11 Stochastic Logic Programs: A Tutorial

Stephen Muggleton and Niels Pahlavi

Stochastic logic programs (SLPs)provide a simple scheme for representing probability distributions over structured objects. Other papers have concentrated on technical issues related to the semantics and machine learning of SLPs. By contrast, this chapter provides a tutorial for the use of SLPs as a means of representing probability distributions over structured objects such as sequences, graphs, and plans.

11.1 Introduction

11.1.1 Logic Programs for Algorithms

Logic programs [5] provide a convenient way of describing computer algorithms in a compact and declarative fashion. A good example of this is the following Prolog [1] representation of the quick-sort algorithm.

```
quick_sort([],[]).
quick_sort([Head|Tail],Sorted) :-
        partition(Tail,Head,BeforeHead,AfterHead),
        quick_sort(BeforeHead,BeforeSorted).
        quick_sort(AfterHead,AfterSorted),
        append(BeforeSorted,[Head|AfterSorted],Sorted).
```

Here the key elements of the algorithm are captured in a few lines, showing the relationship between the various parts of the solution.

11.1.2 Logic Programs and Non-Determinism

An algorithm generally describes an entirely deterministic series of actions. However, apart from their use in describing algorithms logic programs can also describe processes involving non-deterministic choice. For instance, consider the following well-known logic program.

```
member(Element,[Element|_]).
member(Element,[_|List]) :-
        member(Element,List).
```

When provided with the goal *:- member(X,[b,a,c])*, a Prolog interpreter will give the following solutions.

```
X = b
X = a
X = c
```

Each of these three solutions is associated with one of the derivations of the given goal.

11.1.3 Probabilistic Non-Determinism

Consider the following non-deterministic logic program representation of the outcome of tossing a two-sided coin.

```
coin(head).
coin(tail).
```

This logic program can be interpreted as saying that when the coin is tossed it will either come up as heads or tails. However, the logic program does not state the frequency with which we can expect these two outcomes to occur. By associating probability labels with the clauses we get the following stochastic logic program (SLP) [9] representation of a fair coin (a coin with equal probability outcomes of heads and tails).

```
0.5: coin(head).
0.5: coin(tail).
```

Given the goal *:- coin(X)* we would now expect the outcomes $X = head$ and $X = tail$ to occur randomly with probability 0.5 in each case. Here we can view X as a random variable in the statistical sense.

11.2 Mixing Deterministic and Probabilistic Choice

We now consider two more complex representational problems. The first involves a simple game with probabilistic outcomes and the second a simplified version of the famous casino blackjack game. The first game is described below.

11.2.1 Simple Game of Chance

The game involves a player and a banker. The player starts with a quantity of N counters and the banker with M counters. Until the player chooses to stop he does the following repeatedly.

1. The player pays an entrance fee (F counters) to the banker.
2. The player rolls a six-sided dice and gets the value D.
3. The banker rewards the player with D counters.

11.2.2 Representing the Game as an SLP

Below we show how this game can be represented as an SLP. The form of SLP used below is known as an *impure* SLP. An impure SLP is one in which not every definite clause has a probability label. Those without a probability label are treated as normal logic program clauses. Let us start with the unlabeled part of the program.

```
play(State) :-
        act(stop(State,State)).
play(State) :-
        act(pay_entrance(State,State1)),
        act(dice_reward(State1,State2)),
        play(State2).

act(X) :- X.
```

Here we see the general playing strategy for the game. Every action such as *pay_entrance* and *dice_reward* is conducted by the predicate *act*. Each such action transforms one state into another. Play proceeds by recursing via the second clause until the stop action is taken using the first clause.

11.2.3 Representing the Actions

Next we show the way in which actions are represented.

```
stop([Player,Banker],_) :-
        Player1 is Player,
        Banker1 is Banker,
        write('Player = '), write(Player1), nl
        write('Banker = '), write(Banker1), nl

pay_entrance([Player,Banker],[Player-4,Banker+4]).

dice_reward([Player,Banker],[Player+D,Banker-D]) :-
        roll_dice(D).
```

The stop action simply prints out the playing state, which is represented as a two-element list consisting of the number of counters held by the *Player* and *Banker* respectively.

The *pay_entrance* action reduces the player's counters by *4* and increases the banker's counters by *4*.

The *dice_reward* action increases the player's counters by the value D of the rolled dice and decreases the banker's counters by D.

11.2.4 Representing the Dice

The dice represent the only probabilistic element of the game. A fair dice is represented as follows.

```
1/6: roll_dice(1).
1/6: roll_dice(2).
1/6: roll_dice(3).
1/6: roll_dice(4).
1/6: roll_dice(5).
1/6: roll_dice(6).
```

11.2.5 Simplified Blackjack Game

The end of this section will be dedicated to a more extended representational problem which involves a version of the blackjack game. After describing the game, we will show how we can represent it using the SLP framework and why this framework is efficient and adapted for this problem. Finally we will see how we could modify this version and the effect of such modifications on the corresponding SLP representation.

11.2.6 Description and Specifications

Our blackjack game model is very close to the real blackjack game, as described in Wikipedia [14]. We consider this version as a simplification of the real game in the sense that it does not include bets and money. It involves only one player and the player does not have any strategy.

Let us now describe the specifications of the game. Blackjack hands are scored by their point total. The hand with the highest total wins as long as it does not go over 21, which is called a “bust.” Cards 2 through 10 are worth their face value, and face cards (Jack, Queen, King) are also worth 10. An ace counts as 11 unless it would bust a hand, in which case it counts as 1.

In our version there is only one player. His goal is to beat the dealer, by having the higher, unbusted hand. Note that if the player busts, he loses, even if the dealer also busts. If the player's and the dealer's hands have the same point value, this is known as a “push,” and neither player nor dealer wins the hand.

The dealer deals the cards, in our version from one deck of cards. The dealer gives two cards to the player and to himself.

A two-card hand of 21 (an ace plus a ten-value card) is called a “blackjack” or a “natural,” and is an automatic winner.

If the dealer has a blackjack and the player does not, the dealer wins automatically. If the player has a blackjack and the dealer does not, the player wins automatically. If the player and dealer both have blackjack, it is a tie (push). If neither side has a blackjack, in our version the strategy of the player is always to stand, then the dealer plays his hand. He must hit until he has at least 17, regardless of what the player has. The dealer may hit until he has a maximum of five cards in his hands.

The parameters of the game that could be modified, defining another version of the game are:

- the number of decks of cards;
- the maximum number of cards in a hand;
- the strategy of the player;
- the number of players.

11.2.7 Prolog Implementation of the Game

We show here how we can represent the game in Prolog. Such an implementation could lead to the SLP representation for several reasons. First, since SLPs lift the concept of logic programs, representing the game in Prolog allows us to translate it in order to obtain an SLP representation of the game. Secondly, it is interesting to see the difference of expressivity between logic programs and SLPs. Finally, the Prolog implementation permits us to experimentally verify the correctness of our representation.

Let us present the entry clause of the program.

```
game(Result,PScore,PHand,DScore,DHand) :-
  State0 = [[],[],[]],
  act(first_2_cards(State0,State1)),
  act(rest_of_game(State1,State2)),
  end_of_game(State2,Result,PScore,PHand,DScore,DHand).

act(X) :- X.
```

The general playing strategy has the same structure as for the simple game of chance described above. Indeed, every action such as `first_2_cards` and `rest_of_game` is conducted by the predicate `act`. Each such action transforms one state into another. The predicate `end_of_game` does not represent an action but calculates, given the final state of the game, the result, returning also the scores and the hands of the player and the dealer as its last arguments.

A playing state is represented as a list of three lists. The first list represents the player hand, the second the dealer hand, and the third all the cards already dealt.

The rest of the program defines each of the predicates which are mentioned in the body of the `clause`. For instance, let us present the definition of the predicate `rest_of_game`.

```
rest_of_game(State,State2) :-
  act(p_turn(State,State1)),
  act(d_turn(State1,State2)).
```

We need to introduce two other predicates; `p_turn` and `d_turn`. For instance, `d_turn` represents the dealer's turn after he has received his first two cards. In this phase he asks for extra cards until he stands. This corresponds to the following two clauses.

```
d_turn(State,State) :-
  d_stands(State).

d_turn(State,State2) :-
  \+ d_stands(State),
  act(d_deal_card(State,State1)),
  act(d_turn(State1,State2)).
```

The predicate `d_deal_card` represents the action of dealing a card to the dealer. Therefore, it requires taking a card from the deck of cards. Taking a card is represented by the following clause.

```
pick_card(Cards,Card) :-
  random_card(Card),
  non_member(Card,Cards).,
```

where

```
random_card((C,V)) :-
  repeat,
  random(1,5,C),
  random(1,14,V).
```

`random` is a build-in Prolog predicate which simulates the choice of a number between two bounds.

11.2.8 Representing the Blackjack Game as an SLP

SLP is the statistical relational learning (SRL) framework that is arguably the closest to logic programs in terms of declarativeness. Therefore SLP is the most expressive framework to translate the blackjack game into. Indeed, there are two type of clauses in the Prolog program that require two different types of treatment.

- The Prolog clauses without any random aspect are not modified in the SLP representation. Obviously we do not restrict ourselves to the notion of pure SLPs but instead we allow for impure SLP representations.
- The random aspects of the program are transformed in labeled clauses. However, taking a card from a deck of cards is the only probabilistic element of the game. Therefore, the Prolog implementation and the SLP representation of the blackjack game are virtually identical. The sole use of the predicate *random* is replaced by several labeled clauses expressing that taking a card from a deck is a random action. Let us show how the action of taking a card from the deck is translated:

Compared to the Prolog implementation of this action described above, the *random* predicates have to be replaced in the SLP representation by labeled clauses. The clause *choose_color* determines the color of the card and the clause *choose_value* determines the value of the card.

```
pick_card(Cards,Card) :-
  random_card(Card),
  non_member(Card,Cards).

random_card((C,V)) :-
  repeat,
  choose_color(C),
  choose_value(V).

0.25: choose_color(1).
0.25: choose_color(2).
0.25: choose_color(3).
0.25: choose_color(4).


0.07692308: choose_value(1).
0.07692308: choose_value(2).
0.07692308: choose_value(3).
0.07692308: choose_value(4).
0.07692308: choose_value(5).
0.07692308: choose_value(6).
0.07692308: choose_value(7).
0.07692308: choose_value(8).
0.07692308: choose_value(9).
0.07692308: choose_value(10).
0.07692308: choose_value(11).
0.07692308: choose_value(12).
0.07692308: choose_value(13).
```

Thus the SLP representation is almost as expressive as the Prolog program. Since SLPs are logically oriented, it is relatively easy to understand the rules of the game given the SLP representation.

11.2.9 Effect of Several Game Modifications

Let us now present how a modification in the parameters of the game description would affect this model.

- If we added other decks of cards, we would have to add an argument to the description of a card. A card would be defined as C=(Value,Color,Number_of_Deck). We would have to modify this in the relevant clauses but the number of these clauses is limited. We would also have to add a predicate `choose_deck` defined like `choose_color` and `choose_value`.
- If we allowed for more cards in a hand, we would only have to replace 5 by the new number in the definition of `d_stands`.
- If we wanted to assign a cleverer game strategy for the player, we would only have to modify the `p_stands` predicate.
- We would have to make more important changes if we wanted to change the number of players. Indeed, we would have to add equivalent predicates for all the predicates that model the actions of the players.

Thanks to its great expressivity and compactness, the SLP representation would not be modified much when changing the parameters of the game compared to other frameworks.

11.3 Stochastic Grammars

The initial inspiration for SLPs in [9] was the idea of lifting stochastic grammars to the expressive level of logic programs. In this section we show the relationship between stochastic grammars and SLPs.

11.3.1 Stochastic Automata

Stochastic automata, otherwise called hidden Markov models [11], have found many applications in speech recognition. An example is shown in figure 11.1. Stochastic automata are defined by a 5-tuple $A = \langle Q, \Sigma, q_0, F, \delta \rangle$. Q is a set of states. Σ is an alphabet of symbols. q_0 is the initial state and $F \subseteq Q$ ($F = \{q_2\}$ in figure 11.1) is the set of final states. $\delta : (Q \setminus F) \times \Sigma \rightarrow Q \times [0, 1]$ is a stochastic transition function which associates probabilities with labeled transitions between states. The sum of probabilities associated with transitions from any state $q \in (Q \setminus F)$ is 1.

In the following λ represents the empty string. The transition function $\delta^* : (Q \setminus F) \times \Sigma^* \rightarrow Q \times [0, 1]$ is defined as follows. $\delta^*(q, \lambda) = \langle q, 1 \rangle$. $\delta^*(q, au) = \langle q_{au}, p_a p_u \rangle$

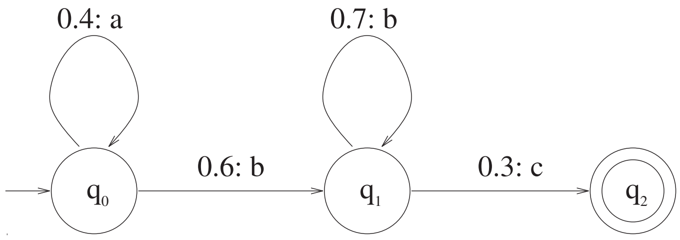

Figure 11.1 Stochastic automaton.

if and only if $\delta(q,a) = \langle q_a, p_a \rangle$ and $\delta^*(q_a,u) = \langle q_{au}, p_u \rangle$. The probability of u being accepted from state q in A is defined as follows. $Pr(u|q,A) = p$ if $\delta^*(q,u) = \langle q', p \rangle$ and $q' \in F$. $Pr(u|q,A) = 0$ otherwise.

Theorem 11.1
Probability of a string being accepted from a particular state Let $A = \langle Q, \Sigma, q_0, F, \delta \rangle$ be a stochastic automaton. For any $q \in Q$ the following holds.

$$\sum_{u \in \Sigma^*} Pr(u|q,A) = 1.$$

Proof Suppose the theorem is false. Either $q \in F$ or $q \notin F$. Suppose $q \in F$. Then by the definition of stochastic automata q has no outgoing transitions. Therefore by definition $Pr(u|q,A)$ is 1 for $u = \lambda$ and 0 otherwise, which is in accordance with the theorem. Therefore suppose $q \notin F$. Suppose in state q the transitions are $\delta(q,a_1) = \langle q, p_1 \rangle, \ldots, \delta(q,a_n) = \langle q, p_n \rangle$. Then each string u is accepted in proportions $p_1, \ldots, p_n$. according to its first symbol. That is to say, $\sum_{u \in \Sigma^*} Pr(u|q,A) = p_1 + .. + p_n$. But according to the definition of δ, $p_1 + \ldots + p_n = 1$, which means $\sum_{u \in \Sigma^*} Pr(u|q,A) = 1$. This contradicts the assumption and completes the proof. □

If the probability of u being accepted by A is now defined as $Pr(u|A) = Pr(u|q_0,A)$, then the following corollary shows that A defines a probability distribution over Σ^*.

Corollary 11.2
Stochastic automata represent probability distributions Given stochastic automaton A,

$$\sum_{u \in \Sigma^*} Pr(u|A) = 1.$$

Proof Special case of theorem 11.1 when $q = q_0$. □

The following example illustrates the calculation of probabilities of strings.

Example 11.1
Probabilities associated with strings For the automaton A in figure 11.1 we have $Pr(abbc|A) = 0.4 \times 0.6 \times 0.7 \times 0.3 = 0.0504$. $Pr(abac|A) = 0$.

$$\begin{aligned}
0.4 &: q_0 \rightarrow aq_0 \\
0.6 &: q_0 \rightarrow bq_1 \\
\\
0.7 &: q_1 \rightarrow bq_1 \\
0.3 &: q_1 \rightarrow cq_2 \\
\\
1.0 &: q_2 \rightarrow \lambda
\end{aligned}$$

Figure 11.2 Labelled production rule representation of stochastic automaton.

A can also be viewed as expressing a probability distribution over the language $L(A) = \{u : \delta^*(q_0, u) = \langle q, p\rangle \text{ and } q \in F\}$. The following theorem places bounds on the probability of individual strings in $L(A)$. The notation $|u|$ is used to express the length of string u.

Theorem 11.3
Probability bounds. Let $A = \langle Q, \Sigma, q_0, F, \delta\rangle$ be a stochastic automaton and let p_{min}, p_{max} be respectively the minimum and maximum probabilities of any transition in A. Let $u \in L(A)$ be a string.

$$p_{min}^{|u|} \leq Pr(u|A) \leq p_{max}^{|u|}.$$

Proof $Pr(u|A) = \prod_{i=1}^{|u|} p_i$, where p_i is the probability associated with the ith transition in A accepting u. Clearly each p_i is bounded below by p_{min} and above by p_{max}, and thus $p_{min}^{|u|} \leq Pr(u|A) \leq p_{max}^{|u|}$. $\square$

This theorem shows that (a) all strings in $L(A)$ have nonzero probability and (b) stochastic automata express probability distributions that decrease exponentially in the length of strings in $L(A)$.

11.3.2 Labeled Productions

Stochastic automata can be equivalently represented as a set of labeled production rules. Each state in the automaton is represented by a nonterminal symbol and each δ transition $\langle q, a\rangle \rightarrow \langle q', p\rangle$ is represented by a production rule of the form $p : q \rightarrow aq'$. Figure 11.2 is the set of labeled production rules corresponding to the stochastic automaton of figure 11.1. Strings can now be generated from this stochastic grammar by starting with the string q_0 and progressively choosing productions to rewrite the leftmost nonterminal randomly in proportion to their probability labels. The process terminates once the string contains no nonterminals. The probability of the generated string is the product of the labels of rewrite rules used.

$$\begin{aligned}0.5 &: S \rightarrow \lambda \\ 0.5 &: S \rightarrow aSb\end{aligned}$$

Figure 11.3 Stochastic context-free grammar

11.3.3 Stochastic Context-free Grammars

Stochastic context-free grammars [4] can be treated in the same way as the labeled productions of the last section. However, the following differences exist between the regular and context-free cases.

- To allow for the expression of context-free grammars the left-hand sides of the production rules are allowed to consist of arbitrary strings of terminals and nonterminals.
- Since context-free grammars can have more than one derivation of a particular string u, the probability of u is the sum of the probabilities of the individual derivations of u.
- The analogue of Theorem 11.3 holds only in relation to the length of the derivation, not the length of the generated string.

Example 11.2
The language a^nb^n Figure 11.3 shows a stochastic context-free grammar G expressed over the language a^nb^n. The probabilities of generated strings are as follows. $Pr(\lambda|G) = 0.5$, $Pr(ab|G) = 0.25$, $Pr(aabb|G) = 0.125$.

11.4 Stochastic Logic Programs

Every context-free grammar can be expressed as a definite clause grammar [2]. For this reason the generalization of stochastic context-free grammars to SLPs is reasonably straightforward. First, a definite clause C is defined in the standard way as having the following form.

$$A \leftarrow B_1, \ldots, B_n,$$

where the atom A is the head of the clause and $B_1, \ldots, B_n$ is the body of the clause. C is said to be range-restricted if and only if every variable in the head of C is found in the body of C. A *stochastic clause* is a pair $p : C$ where p is in the interval $[0, 1]$ and C is a range-restricted clause. A set of stochastic clauses P is called a *stochastic logic program* if and only if for each predicate symbol q in P the probability labels for all clauses with q in the head sum to 1.

$$0.5 : nate(0) \leftarrow$$
$$0.5 : nate(s(N)) \leftarrow nate(N)$$

Figure 11.4 Exponential distribution over natural numbers.

11.4.1 Stochastic SLD Refutations

For SLPs the stochastic refutation of a goal is analogous to the stochastic generation of a string from a set of labeled production rules. Suppose that P is an SLP. Then $n(P)$ will be used to express the logic program formed by dropping all the probability labels from clauses in P. A stochastic SLD procedure will be used to define a probability distribution over the Herbrand base of $n(P)$. The stochastic SLD derivation of atom a is as follows. Suppose $\leftarrow g$ is a unit goal with the same predicate symbol as a, no function symbols, and distinct variables. Next suppose that there exists an SLD refutation of $\leftarrow g$ with answer substitution θ such that $g\theta = a$. Since all clauses in $n(P)$ are range-restricted, θ is necessarily a ground substitution. The probability of each clause selection in the refutation is as follows. Suppose the first atom in the subgoal $\leftarrow g'$ can unify with the heads of stochastic clauses $p_1 : C_1, \ldots, p_n : C_n$, and stochastic clause $p_i : C_i$ is chosen in the refutation. Then the probability of this choice is $\frac{p_i}{p_1+\ldots+p_n}$. The probability of the derivation of a is the product of the probability of the choices in the refutation. As with stochastic context-free grammars, the probability of a is then the sum of the probabilities of the derivations of a.

This stochastic SLD strategy corresponds to a distributional semantics [13] for P. That is, each atom a in the success set of $n(P)$ is assigned a nonzero probability (due to the completeness of SLD derivation). For each predicate symbol q the probabilities of atoms in the success set of $n(P)$ corresponding to q sum to 1 (the proof of this is analogous to theorem 11.1).

11.4.2 Polynomial Distributions

It is reasonable to ask whether theorem 11.3 extends in some form to SLPs. The distributions described in [10] include both those that decay exponentially over the length of formulae and those that decay polynomially. SLPs can easily be used to describe an exponential decay distribution over the natural numbers as follows.

Example 11.3
Exponential distribution Figure 11.4 shows a recursive SLP P which describes an exponential distribution over the natural numbers expressed in Peano arithmetic form. The probabilities of atoms are as follows. $Pr(nate(0)|P) = 0.5$, $Pr(nate(s(0))|P) = 0.25$, and $Pr(nate(s(s(0)))|P) = 0.125$. In general, $Pr(nate(N)|P) = 2^{-N-1}$.

$$1.0 : natp(N) \leftarrow nate(U), bin(U, N)$$

$$0.5 : bin(0, [1]) \leftarrow$$
$$0.5 : bin(s(U), [C|N]) \leftarrow coin(C), bin(U, N)$$

Figure 11.5 Polynomial distribution over natural numbers.

However, SLPs can also be used to define a polynomially decaying distribution over the natural numbers as follows.

Example 11.4

Polynomial distribution Figure 11.5 shows a recursive SLP P which describes a polynomial distribution over the natural numbers expressed in reverse binary form. Numbers are constructed by first choosing the length of the binary representation and then filling out the binary expression by repeated tossing of a fair coin. Since the probability of choosing a number N of length $log_2(N)$ is roughly $2^{-log_2(N)}$ and there are $2^{log_2(N)}$ such numbers, each with equal probability, $Pr(natp(N)|P) \approx 2^{-2log_2(N)} = N^{-2}$.

11.5 Learning Techniques

We will now briefly introduce the different existing learning techniques for SLP. We will begin with the description of data used for learning. We will then focus on studying the parameter estimation techniques and finally the structure learning, after having defined these notions.

11.5.1 Data Used

For SLP, as for stochastic context-free grammars, the evidence used for learning is facts or even clauses .

11.5.2 Parameter Estimation

The aim of parameter estimation is, given a set of examples, to infer the values λ^* of the parameters λ (which represent the quantitative part of the model) that best justify the set of examples. We will focus on the maximum likelihood estimation (MLE) which tries to find $\lambda^* = argmax_\lambda P(E|L, \lambda)$. Yet we cannot calculate exactly the MLE when data is missing, so the expectation maximization (EM) algorithm is the most commonly used technique.

As described in [12], "EM assumes that the parameters have been initialized (e.g., at random) and then iteratively perform 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 using (weighted) frequency counting."

For SLP, one uses the *failure-adjusted maximization* (FAM) algorithm introduced by Cussens [3]. One has to learn the parameters thanks to the evidence, whereas the logical part of the SLP is given. The examples consist of atoms for a predicate p and are logically entailed by the SLP, since they are generated from the target SLP. In order to estimate the parameters, SLD trees are computed for each example. Each path from root to leaf is considered as one of the possible completions. Then, one weights the above completions with the product of probabilities associated with clauses that are used in the completions. Eventually, one obtains the improved estimates for each clause "by dividing the clause's expected counts by the sum of the expected counts of clauses for the same predicate."

11.5.3 Structure Learning

Given a set of examples E and a language bias B, which determines the set of possible hypotheses, one searches for a hypothesis $H^* \in B$ such that

- 1. H^* logically covers the examples E, i.e., cover(H^*, E), and
- 2. the hypothesis H^* is optimal w.r.t. some scoring function scores, i.e., $H^* = argmax_{H \in B} = score(H, E)$.

The hypotheses are of the form (L, λ) where L is the logical part and λ the vector of parameters values defined in section 11.5.2.

The existing approaches use a heuristic search through the space of hypothesis. Hill-climbing or beam-search are typical methods that are applied until the candidate hypothesis satisfies the two conditions defined above. One applies refinement operators during the steps in the search space.

For SLPs, as described in [12], structure learning "involves applying a refinement operator at the *theory* level (i.e. considering multiple predicates) under entailment." It is theory revision in inductive logic programming. This problem being known as very hard, the only approaches have been restricted to learning missing clauses for a single predicate. Muggleton [7], introduced a two-phase approach that separates the structure learning aspects from the parameter estimation phase. In a more recent approach, Muggleton [8] presents an initial attempt to integrate both phases for single predicate learning.

11.6 Conclusion

Stochastic logic programs provide a simple scheme for representing probability distributions over structured objects. This chapter provides a tutorial for the use of SLPs as a means of representing probability distributions over structured objects such as sequences, graphs, and plans.

SLPs were initially applied to the problem of learning from positive examples only [6]. This required the implementation of the following function which defines the generality of an hypothesis.

$$g(H) = \sum_{x \in H} D_X(x).$$

The generality is thus the sum of the probability of all instances of hypothesis H. Clearly such a sum can be infinite. However, if a large enough sample is generated from D_X (implemented as an SLP), then the proportion of the sample entailed by H gives a good approximation of $g(H)$.

Acknowledgments

Our thanks for useful discussions on the topics in this chapter with James Cussens, Kristian Kersting, Jianzhong Chen, and Hiroaki Watanabe. This work was supported by the Esprit IST project "Application of Probabilistic Inductive Logic Programming II (APRIL II)" and the DTI Beacon project, "Metalog - Integrated Machine Learning of Metabolic Networks Applied to Predictive Toxicology."

References

[1] I. Bratko. *Prolog for Artificial Intelligence.* Addison-Wesley, London, 1986.

[2] W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* Springer-Verlag, Berlin, 1981.

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

[4] K. Lari and S. J. Young. The estimation of stochastic context-free grammars using the inside-outside algorithm. *Computer Speech and Language*, 4:35–56, 1990.

[5] J.W. Lloyd. *Foundations of Logic Programming, 2nd edition.* Springer-Verlag, Berlin, 1987.

[6] S.H. Muggleton. Learning from positive data. In *Proceedings of the International Conference on Inductive Logic Programming*, 1997.

[7] S.H. Muggleton. Learning stochastic logic programs. *Electronic Transactions in Artificial Intelligence*, 4(041), 2000.

[8] S.H. Muggleton. Learning structure and parameters of stochastic logic programs. *Electronic Transactions in Artificial Intelligence*, 6, 2002.

[9] S.H. Muggleton. Stochastic logic programs. In L. de Raedt, editor, *Advances in Inductive Logic Programming*, pages 254–264. IOS Press, Amsterdam, 1996. URL `http://www.doc.ic.ac.uk/s̃hm/Papers/slp.pdf`.

[10] S.H. Muggleton and C.D. Page. A learnability model for universal representations. Technical Report PRG-TR-3-94, Oxford University Computing Laboratory, Oxford, UK, 1994.

[11] L.R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[12] L. De Raedt and K. Kersting. Probabilistic logic learning. *ACM-SIGKDD Explorations*, 5(1):31–48, 2003.

[13] T. Sato. A statistical learning method for logic programs with distributional semantics. In *Proceedings of the Twelth International conference on logic programming*, pages 715–729, 1995.

[14] Wikipedia, 2006. Wikipedia's page on the Blackjack game, http://en.wikipedia.org/wiki/Blackjack.