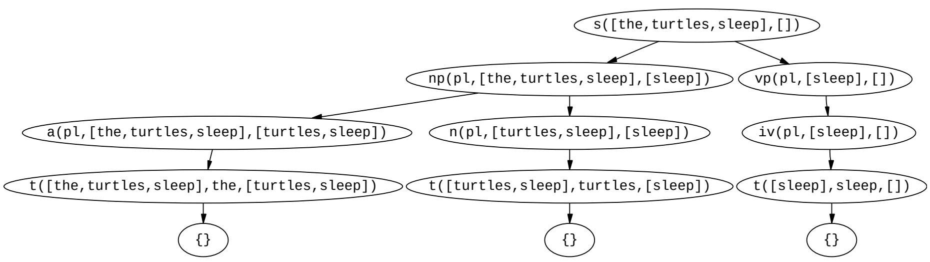
At this point, there exist various possible forms of proof-trees. In this paper, we will – for reasons that will become clear later – assume that the proof-tree is given in the form of an and-tree where the nodes contain ground atoms. More formally:

Definition 4. *t is a proof-tree for T if and only if t is a rooted tree where for every node $n \in t$ with children $child(n)$ satisfies the property that there exists a substitution θ and a clause c such that $n = head(c)\theta$ and $child(n) = body(c)\theta$.*

Example 3. Consider the following *definite clause grammar*.

```
sentence(A, B) :- noun_phrase(C, A, D), verb_phrase(C, D, B).
noun_phrase(A, B, C) :- article(A, B, D), noun(A, D, C).
verb_phrase(A, B, C) :- intransitive_verb(A, B, C).
article(singular, A, B) :- terminal(A, a, B).
article(singular, A, B) :- terminal(A, the, B).
article(plural, A, B)   :- terminal(A, the, B).
noun(singular, A, B) :- terminal(A, turtle, B).
noun(plural, A, B)   :- terminal(A, turtles, B).
intransitive_verb(singular, A, B) :- terminal(A, sleeps, B).
intransitive_verb(plural, A, B) :- terminal(A, sleep, B).
terminal([A|B],A,B).
```

It covers the following proof tree u (where abbreviated accordingly):



Proof-trees contain – as interpretations – a lot of information. Indeed, they contain instances of the clauses that were used in the proofs. Therefore, it may be hard for the user to provide this type of examples. Even though that is generally true, there exist specific situations for which this is feasible. Indeed, consider tree banks such as the UPenn Wall Street Journal corpus [27], which contain parse trees. These trees directly correspond to the proof-trees we talk about. Even though – to the best of the authors' knowledge (but see [43, 3] for inductive logic programming systems that learn from traces) – no inductive logic programming system has been developed to learn from proof-trees, it is not hard to imagine an outline for such an algorithm. Indeed, by analogy with the learning of tree-bank grammars, one could turn all the proof-trees (corresponding to positive examples) into a set of ground clauses which would constitute the initial theory. This theory can then be generalized by taking the least general generalization (under θ-subsumption) of pairwise clauses. Of course, care must be taken that the generalized theory does not cover negative examples.

3 Probabilistic Inductive Logic Programming Settings

Given the interest in probabilistic logic programs as a representation formalism and the different learning settings for inductive logic programming, the question arises as to whether one can extend the inductive logic programming settings with probabilistic representations.

When working with probabilistic logic programming representations, there are essentially two changes:

1. clauses are annotated with probability values, and
2. the covers relation becomes a probabilistic one.

We will use $\mathbf{P}$ to denote a probability distribution, e.g. $\mathbf{P}(x)$, and the normal letter P to denote a probability value, e.g. $P(x = v)$, where v is a state of x. Thus,

> *a probabilistic covers relation takes as arguments an example e, a hypothesis H and possibly the background theory B. It then returns a probability value between 0 and 1. So, $covers(e, H \cup B) = \mathbf{P}(e \mid H, B)$, the likelihood of the example e.*

The task of probabilistic inductive logic programming is then to find the hypothesis H^* that maximizes the likelihood of the data $\mathbf{P}(E \mid H^*, B)$, where E denotes the set of examples. Under the usual i.i.d. assumption this results in the maximization of $\mathbf{P}(E \mid H^*, B) = \prod_{e \in E} \mathbf{P}(e \mid H^*, B)$.

The key contribution of this paper is that we present three probabilistic inductive logic programming settings that extend the traditional ones sketched above.

3.1 Probabilistic Entailment

In order to integrate probabilities in the entailment setting, we need to find a way to assign probabilities to clauses that are entailed by an annotated logic program. Since most inductive logic programming systems working under entailment employ ground facts for a single predicate as examples and the authors are unaware of any existing probabilistic logic programming formalisms that implement a probabilistic covers relation for definite clauses in general, we will restrict our attention in this section to assign probabilities to facts for a single predicate. Furthermore, our proposal in this section proceeds along the lines of the naïve Bayes' framework and represents only one possible choice for probabilistic entailment. It remains an open question as how to formulate more general frameworks for working with entailment (one alternative setting is also presented in Section 3.3).

More formally, let us annotate a logic program H consisting of a set of clauses of the form $p \leftarrow b_i$, where p is an atom of the form $p(V_1, ..., V_n)$ with the V_i different variables, and the b_i are different bodies of clauses. Furthermore, we associate to each clause in H the probability values $\mathbf{P}(b_i \mid p)$; they constitute the conditional probability distribution that for a random substitution θ for which $p\theta$ is ground and true (resp. false), the query $b_i\theta$ succeeds (resp. fails) in the knowledge base B. [2] Furthermore, we assume the prior probability of p is given as $\mathbf{P}(p)$, it denotes the probability that for a random substitution θ, $p\theta$ is true (resp. false). This can then be used to define the covers relation $\mathbf{P}(p\theta \mid H, B)$ as follows (we delete the B as it is fixed):

$$\mathbf{P}(p\theta \mid H) = \mathbf{P}(p\theta \mid b_1\theta, ..., b_k\theta) = \frac{\mathbf{P}(b_1\theta, ..., b_k\theta \mid p\theta) \times \mathbf{P}(p\theta)}{\mathbf{P}(b_1\theta, ..., b_k\theta)} \tag{1}$$

Applying the naïve Bayes assumption yields

$$\mathbf{P}(p\theta \mid H) = \frac{\prod_i \mathbf{P}(b_i\theta \mid p\theta) \times \mathbf{P}(p\theta)}{\mathbf{P}(b_1\theta, ..., b_k\theta)} \tag{2}$$

Finally, since we can $P(p\theta \mid H) + P(\neg p\theta \mid H) = 1$, we can compute $P(p\theta \mid H)$ without $P(b_1\theta, ..., b_k\theta)$ through normalization.

Example 4. Consider again the mutagenicity domain and the following annotated logic program:

$$(0.01, 0.21) : \texttt{mutagenetic(M)} \leftarrow \texttt{atom(M, \_, \_, 8, \_)}$$
$$(0.38, 0.99) : \texttt{mutagenetic(M)} \leftarrow \texttt{bond(M,, A, 1)}, \texttt{atom(M, A, c, 22, \_)}, \texttt{bond(M, A,, 2)}$$

where we denote the first clause by $\texttt{b}_1$ and the second one by $\texttt{b}_2$, and the vectors on the left-hand side of the clauses specify $P(b_i\theta = true \mid p\theta = true)$ and $P(b_i\theta = true \mid p\theta = false)$. The covers relation assigns probaility 0.97 to example 225 because both features fail for $\theta = \{\texttt{M} \leftarrow 225\}$. Hence,

$$\begin{aligned} P(\ \texttt{mutagenetic(225)} = true, &\texttt{b}_1\theta = false,\ \texttt{b}_2\theta = false) \\ &= P(\ \texttt{b}_1\theta = false \mid \texttt{muta(225)} = true\ ) \\ &\quad \cdot P(\ \texttt{b}_2\theta = false \mid \texttt{muta(225)} = true\ ) \\ &\quad \cdot P(\ \texttt{mutagenetic(225)} = true\ ) \\ &= 0.99 \cdot 0.62 \cdot 0.31 = 0.19 \end{aligned}$$

[2] The query q succeeds in B if there is a substitution σ such that $B \models q\sigma$.

and $P(\ \texttt{mutagenetic(225)} = \mathit{false}, \texttt{b}_1\theta = \mathit{false},\ \texttt{b}_2\theta = \mathit{false}) = 0.79 \cdot 0.01 \cdot 0.68 = 0.005$. This yields

$$P(\ \texttt{muta(225)} = \mathit{true} \mid \texttt{b}_1\theta = \mathit{false},\ \texttt{b}_2\theta = \mathit{false}\}) = \frac{0.19}{0.19 + 0.005} = 0.97\ .$$

3.2 Probabilistic Interpretations

In order to integrate probabilities in the learning from interpretation setting, we need to find a way to assign probabilities to interpretations covered by an annotated logic program. In the past few years, this question has received a lot of attention and various different approaches have been developed such as [38]. In this paper, we choose Bayesian logic programs [21] as the probabilistic logic programming system because Bayesian logic programs combine Bayesian networks [37], which represent probability distributions over propositional interpretations, with definite clause logic. Furthermore, Bayesian logic programs have already been employed for learning.

The idea underlying Bayesian logic programs is to view ground atoms as random variables that are defined by the underlying definite clause programs. Furthermore, two types of predicates are distinguished: deterministic and probabilistic ones. The former are called *logical*, the latter *Bayesian*. Likewise we will also speak of Bayesian and logical atoms. A Bayesian logic program now consists of a set of of Bayesian (definite) clauses, which are expressions of the form $\texttt{A} \mid \texttt{A}_1, \ldots, \texttt{A}_n$ where $\texttt{A}$ is a Bayesian atom, $\texttt{A}_1, \ldots, \texttt{A}_n$, $n \geq 0$, are Bayesian and logical atoms and all variables are (implicitly) universally quantified. To quantify probabilistic dependencies, each Bayesian clause c is annotated with its conditional probability distribtion $\mathrm{cpd}(c) = \mathbf{P}(\texttt{A} \mid \texttt{A}_1, \ldots, \texttt{A}_n)$, which quantifies as a macro the probabilistic dependency among ground instances of the clause.

Let us illustrate Bayesian logic programs on Jensen's stud farm example [19], which describes the processes underlying a life threatening heridatary disease.

Example 5. Consider the following Bayesian clauses:

$$\texttt{carrier(X)} \mid \texttt{founder(X)}. \tag{3}$$

$$\texttt{carrier(X)} \mid \texttt{mother(M,X)},\ \texttt{carrier(M)},\ \texttt{father(F,X)},\ \texttt{carrier(F)}. \tag{4}$$

$$\texttt{suffers(X)} \mid \texttt{carrier(X)}. \tag{5}$$

They specify the probabilistic dependencies governing the inheritance process. For instance, clause (4) says that the probability for a horse being a carrier of the disease depends on its parents being carriers.

In this example, the `mother`, `father`, and `founder` are logical, whereas the other ones, such as `carrier` and `suffers`, are Bayesian. The logical predicates are then defined by a classical definite clause program which constitute the background theory for this example. It is listed as interpretation B in Example 2.

The conditional probability distributions for the Bayesian clause are

$P(\texttt{carrier(X)} = \mathit{true})$
0.6

carrier(X)	$P(\texttt{suffers(X)} = \mathit{true})$
true	0.7
false	0.01

carrier(M)	carrier(F)	P(carrier(X) $= true$)
$true$	$true$	0.6
$true$	$false$	0.5
$false$	$true$	0.5
$false$	$false$	0.0

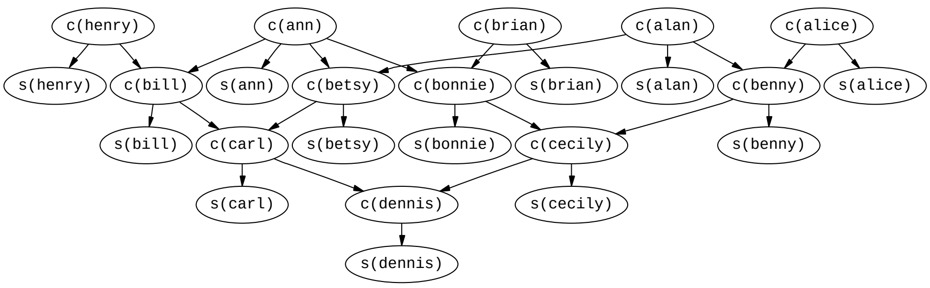
Observe that logical atoms, such as mother(M, X), do not affect the distribution of Bayesian atoms, such as carrier(X), and are therefore not considered in the conditional probability distribution. They only provide variable bindings, e.g., between carrier(X) and carrier(M).

By now, we are able to define the covers relation for Bayesian logic programs. A set of Bayesian logic program together with the background theory induces a Bayesian network. The random variables A of the Bayesian network are the Bayesian ground atoms in the least Herbrand model I of the annotated logic program. A Bayesian ground atom, say carrier(alan), influences another Bayesian ground atom, say carrier(betsy), if and only if there exists a Bayesian clause c such that

1. carrier(alan) $\in body(c)\theta \subseteq I$, and
2. carrier(betsy) $\equiv head(c)\theta \in I$.

Each node A has cpd$(c\theta)$ as associated conditional probability distribution. If there are multiple ground instances in I with the same head, a *combining rule* combine$\{\cdot\}$ is used to quantified the combined effect. A combining rule is a function that maps finite sets of conditional probability distributions onto one (*combined*) conditional probability distribution. Examples of combining rules are *noisy-or*, and *noisy-and*, see e.g. [19]

Example 6. The Stud farm Bayesian logic program induces the following Bayesian network.



Note that we assume that the induced network is acyclic and has a finite branching factor. The probability distribution induced is now

$$\mathbf{P}(I|H) = \prod_{\text{Bayesian atom } \mathtt{A} \in I} \text{combine}\{\text{cpd}(c\theta)\ |body(c)\theta \subseteq I, head(c)\theta \equiv \mathtt{A}\}. \quad (6)$$

From Equation (6), we can see that an example e consists of a **logical part** which is a Herbrand interpretation of the annotated logic program, and a **probabilistic part** which is a partial state assignment of the random variables occuring in the logical part.

Example 7. A possible example e in the Stud farm domain is

$$\begin{array}{lll}
\{\texttt{carrier(henry)} = false, & \texttt{suffers(henry)} = false, & \texttt{carrier(ann)} = true, \\
\texttt{suffers(ann)} = false, & \texttt{carrier(brian)} = false, & \texttt{suffers(brian)} = false, \\
\texttt{carrier(alan)} = ?, & \texttt{suffers(alan)} = false, & \texttt{carrier(alice)} = false, \\
\texttt{suffers(alice)} = false, & \ldots\} &
\end{array}$$

where ? denotes an unobserved state. The covers relation for e can now be computed using any Bayesian network inference engine based on Equation (6).

3.3 Probabilistic Proofs

The framework of stochastic logic programs [12, 31, 4] was inspired on research on stochastic context free grammars [1, 26]. The analogy between context free grammars and logic programs is that

1. grammar rules correspond to definite clauses,
2. sentences (or strings) to atoms, and
3. derivations to proofs.

Furthermore, in stochastic context-free grammars, the rules are annotated with probability labels in such a way that the sum of the probabilities associated to the rules defining a non-terminal is 1.0 .

Eisele and Muggleton have exploited this analogy to define stochastic logic programs. These are essentially definite clause programs, where each clause c has an associated probability label p_c such that the sum of the probabilities associated to the rules defining any predicate is 1.0 (though less restricted versions have been considered as well [4]).

This framework now allows to assign probabilities to proofs for a given predicate q given a stochastic logic program $H \cup B$ in the following manner. Let D_q denote the set of all possible ground proofs for atoms over the predicate q. For simplicity reasons, it will be assumed that there is a finite number of such proofs and that all proofs are finite (but again see [4] for the more general case). Now associate to each proof $t_q \in D_q$ the value

$$v_t = \prod_c p_c^{n_{c,t}}$$

where the product ranges over all clauses c and $n_{c,t}$ denotes the number of times clause c has been used in proof t_q. For stochastic context free grammars, the values v_t correspond to the probabilities of the derivations. However, the difference between context free grammars and logic programs is that in grammars two rules of the form $n \rightarrow q, n_1, ..., n_m$ and $q \rightarrow q_1, ..., q_k$ always 'resolve' to give $n \rightarrow q_1, ..., q_k, n_1, ..., n_m$ whereas resolution may fail due to unification. Therefore, the probability of a proof tree t in D_q

$$P(t \mid H, B) = \frac{v_t}{\sum_{s \in D_q} v_s} \; .$$

The probability of a ground atom is then defined as the sum of all the probabilities of all the proofs for that ground atom.

Example 8. Consider a stochastic variant of the definite clause grammer in Example 3 with uniform probability values for each predicate. The value v_u of the proof (tree) u in Example 3 is $v_u = \frac{1}{3} \cdot \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{12}$. The only other ground proofs s_1, s_2 of atoms over the predicate `sentence` are those of $\texttt{sentence}([\texttt{a}, \texttt{turtle}, \texttt{sleeps}], [])$ and $\texttt{sentence}([\texttt{the}, \texttt{turtle}, \texttt{sleeps}], [])$. Both get value $v_{s_1} = v_{s_2} = \frac{1}{12}$, too. Because there is only one proof for each of the sentences, $P(\texttt{sentence}([\texttt{the}, \texttt{turtles}, \texttt{sleep}], [])) = v_u = \frac{1}{3}$.

At this point, there are at least two different settings for probabilistic inductive logic programming using stochastic logic programs. The first actually corresponds to a learning from entailment setting in which the examples are ground atoms entailed by the target stochastic logic program. This setting has been studied by Cussens [5], who solves the parameter estimation problem, and Muggleton [32, 33], who presents a preliminary approach to structure learning (concentrating on the problem of adding one clause to an existing stochastic logic program).

In the second setting, which we sketch below, the idea is to employ learning from proofs instead of entailment. This significantly simplifies the structure learning process because proofs carry a lot more information about the structure of the underlying stochastic logic program than clauses that are entailed or not. Therefore, it should be much easier to learn from proofs. Furthermore, learning stochastic logic programs from proofs can be considered an extension of the work on learning stochastic grammars. It should therefore also be applicable to learning unification based grammars.

4 Probabilistic Inductive Logic Programming

By now, the problem of probabilistic inductive logic programming can be defined as follows:

Given
- a set of examples E,
- a probabilistic covers relation $P(e \mid H, B)$,
- a probabilistic logic programming representation language, and
- possibly a background theory B.

Find the hypothesis $H^* = argmax_H P(E \mid H, B)$.

Because $H = (L, \lambda)$ is essentially a logic program L annotated with probabilistic parameters λ, one distinguishes in general two subtasks:

1. *Parameter estimation*, where it is assumed that the underlying logic program L is fixed, and the learning task consists of estimating the parameters λ that maximize the likelihood.
2. *Structure learning*, where both L and λ have to be learned from the data.

Below, we briefly sketch the basic parameter estimation and structure learning techniques for probabilistic inductive logic programming. A complete survey of learning probabilistic logic representations can be found in [9].

4.1 Parameter Estimation

The problem of parameter estimation is thus concerned with estimating the values of the parameters λ^* of a fixed probabilistic logic program L that best explain the examples E. So, λ is a set of parameters and can be represented as a vector. As already indicated above, to measure the extent to which a model fits the data, one usually employs the likelihood of the data, i.e. $P(E \mid L, \lambda)$, though other scores or variants could – in principle – be used as well.

When all examples are fully observable, maximum likelihood reduces to frequency counting. In the presence of missing data, however, the maximum likelihood estimate typically cannot be written in closed form. It is a numerical optimization problem, and all known algorithms involve nonlinear optimization The most commonly adapted technique for probabilistic logic learning is the Expectation-Maximization (EM) algorithm [10, 28]. EM is based on the observation that learning would be easy (i.e., correspond to frequency counting), if the values of all the random variables would be known. Therefore, it first estimates these values, and then uses these to maximize the likelihood, and then iterates. More specifically, EM assumes that the parameters have been initialized (e.g., at random) and then iteratively performs the following two steps until convergence:

E-Step: On the basis of the observed data and the present parameters of the model, compute a distribution over all possible completions of each partially observed data case.

M-Step: Using each completion as a fully-observed data case weighted by its probability, compute the updated parameter values using (weighted) frequency counting.

The frequencies over the completions are called the *expected counts*.

4.2 Structure Learning

The problem is now to learn both the structure L and the parameters λ of the probabilistic logic program from data. Often, further information is given as well. It can take various different forms, including:

1. a *language bias* that imposes restrictions on the syntax of the definite clauses allowed in L,
2. a *background theory*,
3. an *initial hypothesis* (L, λ) from which the learning process can start, and
4. a scoring function $score(L, \lambda, E)$ that may correct the maximum likelihood principle for complex hypthesis; this can be based on a Bayesian approach which takes into account priors or the minimum description length princple.

Nearly all (score-based) approaches to structure learning perform a heuristic search through the space of possible hypotheses. Typically, hill-climbing or beam-search is applied until the hypothesis satisfies the logical constraints and the score(H, E) is no longer improving. The logical constraints typically require that the exammples are covered in the logical sense. E.g., when learning stochastic logic programs from entailment, the example clauses must be entailed by the logic program, and when learning Bayesian logic programs, the interpretation must be a model of the logic program.

At this point, it is interesting to observe that in the learning from entailment setting the examples do have to be covered in the logical sense when using the setting combining FOIL and naïve Bayes, whereas using the stochastic logic programs all examples must be logically entailed.

The steps in the search-space are typically made through the application of so-called refinement operators [43, 36], which make perform small modifications to a hypothesis. From a logical perspective, these refinement operators typically realize elementary generalization and specialization steps (usually under θ-subsumption).

Before concluding, we will now sketch for each probabilistic inductive logic learning setting a structure learning algorithm.

4.3 Learning from Probabilistic Entailment

One promising approach to address learning from probabilistic entailment is to adapt FOIL [41] with the conditional likelihood as described in Equation (2) as the scoring function $score(L, \lambda, E)$ (see N. Landwehr, K. Kersting and L. De Raedt, forthcoming, for more details).

Given a training set E containing positive and negative examples (i.e. true and false ground facts), this algorithm computes Horn clause features $b_1, b_2, \ldots$ in an outer loop. It terminates when no further improvements in the score are obtained, i.e, when $score(\{b_1, \ldots, b_i\}, \lambda_i, E) < score(\{b_1, \ldots, b_{i+1}\}, \lambda_{i+1}, E)$, where λ denotes the maximum likelihood parameters. A major difference with FOIL is, however, that the covered positive examples are *not* removed.

The inner loop is concerned with inducing the next feature b_{i+1} top-down, i.e., from general to specific. To this aim it starts with a clause with an empty body, e.g., $\mathtt{muta(M)} \leftarrow$. This clause is then specialised by repeatedly adding atoms to the body, e.g., $\mathtt{muta(M)} \leftarrow \mathtt{bond(M, A, 1)}$, $\mathtt{muta(M)} \leftarrow \mathtt{bond(M, A, 1), atom(M, A, c, 22, \_)}$, etc. For each refinement b'_{i+1} we then compute the maximum-likelihood parameters λ'_{i+1} and $score(\{b_1, \ldots, b'_{i+1}\}, \lambda'_{i+1}, E)$. The refinement that scores best,

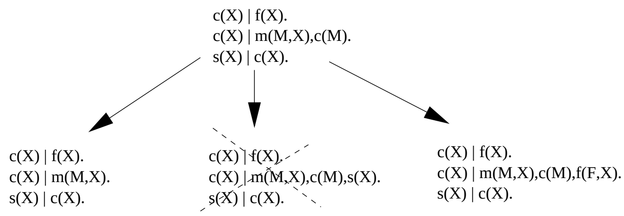


Fig. 1. The use of refinement operators during structural search within the framework of Bayesian logic programs. We can add an atom or delete an atom from the body of a clause. Candidates crossed out illegal because they are cyclic. Other refinement operators are reasonable such as adding or deleting logically valid clauses.

say b''_{i+1}, is then considered for further refinement and the refinement process terminates when $score(\{b_1, \ldots, b_{i+1}\}, \lambda_{i+1}, E) < score(\{b_1, \ldots, b''_{i+1}\}, \lambda''_{i+1}, E)$. Preliminary results with a prototype implementation are promising.

4.4 Learning from Probabilistic Interpretations

Scooby [22, 20, 23] is a greedy hill-climbing approach for learning Bayesian logic programs. Scooby takes the initial Bayesian logic program $H = (L, \lambda)$ as starting point and computes the parameters maximizing $score(L, \lambda, E)$. Then, refinement operators generalizing respectively specializing H are used to to compute all legal neighbours of H in the hypothesis space, see Figure 1. Each neighbour is scored. Let $H' = (L', \lambda')$ be the legal neighbour scoring best. If $score(L, \lambda, E) < score(L', \lambda', E)$ then Scooby takes H' as new hypothesis. The process is continued until no improvements in score are obtained.

Scooby is akin to theory revision approaches in inductive logic prorgamming. In case that only propositional clauses are considered, Scooby coincides with greedy hill-climbing approaches for learning Bayesian networks [16].

4.5 Learning from Probabilistic Proofs

Given a training set E containing ground proofs as examples, one possible approach combines ideas from the early inductive logic programming system Golem [34] that employs Plotkin's [39] least general generalization (LGG) with bottom-up generalization of grammars and hidden Markov models [46]. The resulting algorithm employs the likelihood of the proofs $score(L, \lambda, E)$ as the scoring function. It starts by taking as L_0 the set of ground clauses that have been used in the proofs in the training set and scores it to obtain λ_0. After intialization, the algorithm will then repeatedly select a pair of clauses in L_i, and replace the pair by their LGG to yield a candidate L'. The candidate that scores best is then taken as $H_{i+1} = (L_{i+1}, \lambda_{i+1})$, and the process iterates untill the score no longer improves. One interesting issue is that strong logical constraints can be imposed on the LGG. These logical constraints directly follow from the fact that the example proofs should still be valid proofs for the logical component L of all hypotheses considered. Therefore, it makes sense to apply the LGG only to clauses that define the same predicate, that contain the same predicates, and whose (reduced) LGG also has the same length as the original clauses. More details on this procedure will be worked out in a forthcoming paper (by S. Torge, K. Kersting and L. De Raedt).

5 Conclusions

In this paper, we have presented three settings for probabilistic inductive logic programming: *learning from entailment, from interpretations and from proofs*. We have also sketched how inductive logic programming and probabilistic learning techniques can – in principle – be combined to address these settings. Nevertheless, more work is needed before these techniques will be as applicable as traditional probabilistic learning or inductive logic programming systems. The authors hope that this paper will inspire and motivate researchers in probabilistic learning and inductive logic programming to join the exciting new field lying at the intersection of probabilistic reasoning, logic programming and machine learning.

Acknowledgements: The authors would like to thank Niels Landwehr and Sunna Torge for interesting collaborations on probabilisitc learning from entailment and proofs. This research was supported by the European Union under contract number FP6-508861, *Application of Probabilistic Inductive Logic Programming II.*

Appendix: Logic Programming Concepts

A first order *alphabet* is a set of predicate symbols, constant symbols and functor symbols. A *definite clause* is a formula of the form $A \leftarrow B_1, ..., B_n$ where A and B_i are logical atoms. An atom $p(t_1, ..., t_n)$ is a predicate symbol p/n followed by a bracketed n-tuple of terms t_i. A term t is a variable V or a function symbol $f(t_1, ..., t_k)$ immediately followed by a bracketed n-tuple of terms t_i. Constants are function symbols of arity 0. *Functor-free* clauses are clauses that contain only variables as terms. The above clause can be read as A if B_1 and ... and B_n. All variables in clauses are universally quantified, although this is not explicitly written. We call A the *head* of the clause and $B_1, ..., B_n$ the *body* of the clause. A *fact* is a definite clause with an empty body, $(m = 1,\ n = 0)$. Throughout the paper, we assume that all clauses are *range restricted*, which means that all variables occurring in the head of a clause also occur in its body. A *substitution* $\theta = \{V_1 \leftarrow t_1, ..., V_k \leftarrow t_k\}$ is an assignment of terms to variables. Applying a substitution θ to a clause, atom or term e yields the expression $e\theta$ where all occurrences of variables V_i have been replaced by the corresponding terms. A *Herbrand interpretation* is a set of ground facts over an alphabet A. A Herbrand interpretation I is a model for a clause c if and only if for all θ such that

$body(c)\theta \subseteq I \rightarrow head(c)\theta \in I$; it is a model for a set of of clauses H if and only if it is a model for all clauses in H. We write $H \models e$ if and only if all models of H are also a model of e.

References

1. S. Abney. Stochastic Attribute-Value Grammars. *Computational Linguistics*, 23(4):597–618, 1997.
2. C. R. Anderson, P. Domingos, and D. S. Weld. Relational Markov Models and their Application to Adaptive Web Navigation. In D. Hand, D. Keim, O. R. Zaïne, and R. Goebel, editors, *Proceedings of the Eighth International Conference on Knowledge Discovery and Data Mining (KDD-02)*, pages 143–152, Edmonton, Canada, 2002. ACM Press.
3. F. Bergadano and D. Gunetti. *Inductive Logic Programming: From Machine Learning to Software Engeneering*. MIT Press, 1996.
4. J. Cussens. Loglinear models for first-order probabilistic reasoning. In K. B. Laskey and H. Prade, editors, *Proceedings of the Fifteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI-99)*, pages 126–133, Stockholm, Sweden, 1999. Morgan Kaufmann.
5. J. Cussens. Parameter estimation in stochastic logic programs. *Machine Learning*, 44(3):245–271, 2001.
6. L. De Raedt. Logical settings for concept-learning. *Artificial Intelligence*, 95(1):197–201, 1997.
7. L. De Raedt and L. Dehaspe. Clausal discovery. *Machine Learning*, 26(2-3):99–146, 1997.
8. L. De Raedt and S. Džeroski. First-Order jk-Clausal Theories are PAC-Learnable. *Artificial Intelligence*, 70(1-2):375–392, 1994.
9. L. De Raedt and K. Kersting. Probabilistic Logic Learning. *ACM-SIGKDD Explorations: Special issue on Multi-Relational Data Mining*, 5(1):31–48, 2003.
10. A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *J. Royal Stat. Soc.*, B 39:1–39, 1977.
11. T. Dietterich, L. Getoor, and K. Murphy, editors. *Working Notes of the ICML-2004 Workshop on Statistical Relational Learning and its Connections to Other Fields (SRL-04)*, 2004.
12. A. Eisele. Towards probabilistic extensions of contraint-based grammars. In J. Dörne, editor, *Computational Aspects of Constraint-Based Linguistics Decription-II*. DYNA-2 deliverable R1.2.B, 1994.
13. J. Fürnkranz. Separate-and-Conquer Rule Learning. *Artificial Intelligence Review*, 13(1):3–54, 1999.
14. L. Getoor and D. Jensen, editors. *Working Notes of the IJCAI-2003 Workshop on Learning Statistical Models from Relational Data (SRL-03)*, 2003.
15. P. Haddawy. Generating Bayesian networks from probabilistic logic knowledge bases. In R. López de Mántaras and D. Poole, editors, *Proceedings of the Tenth Annual Conference on Uncertainty in Artificial Intelligence (UAI-1994)*, pages 262–269, Seattle, Washington, USA, 1994. Morgan Kaufmann.
16. D. Heckerman. A Tutorial on Learning with Bayesian Networks. Technical Report MSR-TR-95-06, Microsoft Research, March 1995.

17. N. Helft. Induction as nonmonotonic inference. In R. J. Brachman, H. J. Levesque, and R. Reiter, editors, *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning (KR-1989)*, pages 149–156, Toronto, Canada, May 15-18 1989. Morgan Kaufmann.
18. M. Jaeger. Relational Bayesian networks. In D. Geiger and P. P. Shenoy, editors, *Proceedings of the Thirteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI-97)*, pages 266–273, Providence, Rhode Island, USA, 1997. Morgan Kaufmann.
19. F. V. Jensen. *Bayesian networks and decision graphs*. Springer-Verlag New, 2001.
20. K. Kersting and L. De Raedt. Adaptive Bayesian Logic Programs. In C. Rouveirol and M. Sebag, editors, *Proceedings of the Eleventh Conference on Inductive Logic Programming (ILP-01)*, volume 2157 of *LNCS*, Strasbourg, France, 2001. Springer.
21. K. Kersting and L. De Raedt. Bayesian logic programs. Technical Report 151, University of Freiburg, Institute for Computer Science, April 2001.
22. K. Kersting and L. De Raedt. Towards Combining Inductive Logic Programming and Bayesian Networks. In C. Rouveirol and M. Sebag, editors, *Proceedings of the Eleventh Conference on Inductive Logic Programming (ILP-01)*, volume 2157 of *LNCS*, Strasbourg, France, 2001. Springer.
23. K. Kersting and L. De Raedt. Principles of Learning Bayesian Logic Programs. Technical Report 174, University of Freiburg, Institute for Computer Science, June 2002.
24. K. Kersting, T. Raiko, S. Kramer, and L. De Raedt. Towards discovering structural signatures of protein folds based on logical hidden markov models. In R. B. Altman, A. K. Dunker, L. Hunter, T. A. Jung, and T. E. Klein, editors, *Proceedings of the Pacific Symposium on Biocomputing*, pages 192 – 203, Kauai, Hawaii, USA, 2003. World Scientific.
25. D. Koller and A. Pfeffer. Probabilistic frame-based systems. In C. Rich and J. Mostow, editors, *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-1998)*, pages 580–587, Madison, Wisconsin, USA, July 1998. AAAI Press.
26. C. H. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing*. The MIT Press, 1999.
27. M. Marcus, M. A. Marcinkiewicz, and B. Santorini. Building a large annotated corpus of English: The Penn TREEBANK. *Computational Linguistics*, 19(2):313–330, 1993.
28. G. J. McKachlan and T. Krishnan. *The EM Algorithm and Extensions*. John Eiley & Sons, Inc., 1997.
29. T. M. Mitchell. *Machine Learning*. The McGraw-Hill Companies, Inc., 1997.
30. S. H. Muggleton. Inverse Entailment and Progol. *New Generation Computing Journal*, 13(3–4):245–286, 1995.
31. S. H. Muggleton. Stochastic logic programs. In L. De Raedt, editor, *Advances in Inductive Logic Programming*, pages 254–264. IOS Press, 1996.
32. S. H Muggleton. Learning stochastic logic programs. *Electronic Transactions in Artificial Intelligence*, 4(041), 2000.
33. S. H. Muggleton. Learning structure and parameters of stochastic logic programs. In S. Matwin and C. Sammut, editors, *Proceedings of the Twelfth International Conference on Inductive Logic Prgramming (ILP-02)*, volume 2583 of *LNCS*, pages 198–206, Sydney, Australia, 2002. Springer.
34. S. H. Muggleton and C. Feng. Efficient induction of logic programs. In S. H. Muggleton, editor, *Inductive Logic Programming*, pages 281–298. Acadamic Press, 1992.

35. L. Ngo and P. Haddawy. Answering queries from context-sensitive probabilistic knowledge bases. *Theoretical Computer Science*, 171(1–2):147–177, 1997.
36. S.-H. Nienhuys-Cheng and R. de Wolf. *Foundations of Inductive Logic Programming*. Springer-Verlag, 1997.
37. J. Pearl. *Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 2. edition, 1991.
38. A. J. Pfeffer. *Probabilistic Reasoning for Complex Systems*. PhD thesis, Stanford University, 2000.
39. G. D. Plotkin. A note on inductive generalization. In *Machine Intelligence*, volume 5, pages 153–163. Edinburgh University Press, 1970.
40. D. Poole. Probabilistic Horn abduction and Bayesian networks. *Artificial Intelligence*, 64(1):81–129, 1993.
41. J. R. Quinlan and R. M. Cameron-Jones. Induction of logic programs: FOIL and related systems. *New Generation Computing*, 13(3–4):287–312, 1995.
42. T. Sato. A Statistical Learning Method for Logic Programs with Distribution Semantics. In L. Sterling, editor, *Proceedings of the Twelfth International Conference on Logic Programming (ICLP-1995)*, pages 715 – 729, Tokyo, Japan, 1995. MIT Press.
43. E. Shapiro. *Algorithmic Program Debugging*. MIT Press, 1983.
44. A. Srinivasan. *The Aleph Manual*. Available at `http://www.comlab.ox.ac.uk/oucl/řesearch/areas/machlearn/Aleph/`.
45. A. Srinivasan, S. H. Muggleton, R. D. King, and M. J. E. Sternberg. Theories for mutagenicity: A study of first-order and feature based induction. *Artificial Intelligence*, 85(1–2):277–299, 1996.
46. A. Stolcke and S. Omohundro. Inducing Probabilistic Grammars by Bayesian Model Merging. In R. C. Carrasco and J. Oncina, editors, *Proceedings of the Second International Colloquium on Grammatical Inference and Applications (ICGI-94)*, number 862 in LNCS, pages 106–118, Alicante, Spain, September 21-23 1994.
47. L. G. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, November 1984.