Towards Learning Stochastic Logic Programs from Proof-Banks

Luc De Raedt and **Kristian Kersting** and **Sunna Torge**
University of Freiburg, Machine Learning Lab
Georges-Koehler-Allee, Building 079, 79110 Freiburg, Germany
{deraedt,kersting,torge}@informatik.uni-freiburg.de

Abstract

Stochastic logic programs combine ideas from probabilistic grammars with the expressive power of definite clause logic; as such they can be considered as an extension of probabilistic context-free grammars. Motivated by an analogy with learning tree-bank grammars, we study how to learn stochastic logic programs from proof-trees. Using proof-trees as examples imposes strong logical constraints on the structure of the target stochastic logic program. These constraints can be integrated in the *least general generalization* (lgg) operator, which is employed to traverse the search space. Our implementation employs a greedy search guided by the maximum likelihood principle and failure-adjusted maximization. We also report on a number of simple experiments that show the promise of the approach.

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 (De Raedt & Kersting 2003; Getoor & Jensen 2003; Dietterich, Getoor, & Murphy 2004). This work is known under the names of statistical relational learning (SRL), probabilistic logic learning, or probabilistic inductive logic programming. By combining logic, probability, and machine learning, SRL approaches can perform robust and accurate reasoning and learning about complex relational data, that is, data stored in more than a single relational table. A great variety of SRL approaches has been proposed, such as PRISM (Sato & Kameya 1997), ICL (Poole 1993), SLPs (Muggleton 1996), BLPs (Kersting & De Raedt 2001), PRMs (Getoor *et al*. 2001), and MLNs (Domingos & Richardson 2004). Typically, two different learning problems are considered: parameter estimation and structure learning. So far, with a few notable exceptions (Getoor *et al*. 2001; Kersting & De Raedt 2001; Muggleton 2002; Domingos & Richardson 2004), the vast majority of present SRL approaches has focused on parameter estimation. The main reason is that in order to learn the structure of a probabilistic inductive logic programming model, one needs to solve the underlying inductive logic programming problem, which is a non-trivial task by itself, as well as the parameter estimation problem.

In this paper, we focus on learning the structure of stochastic logic programs (SLPs), a probabilistic logic introduced by Muggleton (1996). To the best of the authors' knowledge, the only approach so far to learn the structure of SLPs is due to Muggleton (2002). This approach incrementally learns an additional clause for a single predicate in a SLP. From an inductive logic programming perspective, this corresponds to a typical single predicate learning setting under entailment. Indeed, the examples are ground facts for a single predicate and the goal is to learn the corresponding predicate definition annotated with probability labels. Learning logic programs (involving multiple predicates) under entailment (De Raedt 1997) is hard because the examples are not very informative. Therefore, alternative settings, such as learning from interpretations and learning from traces (Shapiro 1983; Bergadano & Gunetti 1996) have been considered in ILP. This raises the question as to whether it is possible to learn SLPs in these other settings.

The main contribution of the present paper is that we show how to learn the structure of SLPs from proofs. In this setting, the examples are proof-trees of the unknown target SLP. This *learning from proofs* setting is not only motivated from an inductive logic programming perspective, it is also motivated by the work on learning probabilistic context-free grammars (Charniak 1996) and combinatory categorial grammars (Hockenmaier 2003) from tree-banks. Indeed, within the empirical natural language processing community, the learning of probabilistic context-free grammars from parse trees (such as the Wall-Street Journal) is an established and successful practice. Because SLPs upgrade probabilistic context-free grammars, and parse trees for probabilistic context-free grammars correspond to proof-trees for SLPs, it seems natural to try to upgrade the tree-bank grammar to SLPs. This is in line with the common practice in inductive logic programming to derive new inductive logic programming systems by upgrading existing ones for simpler representations.

Stochastic Logic Programs (SLPs)

SLPs combine definite clauses with ideas from probabilistic context-free grammars. A SLP is a definite clause program, where each of the clauses is labeled with a probability value.

Definition 1 *A stochastic logic program is a set of clauses* *of the form $p : h \leftarrow b_1, ..., b_m$, where h and the b_i are logical atoms and $p > 0$ is a probability label.*

Furthermore, as in probabilistic context-free grammars, we will assume that the sum of the probability values for all clauses defining a particular predicate q (i.e. with q in the left-hand side or head of the clause) is equal to 1 (though less restricted versions have been considered as well).

Example 1 *Consider the following probabilistic definite clause grammar*[1]*:*

$1 : \mathsf{s(A, B)} \leftarrow \mathsf{np(Number, A, C), vp(Number, C, B).}$
$1/2 : \mathsf{np(Number, A, B)} \leftarrow \mathsf{det(A, C), n(Number, C, B).}$
$1/2 : \mathsf{np(Number, A, B)} \leftarrow \mathsf{pronom(Number, A, B).}$
$1/2 : \mathsf{vp(Number, A, B)} \leftarrow \mathsf{v(Number, A, B).}$
$1/2 : \mathsf{vp(Number, A, B)} \leftarrow \mathsf{v(Number, A, C), np(D, C, B).}$
$1 : \mathsf{det(A, B)} \leftarrow \mathsf{term(A, the, B).}$
$1/4 : \mathsf{n(s, A, B)} \leftarrow \mathsf{term(A, man, B).}$
$1/4 : \mathsf{n(s, A, B)} \leftarrow \mathsf{term(A, apple, B).}$
$1/4 : \mathsf{n(pl, A, B)} \leftarrow \mathsf{term(A, men, B).}$
$1/4 : \mathsf{n(pl, A, B)} \leftarrow \mathsf{term(A, apples, B).}$
$1/4 : \mathsf{v(s, A, B)} \leftarrow \mathsf{term(A, eats, B).}$
$1/4 : \mathsf{v(s, A, B)} \leftarrow \mathsf{term(A, sings, B).}$
$1/4 : \mathsf{v(pl, A, B)} \leftarrow \mathsf{term(A, eat, B).}$
$1/4 : \mathsf{v(pl, A, B)} \leftarrow \mathsf{term(A, sing, B).}$
$1 : \mathsf{pronom(pl, A, B)} \leftarrow \mathsf{term(A, you, B).}$
$1 : \mathsf{term([A|B], A, B)} \leftarrow$

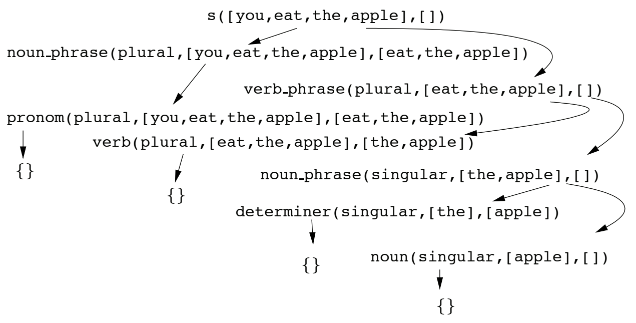
SLPs define a probability distribution over proof-trees. At this point, there exist various possible forms of proof-trees. In this paper, we will – for reasons that will become clear later – assume that the proof-tree is given in the form of an and-tree where the nodes contain ground atoms. More formally:

Definition 2 *t is a proof-tree for a SLP S if and only if t is a rooted tree where for all nodes $n \in t$ with children $child(n)$ there exists a substitution θ and a clause $c \in S$ such that $n = head(c)\theta$ and $child(n) = body(c)\theta$*[2].

If the property holds for all nodes, except for some of the leaves of the tree, we will talk about a *partial proof-tree*. A partial proof-tree is a representation of a (successful or failed) SLD-derivation, cf. also the Appendix. A *proper* proof-tree encodes an SLD-refutation, i.e. a successful SLD-derivation, see Figure 1 for an example. As probabilistic context-free grammars, SLPs define probability distributions over derivations and atoms (or sentences). However, there is one crucial difference between context-free grammars and logic programs. Resolution derivations for logic programs can fail, whereas derivations in a context-free grammar never fail. Indeed, if an intermediate grammar rule of the form $S \rightarrow t_1, ..., t_k, n_1, ..., n_m$ is derived in a context-free grammar, where the t_i are terminal symbols and the n_j are non-terminal ones, it is always possible to "resolve" a non-terminals n_j away using any rule defining n_j. In contrast, in a logic program, this is not always possible because the two literals may not unify. That is also the reason why some partial proof-trees cannot be expanded into a proof-tree.

[1]We follow the typical Prolog convention, where variables start with an upper-case and constants with a lower-case character.

[2]In order to avoid having to account for several representations of the same proof, we will assume that the children of a node are ordered in the same order as they occur in the clauses.



s([you,eat,the,apple],[])
noun_phrase(plural,[you,eat,the,apple],[eat,the,apple])
verb_phrase(plural,[eat,the,apple],[])
pronom(plural,[you,eat,the,apple],[eat,the,apple])
verb(plural,[eat,the,apple],[the,apple])
{}
noun_phrase(singular,[the,apple],[])
{}
determiner(singular,[the],[apple])
{}
noun(singular,[apple],[])
{}

Figure 1: A proof tree for the probabilistic definite clause grammar.

We can now define three different probability distributions (where we will – for simplicity – assume that all SLD-trees are finite, cf. (Cussens 2001) for a more general case). First, let us define the distribution over partial proof-trees or derivations of a variabilized atom a (i.e. an atom of the form $p(X_1, ..., X_n)$, where all the arguments are different variables). This is similar to the case for probabilistic context-free grammars: The probability $P_D(d)$ of a partial proof-tree d, in which the clauses $c_1, ..., c_n$ with associated probability labels $p_1, ..., p_n$ are used $m_1, ..., m_n$ times, is

$$P_D(d) = \prod_i p_i^{m_i} \ .$$

Observe that the probability distribution P_D also assigns a non-zero probability to failed derivations. Usually, we are interested in successful derivations or proof-trees, i.e. refutations ending in the empty clause. The probability distribution $P_R(r)$, defined on a refutation r for a variabilized atom a and induced by P_D and the logic program, can be obtained by normalizing P_D:

$$P_R(r) = P_D(r) / \left(\sum_{r(a)} P_D(r(a)) \right)$$

where the sum ranges over all proof-trees or refutations $r(a)$ of the atom a. This in turn allows us to define the probability $P_A(a\theta)$ of a ground atom $a\theta$:

$$P_A(a\theta) = \left(\sum_{r(a\theta)} P_R(r(a\theta)) \right) / \left(\sum_{r(a)} P_R(r(a)) \right)$$

where $r(a\theta)$ ranges over all proof-trees of $a\theta$ and $r(a)$ over all proof-trees of a.

Example 2 *Consider the previous SLP. The probability $P_D(u)$ of the proof (tree) u in Figure 1 is $P_D(u) = \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{4} \cdot \frac{1}{2} \cdot \frac{1}{4} = \frac{1}{128}$. All in all, there are 60 ground proofs $s_i, i = 1, ...60$ of atoms over the predicate* $\mathtt{s}$*. In order to get the value of $P_R(\mathtt{s([you, eat, the, apple], [])})$, first the probability $P_D(s_i)$ must be calculated for each proof-tree s_i. Secondly, all probabilities $P_D(s_i)$ are summed up.*

Multiplying the inverse of this sum with $\frac{1}{128}$ finally yields $P_R(\mathtt{s}([\mathtt{you}, \mathtt{eat}, \mathtt{the}, \mathtt{apple}], []))$.

By now we are able to define the learning setting addressed in this paper. Given are a set of proof-trees E that are sampled from an *unknown* SLP S, and the goal is to try to reconstruct S from E. As we are given only a sample from S it will – in general – be impossible to reconstruct S from E. Instead, we want to find the best possible approximation of S, i.e. the hypotheses H which maximizes the likelihood of the data. More formally, we can specify this problem as follows:

Given • a set of proof-trees E for an *unknown* target SLP selected according to P_R, and
• a space of possible SLPs $\mathcal{L}$,
Find a SLP $H \in \mathcal{L}$, such that
$$H = \arg\max_{H\in\mathcal{L}} \prod_{e\in E} P_R(e|H) \ .$$

For simplicity, we employ a maximum likelihood criterion here. Instead, one may want to employ a Bayesian or a minimum description length criterion.

This setting upgrades the problem of learning probabilistic context-free grammars from parse trees or tree-banks. It also introduces a novel setting for inductive logic programming, in which the examples are proof-trees rather than interpretations or clauses. To the best of the authors' knowledge, this setting has not yet been pursued within inductive logic programming even though there are certainly related approaches, most notably Bergadano and Gunetti's TRACY (Bergadano & Gunetti 1996) and Shapiro's seminal Model Inference System (Shapiro 1983). Whereas TRACY learns logic programs from traces, Shapiro's tries to reconstruct a trace by querying an oracle.

Observe also that this setting is meaningful in several applications. First, it applies very naturally to *natural language processing* problems, where one could e.g. try to induce probabilistic unification based grammar, e.g. probabilistic definite clause grammars, from parse trees. SLPs have actually been defined and inspired by applications in natural language processing, cf. (Muggleton 1996; Riezler 1998). This type of application is similar in spirit to our running example. Second, consider *web-mining* applications such as analyzing traces through web-sites as illustrated in the next example.

Example 3 *In the following stochastic logic program*

```
0.1 : deptp(D) ← dept(D).
0.5 : deptp(D) ← dept(D), prof(P), bel(D, P), profp(P).
0.4 : deptp(D) ← dept(D), cou(C), off(D, C), coursep(P).
0.1 : coursep(C) ← cou(C).
0.3 : coursep(C) ← cou(C), prof(P), tea(P, C), profp(P)
0.6 : coursep(C) ← cou(C), dept(D), off(D, P), deptp(D)
0.1 : dept(cs) ← ...
0.1 : prof(tom) ← ...
0.2 : cou(ai) ← ...
```

ground facts for predicates ending in `p` *denote a particular page, e.g.* `deptp(cs)`. *Other unary predicates such as* `cou` *denote a particular set of entities (in this case courses). Finally, binary relations represent possible transitions, such as* `bel(ongs), off(ers)` *and* `tea(ches)`. *Using this SLP, a web log corresponds to a proof-tree.*

This example is similar in spirit to the application and framework for Relational Markov Models (RMMs) (Anderson, Domingos, & Weld 2002). Third, in a similar setting one can well imagine applications within *bioinformatics*. The biological functions of proteins/RNA molecules depend on the way they fold. Kersting *et al.* (2003) report on promising results for protein fold classification. They represent the *secondary structures*[3] of proteins as sequences of ground atoms, i.e. proof chains. Similarly, the secondary structures of mRNA molecules contain special subsequences called signal structures that are responsible for special biological functions, such as RNA-protein interactions and cellular transport. In contrast to proteins, the secondary structures of mRNA molecules do not form chains but (proof) trees.

Learning from proofs

As for any probabilistic inductive logic programming approach, there are *logical* and *probabilistic* issues to be taken into account. The former are related to the structure of the SLP, i.e. the clauses, the latter to the parameters, i.e. the probability labels.

Logical issues

From a logical perspective, an important constraint is imposed on target SLPs by the *learning from proofs* setting. It follows from the following property.

Property 1 *For all SLPs S: e is a (proper) proof-tree of S if and only if $P_R(e|S) > 0$.*

This property implies that the learner should restrict its attention to those (stochastic) logic programs S for which all examples e are proper proof-trees (i.e. $P_R(e|S) > 0$). If there exists an example e that is no longer a proper proof-tree for a particular hypothesis H, $P_R(e|H) = 0$ and therefore, the likelihood of that hypothesis is 0. Hence, such hypotheses should not be considered during the search. This raises the question as to how to organize the search through the search space of (stochastic) logic programs.

A starting hypothesis H_0 that satisfies the above constraint can easily be obtained by collecting all ground clauses that appear in the examples E and assigning them non-zero probability labels. More formally, let

$$H_0 = \{n \leftarrow c_1, \cdots, c_m | \text{ node } n \text{ has children } c_i \text{ in } e \in E\}.$$

The hypothesis H_0 also corresponds to the initial grammar that is constructed from tree-banks. This grammar will however be too specific because it only contains ground clauses, which raises the question as to how we could generalize these clauses. Again, motivated by common practice in tree-bank grammars (Charniak 1996) and inductive logic programming (Muggleton & Feng 1992), we consider generalizing clauses by applying the well-known lgg operator introduced by (Plotkin 1970), cf. also the Appendix. One property of the lgg is that $lgg(c_1, c_2) \models c_i$, i.e. the lgg entails the

[3]The secondary structure is an intermediate representation of how proteins/RNA molecules fold up in nature.

original clauses. This means that if one replaces two clauses $c_1, c_2 \in H$ by $lgg(c_1, c_2)$ to yield H' all facts entailed by H are still entailed by H'. However, in our context, we have the logical constraint that all proof-trees for H must still be proof-trees for H'. This constraint does not hold in general.

Example 4 *Consider the clauses $h \leftarrow q$ and $h \leftarrow r$ and assume that the hypothesis H also contains the fact $q \leftarrow$. Then the proof-tree for h in H is no longer a valid proof-tree in H' consisting of $h \leftarrow$ and $q \leftarrow$.*

This provides us with a powerful means to prune away uninteresting generalizations. Indeed, whenever a candidate lgg does not preserve the proof-trees, it is overly general and should be pruned. A naive way of verifying this condition, would be to compute the proofs for the generalized hypothesis H'. However, this would be computationally very expensive. Fortunately, it is possible to guarantee this condition much more efficiently. Indeed, it suffices to verify that

1. $\exists$ substitutions $\theta_1, \theta_2 : lgg(c_1, c_2)\theta_1 = c_1$ and $lgg(c_1, c_2)\theta_2 = c_2$
2. there is a one-to-one mapping from literals in $lgg(c_1, c_2)\theta_i$ to literals in c_i

If the first condition is violated, then there are literals in c_i that do not occur in $lgg(c_1, c_2)\theta_i$. These literals would be lost from the corresponding proofs (as sketched in the example) and so the proofs would not be preserved. If the second condition is violated, a standard theorem prover would derive different proofs (containing more children), and again the proofs would not be preserved. Furthermore, as common in inductive logic programming, we will assume that all clauses in the target (and intermediate) logic programs are reduced w.r.t. θ-subsumption. The above conditions allow us to verify whether the proofs are preserved *after* computing the lgg. However, one needs to compute and consider only the lgg of two clauses if the multi-set of predicates occurring in these two clauses are identical.

Probabilistic Issues

From a probabilistic perspective, the examples also constrain the target SLP as each proof can be viewed as a sample from a Markov chain induced by the SLP, which constrains the probability labels associated to the clauses.

Once the structure of the SLP H is fixed, the problem of parameter estimation is that of finding those parameters λ^* of H that best explain the examples E. As already indicated earlier, we employ the maximum likelihood criterion

$$\lambda^* = \arg\max_{\lambda} P_D(E|H, \lambda) .$$

(possibly with a penalty for the complexity of the model).

For SLPs, the parameter estimation problem has been thoroughly studied by James Cussens (2001). However, the setting he uses is that of learning from entailment, in which the examples are ground facts instead of proof-trees. Since proof-trees carry more information than facts, it is possibly to adapt and simplify Cussens' *Failure Adjusted Maximization* (FAM) approach for our purposes.

Cussens's FAM approach is essentially an instance of the *Expectation - Maximization* (EM) algorithm. The EM scheme is based on the observation that parameter estimation would correspond to frequency counting, if the values of all random variables were known. Therefore, the EM iteratively performs two steps to find the maximum likelihood parameters:

(E-Step) Based on the current parameters λ and the observed data E, the algorithm computes a distribution over all possible completions of E.

(M-Step) Each completion is then treated as a fully observed example weighted by its probability. New parameters are then computed based on (expected) frequency counting.

In Cussens' FAM approach, the atoms are treated as an incomplete example set derived from a complete example set of derivations (i.e. proofs and failures), truncated to yield refutations only, and finally grouped to produce the set of observed atoms. As shown by (Cussens 2001), this yields the following formula for computing the expected counts of a clause C_i given E for the **E-Step**:

$$\mathrm{ec}_\lambda(C_i|E) = \sum_{e \in E} \left(\mathrm{ec}_\lambda(C_i|e) + (Z^{-1} - 1)\, \mathrm{ec}_\lambda(C_i|\texttt{fail}) \right) \quad (1)$$

Here, $\mathrm{ec}_\lambda(C_i|e)$ (resp. $\mathrm{ec}_\lambda(C_i|\texttt{fail})$) denotes the expected number of times clause C_i has been used to derive atom e (resp. to derive a failure), and Z is the normalization constant

$$Z = \sum_{r(h)} P_D(r(h))$$

where the sum ranges over all proof-trees $r(h)$ of the variabilized head h of clauses C_i. In the **M-Step**, FAM computes the improved probability label p_i for each clause C_i as

$$p_i \leftarrow \mathrm{ec}_\lambda(C_i|E) / \sum\nolimits_{C'} \mathrm{ec}_\lambda(C'|E)$$

where the sum ranges over all clauses C' with the same predicate in the head as C_i.

Because proofs carry more information than ground facts, we can directly compute the number of times $\mathrm{c}(C_i|e)$ clause C_i has been used in the given proof-tree e instead of having to compute the expected number $\mathrm{ec}_\lambda(C_i|e)$ clause C_i is used to prove an atom, in Equation (1). This is the only modification needed to adapt the FAM algorithm to the *learning from proofs* setting.

The Learning Algorithm

Our algorithm is summarized in Algorithm 1. It naturally takes both the logical and the probabilistic constraints imposed by the *learning from proofs* setting into account.

The algorithm performs a greedy hill-climbing search through the space of possible logic programs and is guided by the maximum likelihood score. The starting point of the search H_0 consists of the SLP containing all ground clauses in E. The lgg is then used to compute the legal "neighbours" (candidates, where subsumed clauses are removed) of H. The best neighbor (according to the score) is then selected

Algorithm 1 Learning SLPs from Proofs.

```
 1: $H_0 = \{n \leftarrow c_1, \cdots, c_m |$ node $n$ has children $c_i$ in $e \in E\}$
 2: estimate parameters of $H_0$
 3: compute $score(H_0, E)$
 4: $i := -1$
 5: **repeat**
 6:     $i := i + 1$
 7:     $H_{i+1} := H_i$
 8:     **for** all clauses $c_1, c_2 \in H_i$ with identical multi-sets of
        predicates **do**
 9:         compute the reduced $lgg(c_1, c_2)$
10:         **if** $lgg(c_1, c_2)$ preserves the proofs **then**
11:             $H' := H_i$
12:             delete from $H'$ all clauses subsumed by
                $lgg(c_1, c_2)$
13:             add $lgg(c_1, c_2)$ to $H'$
14:             re-estimate parameters of $H'$
15:             compute $score(H', E)$
16:             **if** $score(H', E)$ is better than $score(H_{i+1}, E)$
                **then**
17:                 $H_{i+1} := H'$
18: **until** $score(H_{i+1}, E) \approxeq score(H_i, E)$
```

and the search is repeated until no further improvements in score are obtained.

Instead of the algorithm presented, one could also easily adapt Friedman's structural EM (Friedman 1997). This algorithm would take the current model H_k and run the FAM algorithm for a while to get reasonably completed data. It would then fix the completed data case, i.e. the failures, and use them to compute the ML parameters λ' of each neighbour H' and choose the neighbour with the best improvement in score as the new best hypothesis H^{k+1}, if it improves upon H_k, and iterate.

Experimental Evaluation

Our intentions here are to investigate whether

(H) our proposed bottom-up learning approach can discover the SLP underlying a set of proofs.

To this aim, we implemented the algorithm using the SICS-TUS Prolog 3.10.1. Instead of computing and counting all failure derivations, sampling is used to avoid infinite computations. We use the log-likelihood as score. Following the *minimum description length* score for Bayesian networks, we use additionally the penalty $|H| \log(|E|)/2$. Data were generated from the SLP in example 1. Note that this is not a context-free grammar as failure derivations occur due to variable bindings.

From the target SLP, we generated (independent) training sets of 50, 100, 200, and 500 proofs. For each training set, 4 different random initial sets of parameters were tried. We ran the learning algorithm on each data set starting from each of the initial sets of parameters. The algorithm stopped when a limit of 200 iterations was exceeded or a change in log-likelihood between two successive iterations was smaller than 0.0001.

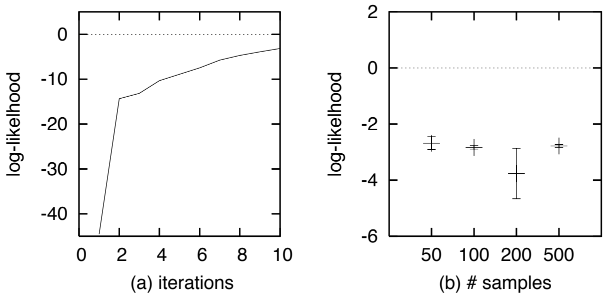


Figure 2: Experimental results. **(a)** A typical learning curve. **(b)** Final log-likelihood averaged over 4 runs. The error bars show the standard deviations.

Figure 2 **(a)** shows a typical learning curve, and Figure 2 **(b)** summarizes the overall results. In all runs, the original structure was induced from the proof-trees. This validates **H**. Moreover, already 50 proof-trees suffice to rediscover the structure of the original SLP. (Note, that example 1 yields 60 sentences.) Further experiments with 20 and 10 samples respectively show that even 20 samples suffice to learn the given structure. Sampling 10 proofs, the original structure is rediscovered in one of five experiments. This supports that the *learning from proof trees* setting carries more information than the *learning from entailment* setting. Furthermore, our methods scales well. Runs on two independently sampled sets of 1000 training proofs yield similar results: -4.77 and -3.17, and the original structure was learned in both cases.

Conclusions and Related Work

We have presented a novel framework for probabilistic inductive logic programming, in which one learns from proof-trees rather than from facts or interpretations. It was motivated by an analogy with learning of tree-bank grammars (Charniak 1996). The advantage of this framework is that it provides strong logical constraints on the structure of the stochastic logic program and at the same time simplifies the parameter estimation problem (as compared to (Cussens 2001)).

From an inductive logic programming perspective, this work is closest to the Golem system (Muggleton & Feng 1992), which also employed the lgg as its key operation to induce logic programs (without probability labels). It also extends Cussens' FAM algorithm with structure learning, and the work by Muggleton (2002) in that it learns a set of clauses simultaneously instead of a single predicate (or clause). The present work is also related to various approaches in grammar learning, such as the work on tree-bank grammars (Charniak 1996), and the work on generalizing grammars by for instance state-merging (e.g. (Cicchello & Kremer 2003)). Nevertheless, to the best of our knowledge, learning from proofs is a new setting for inductive logic programming, and has not yet been used in this form to learn probabilistic extensions of universal Turing machines.

Further work will be concerned with applying our framework to practical problems, both in natural language processing and in contexts where relational and logical Markov models have been used. As SLPs can be regarded as a generalization of RMMs (Anderson, Domingos, & Weld 2002) and LOHMMs (Kersting *et al.* 2003) they should essentially apply to this setting. It would also be interesting to see whether our techniques could be employed to learning unification based grammars such as HPSG (for instance for the HPSG tree bank for Bulgarian, see http://www.bultreebank.org/).

Acknowledgements: The authors would like to thank James Cussens for helpful discussions on parameter estimation for SLPs. The research was supported by the EU IST programme, contract no. FP6-508861, *Application of Probabilistic Inductive Logic Programming II*.

Appendix - Logic

SLD resolution is a logical inference operator for proving statements in first-order logic. Given a goal $\texttt{:-}\,\mathtt{G}_1, \mathtt{G}_2 \ldots, \mathtt{G}_\mathtt{n}$ and a clause $\mathtt{G}\texttt{:-}\mathtt{L}_1, \ldots, \mathtt{L}_\mathtt{m}$ such that $\mathtt{G}_1\theta = \mathtt{G}\theta$, applying SLD resolution yields the new goal $\texttt{:-}\,\mathtt{L}_1\theta, \ldots, \mathtt{L}_\mathtt{m}\theta, \mathtt{G}_2\theta \ldots, \mathtt{G}_\mathtt{n}\theta$. A *successful* refutation, i.e. a proof of a goal is then a sequence of resolution steps yielding the empty goal, i.e. :- . *Failed* proofs do not end in the empty goal.

The *least general generalization* lgg (under θ-subsumption) for clauses is defined recursively. The lgg of two terms is defined as follows: $lgg(t,t) = t$, $lgg(f(s_1,...,s_n), f(t_1,...,t_n)) = f(lgg(s_1,t_1),...,lgg(s_n,t_n))$, $lgg(f(s_1,...,s_n), g(t_1,...,t_m)) = V$ where $f \neq g$ and V is a variable. The lgg of two atoms $p(s_1,...,s_n)$ and $p(t_1,...,t_n)$ is $lgg(p(lgg(s_1,t_1),\ldots,lgg(s_n,t_n))$, and it is undefined if the sign or predicate symbols are unequal. Finally, the lgg of two clauses c_1 and c_2 is the clause $\{lgg(l_1,l_2) | lgg(l_1,l_2)$ is defined and $l_i \in c_i\}$.

References

Anderson, C.; Domingos, P.; and Weld, D. 2002. Relational Markov Models and their Application to Adaptive Web Navigation. In *Proceedings of the 21st International Conference on Knowledge Discovery and Data Mining (KDD-02)*, 143–152.

Bergadano, F., and Gunetti, D. 1996. *Inductive Logic Programming: From Machine Learning to Software Engeneering*. MIT Press.

Charniak, E. 1996. Tree-Bank Grammars. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, volume 2, 1031–1036.

Cicchello, O., and Kremer, S. 2003. Inducing Grammars from Sparse Data Sets: A Survey of Algorithms and Results. *JMLR* 4:603–632.

Cussens, J. 2001. Parameter estimation in stochastic logic programs. *Machine Learning* 44(3):245–271.

De Raedt, L., and Kersting, K. 2003. Probabilistic Logic Learning. *ACM-SIGKDD Explorations: Special issue on Multi-Relational Data Mining* 5(1):31–48.

De Raedt, L. 1997. Logical settings for concept-learning. *Artificial Intelligence* 95(1):197–201.

Dietterich, T.; Getoor, L.; and Murphy, K., eds. 2004. *Proceedings of the ICML Workshop on Statistical Relational Learning and its Connections to Other Fields (SRL-04)*.

Domingos, P., and Richardson, M. 2004. Markov Logic: A Unifying Framework for Statistical Relational Learning. In Dietterich et al. (2004), 49–54.

Friedman, N. 1997. Learning belief networks in the presence of missing values and hidden variables. In *Proceedings of the Fourteenth International Conference on Machine Learning (ICML-1997)*, 125–133.

Getoor, L., and Jensen, D., eds. 2003. *Working Notes of the IJCAI Workshop on Learning Statistical Models from Relational Data (SRL-03)*.

Getoor, L.; Friedman, N.; Koller, D.; and Pfeffer, A. 2001. Learning probabilistic relational models. In Džeroski, S., and Lavrač, N., eds., *Relational Data Mining*, 307–338. Springer.

Hockenmaier, J. 2003. *Data and models for statistical parsing with Combinatory Categorial Grammar*. Ph.D. Dissertation, School of Informatics, Univ. of Edinburgh.

Kersting, K., and De Raedt, L. 2001. Towards Combining Inductive Logic Programming and Bayesian Networks. In *Proceedings of the Eleventh Conference on Inductive Logic Programming (ILP-01)*, 118–131.

Kersting, K.; Raiko, T.; Kramer, S.; and De Raedt, L. 2003. Towards discovering structural signatures of protein folds based on logical hidden markov models. In *Proceedings of the Pacific Symposium on Biocomputing*, 192 – 203.

Muggleton, S., and Feng, C. 1992. Efficient induction of logic programs. In Muggleton, S., ed., *Inductive Logic Programming*, 281–298. Acadamic Press.

Muggleton, S. 1996. Stochastic logic programs. In De Raedt, L., ed., *Advances in Inductive Logic Programming*, 254–264. IOS Press.

Muggleton, S. 2002. Learning structure and parameters of stochastic logic programs. In *Proceedings of the Twelfth International Conference on Inductive Logic Prgramming (ILP-02)*, 198–206.

Plotkin, G. 1970. A note on inductive generalization. In *Machine Intelligence 5*, 153–163. Edinburgh Univ. Press.

Poole, D. 1993. Probabilistic Horn abduction and Bayesian networks. *Artificial Intelligence* 64:81–129.

Riezler, S. 1998. *Probabilistic Constraint Logic Programming*. Ph.D. Dissertation, Universität Tübingen, AIMS Report 5(1), IMS, Universität Stuttgart.

Sato, T., and Kameya, Y. 1997. Prism: A symbolic-statistical modeling language. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI-97)*, 1330–1339.

Shapiro, E. 1983. *Algorithmic Program Debugging*. MIT Press.