Imperial College London
Department of Computing

ProbPoly - A Probabilistic Inductive Logic Programming Framework with Application in Learning Requirements

by

Călin-Rareş Turliuc

Submitted in partial fulfilment of the requirements for the MSc Degree in Computing Science / Artificial Intelligence of Imperial College London

September 2011

Acknowledgements

I would like to thank my supervisor Dr. Alessandra Russo for her continuous assistance and support, for long, challenging and inspiring discussions, and for invaluable feedback and evaluation of my work.

I would also like to thank my second marker Dr. Krysia Broda for an excellent collaboration for my ISO project and incommensurable help and advice.

I am thankful for the edifying meetings and unconditional aid of Dalal Alrajeh and Domenico Corapi.

On a personal note, I am grateful for the discussions with Alexandru Marinescu and his suggestions, and immensely indebted for my parents' love and financial support.

Abstract

The focus of this project has been on developing a novel Probabilistic Inductive Logic Programming Framework (PILP) called ProbPoly. The methodology is based on the semantics of Stochastic Logic Programs (SLPs), and by using probabilistic examples and an adequate score function to learn probabilities of clauses. Many limitations of existing learning methods for SLPs are overcome, such as non-recursiveness. We also offer a natural extension for incorporating negation as failure, as well as for learning multiple clauses. Our representation of predicted probabilities of examples is a (multivariate) polynomial, whose constrained minimization ensures that the predicted probabilities are close to the observed ones. The minimization task is performed using specific mathematical techniques of relaxation, and in case we can't ensure a global optimum, we use a state-of-the-art metaheuristic – particle swarm optimization.

PILP has been successfully applied in numerous areas of research, such as bioinformatics and natural language processing. We aim to apply it in a software engineering context, with the purpose of learning requirements. We review related systems, and show how we can use ProbPoly in conjunction with the PRISM probabilistic model checker to learn probabilities of a simple discrete time Markov chain, such that the new model satisfies a list of properties. To our knowledge, an experiment of learning requirements involving PILP and probabilistic model checking has never been done before.

Our conclusions are that ProbPoly is a promising idea for PILP, with a well founded theoretical background, and spanning numerous directions for improvement, both in theoretical and technical aspects.

Contents

Chapter 1

Introduction

Probabilistic Inductive Logic Programming (PILP) is a prominent area of research which aims to combine the expressiveness of learning clauses in a first-order logic language with the uncertainty of real-world data, quantified by probabilities. PILP has been successfully used in bioinformatics and natural language processing, and still presents many challenges for researchers interested in probabilities, logic and machine learning.

Probabilities have also been introduced in the verification of systems. The development of probabilistic model checkers, which verify probabilistic properties for probabilistic models such as Markov chains, has opened the possibility of designing new methods which learn requirements in a probabilistic setting.

The main contribution of our thesis is the creation of an original framework for PILP. We combine elements from Stochastic Logic Program (SLP) learning, which has been studied only for non-recursive programs and non-probabilistic positive examples, and learners based on distribution semantics, which use probabilistic examples. The aim of such learners is to learn probabilities and/or rules which satisfy certain criteria defined using the probabilities in the background knowledge and of the examples.

The merits of our framework are that it allows learning on SLPs with probabilistic examples with recursive programs, which is a significant improvement. We further extend our methodology to include negation as failure, as well as learning probabilities for multiple clauses simultaneously. We also reason about negative examples in the context of SLPs and about various quantitative measures used in information retrieval - true/false positives/negatives, and adapted to SLPs.

The main challenge of our approach for learning probabilities will be the constrained minimization of a multivariate polynomial, a problem overcome by using two state-of-the-art methods. In the case of non-recursive programs or when learning a single clause for a recursive program, we find the probabilities using the analytical method of gradient descent. In the multivariate case, initially we try to find a global minimum using a tool called GloptiPoly, designed specifically for optimization of polynomial functions. If we don't succeed, we fall back on the clever metaheuristic of Particle Swarm Optimization, which is believed to converge faster than other methods of evolutionary computation.

In the second part of our thesis we analyse the possibility of using PILP in conjunction with a probabilistic model checker in order to learn new probabilities of an initial model, such that the properties which are not satisfied in the initial are valid in the new model. An experiment is carried out on a simple model and proves successful.

Finally we identify the possible directions of future research, which include extending the learning of SLPs to multiple predicates, learning with proofs of infinite length, and combining our method with an ILP system to in order to interleave the process of learning the logical clauses with that of learning their associated probabilities.

Chapter 2

Background

This chapter will introduce elementary concepts from different areas of mathematics, logic and computer science, which are necessary for the understanding of the rest of the thesis. Since the main contribution is a probabilistic inductive logic programming framework, we begin our discussion with the topics of logic and logic programming in Section 2.1, and learning using logic: inductive logic programming and probabilistic inductive logic programming in Section 2.2. As we shall see, probabilistic inductive logic programming is a paradigm in which the definition of probabilities with respect to logic programming has a crucial impact. We treat two of perhaps the most popular views on semantics of probabilities: distribution semantics in Section 2.3 and stochastic logic programs in Section 2.4. We will adopt the latter in our novel framework.

Finally, our framework relies on the manipulation and, most importantly, the constrained optimization of (multivariate) polynomials, so we introduce relevant aspects of such fundamental mathematical expressions in Section 2.5.

2.1 Logic and Logic Programming

We assume familiarity with the syntax and semantics of propositional and first-order logic, unification and resolution. We also assume knowledge about *Linear Temporal Logic* (LTL), which extends first-order logic with operators for temporal sequences; however, we will explain intuitively the required operators when they are introduced. We also assume knowledge of the Prolog notation, which we will use throughout the thesis (atoms and predicates start with lower-case letters, variables start with upper-case letters, we use ":-" with the same meaning as logical "$\leftarrow$"). In the remainder of the section we introduce in a semi-formal manner a minimal amount of concepts from first-order logic and logic programming. The presentation is inspired by the first part (especially chapters 2 and 7) of [Nienhuys-Cheng and Wolf, 1997], which deals with these matters formally and at length.

A *predicate* is also called an *atom*, or, when referring to logic programming, a *goal*, and can be assigned truth values (*true* or *false*).

A *literal* is an atom or a negated atom. Usual operations on literals include *conjunction* ($\wedge$), *disjunction* ($\vee$), *negation* ($\neg$), *implication* ($\rightarrow$, and its inverse, the *conditional* $\leftarrow$) and *equivalence* ($\leftrightarrow$).

A *clause* is a disjunction of literals ($l_1 \vee l_2 ... \vee l_n$). An *empty clause* is denoted by [ ] or $\Box$, and its truth value is $false$.

A *Horn clause* C is a clause with at most one positive literal. It will be written in the form $q \leftarrow l_1, ..., l_n$ with $n \geq 0$, where q is an atom and all l_i with $1 \leq i \leq n$ are atomic literals, and q, or C^+, is called the *head* and $l_1, \ldots, l_n$, or C^-, is called the *body* of the clause. Clause bodies are understood to be conjunctions of literals.

If all l_i are atoms a clause is called *definite*. If we allow negation, i.e. $l_i = not\ A$, where A is an atom, a clause is called *normal*. A *denial clause* is a clause of the form $\leftarrow l_1, \ldots, l_n$. The number of literals in the body of a clause is called the *length* of the clause.

A *(normal logic) program* P is a finite set of (normal) clauses and a *definite logic program* is a finite set of definite clauses.

An *interpretation* maps predicates to *true* or *false*. We will usually identify an interpretation with the set of predicates which it maps to true (the rest are implicitly assumed to be false). An interpretation is extended to literals, clauses and programs in the usual way.

A *model* of a clause C is an interpretation I which maps C to true (in symbols: $I \Vdash C$). A model of a program P is an interpretation which maps every clause in P to true.

Let Σ be a set of Horn clauses, and C be a Horn clause. An *SLD-derivation* of length k of C from Σ is a finite sequence of Horn clauses $R_0, \ldots, R_k$, and $R_k = C$, such that $R_0 \in \Sigma$ and each R_i, $i = \overline{1,k}$, is a binary resolvent of R_{i-1} and a definite program clause $C_i \in \Sigma$, using the head of C_i and a *selected atom* in the body of R_{i-1} as the literals resolved upon.

An SLD-derivation of the empty clause [ ] from Σ is called an *SLD-refutation* of Σ. A *goal* is a clause.

Let P be a definite program, and G a definite clause. An *SLD-tree* for $P \cup \{G\}$ is a tree satisfying the following:

- Each node of the tree is a (possibly empty) definite goal.
- The root node is G.
- Let $N = \leftarrow A_1, \ldots, A_s, \ldots, A_k$, $k \geq 1$, be a node in the tree, with A_s as selected atom. Then, for each clause C in P such that A_s and (a variant of) C^+ are unifiable, the node N has exactly one resolvent of N and C, B_s, as a child. The node has no other children for the same A_s and C.
- Nodes which are the empty clause [ ] have no children.

Negation as Failure (NaF) is a method that allows us to introduce negated atoms, and thus working with normal logic programs rather than definite logic programs. The essential idea is that for a definite goal G, we assume that $\neg G$ is proven if G *finitely fails*. Let P be a definite program, an SLD-tree for $P \cup \{G\}$ is called *finitely failed* if it is finite and contains no success branches.

SLD-resolution that takes into account negation as failure is called *SLDNF-resolution*, which is the standard resolution technique used in logic programming. However, we will leave out all the technical details about SLDNF-trees and SLDNF-resolution, because they are beyond the scope of this thesis.

2.2 Inductive Logic Programming and Probabilistic ILP

Inductive Logic Programming (ILP) is a field related to Logic, Logic Programming and Machine Learning, which aims at learning a (normal logic) program, referred to as theory or hypothesis H, based on domain knowledge, encoded as *background knowledge*, *language bias*, which is essential to reduce the search space, especially when learning complex predicates, and a set of *positive* and *negative examples*, the quality of which is crucial to the success of the learning.

The usual setting of ILP, as defined in [Nienhuys-Cheng and Wolf, 1997], is: given a set of clauses B (background knowledge) and sets of clauses E^+ and E^- (positive and negative examples), find a theory (i.e. a set of clauses) H such that $H \cup B$ is correct with respect to E^+ and E^-, which we will denote by $H \cup B \models E$. A set of clauses P is correct with respect to positive examples E^+ if: $P \models e$, $\forall e \in E^+$. A set of clauses P is correct with respect to negative examples E^- if: $P \not\models e$, $\forall e \in E^-$. It can be proven that we can transform examples into ground examples, since we make the Closed World Assumption (CWA), that is, everything which is not asserted to be true by our program is implicitly false.

As an example, consider:

$B = \{ \quad parent(bob, alan), parent(mia, kate),$
$male(bob), male(alan),$
$female(mia), female(kate)\},$

and examples $E^+ = \{has\_daughter(mia)\}$ and $E^- = \{has\_daughter(bob)\}$.

Furthermore, assume $H = \{has\_daughter(X) \leftarrow parent(X, Y), female(Y)\}$. This is the intuitive "human" definition of the *has_daughter* predicate, and in this case $H \cup B \models E$. Now let us make $H = \{has\_daughter(X) \leftarrow parent(X, Y)\}$, which would correspond to the "intuitive" *has_children* predicate, then $H \cup B \not\models E$, because $H \cup B \models has\_daughter(bob)$, and $has\_daughter(bob) \in E^-$. Finally, we may learn a theory such as $H = \{has\_daughter(X) \leftarrow parent(X, Y), female(X)\}$, which again corresponds to the "intuitive" *is_mother* predicate, but we have $H \cup B \models E$, so the learned theory is "correct". This can be seen as an ILP case of *overfitting*, and as mentioned earlier, we must have either an intuition about the predicate we want to learn, to generate appropriate examples, or have an oracle generate a large number of examples, to precisely guide the search towards the correct theory.

We also mention the task of abductive logic programming ([Kakas et al., 1993]). An ALP task, as defined in [Corapi et al., 2010], is based on $\langle g, T, A, I\rangle$, where g is a ground goal, T is a normal logic program, A is a set of ground facts called *abducibles*, and I is a set of denial clauses called *integrity constraints*. The output of the abductive procedure is a subset of A, called abductive solution – Δ, such that $T \cup \Delta$ is consistent, $T \cup \Delta \models g$ and $T \cup \Delta \models I$.

Probabilistic Inductive Logic Programming (PILP) is a branch of ILP which aims at combining the expressiveness of inductive logic learning with probabilistic reasoning, motivated by the uncertainty inherent in data. An excellent review can be found in [Raedt and Thon, 2010], which distinguishes between learning from entailment, interpretations and proofs, and proposes an adaptation of FOIL [Quinlan and Cameron-Jones, 1993] for probabilistic learning from entailment. However, unlike ILP, there is no general consensus as to what the task of PILP should be. This is due to the various semantics of probabilities in logic programming. We will overview two perspectives on the semantics of probabilities in Sections 2.3 (Distribution Semantics) and 2.4 (Stochastic Logic Programs Semantics).

2.3 Distribution Semantics

It is fundamental for PILP to have a theoretically founded semantics. Sato's distribution semantics, introduced in [Sato, 1995], assigns a probability for each formula over an infinite Herbrand universe, such that the *probability axioms* of Kolmogorov hold. The three axioms are:

1. Any probability is a non-negative real number.
2. The probability of an elementary event to occur over a sample space is 1. Intuitively, this means that we take into consideration all possible events in the sample space.
3. In the case of pairwise disjoint events, the probability of all the events is the sum of the probabilities of the individual events.

Sato considers definite clause programs DB of the form: $DB = F \cup R$, where F is the set of facts, and R is the set of rules. The key idea is to initially define a probability distribution over F, P_F, and to extend it over the whole program DB, thus obtaining P_{DB}, in a process influenced by the set of rules R. DB is assumed to be ground, denumerably infinite, and satisfying the disjoint condition, i.e. no fact in F unifies with the head of a rule in R.

Assume that $A_1, A_2, \ldots, A_n$ are atoms in F, $P_F^{(n)}$ is the joint probability over the first n variables in F, i.e. $A_1, A_2, \ldots, A_n$, and that $x_1, x_2, \ldots, x_n$ are 0 or 1 (corresponding to false or true, i.e. $A_i = 1$ means that atom A_i is true in the current interpretation). The existence of P_F is proved by ensuring that the following conditions hold:

1. $0 \leq P_F^{(n)}(A_1 = x_1, A_2 = x_2, \ldots, A_n = x_n) \leq 1$
2. $\displaystyle\sum_{x_1, x_2, \ldots, x_n} P_F^{(n)}(A_1 = x_1, A_2 = x_2, \ldots, A_n = x_n) = 1$
3. $\displaystyle\sum_{x_{n+1}} P_F^{(n+1)}(A_1 = x_1, A_2 = x_2, \ldots, A_{n+1} = x_{n+1}) = P_F^{(n)}(A_1 = x_1, A_2 = x_2, \ldots, A_n = x_n)$

The first condition ensures that the values of the probabilities defined are valid. The second condition is motivated by the need of completeness of the sample space (in this case, the probabilities over all interpretations must sum up to 1). The third condition, called the *compatibility condition* defines an additive property of probabilities, and is related to the completeness problem (an interpretation must be either true or false, so marginalizing over these two possibilities must sum up to 1).

The probability P_{DB} is constructed from P_F by considering all the interpretations I for which, their least models ($LM(I)$) logically imply the atoms in F. P_{DB} is then defined as P_F over the set of such interpretations:

$[A_1^{x_1} \wedge A_2^{x_2} \wedge \cdots \wedge A_n^{x_n}]_F \overset{\text{def}}{=} \{I \mid LM(I) \models (A_1^{x_1} \wedge A_2^{x_2} \wedge \cdots \wedge A_n^{x_n})\}$, where $A_i^{x_i}$ is A if $x_i = 1$ or $\neg A$ if $x_i = 0$.

The formal definition of P_{DB} is:

$P_{DB}^{(n)}(A_1 = x_1, A_2 = x_2, \ldots, A_n = x_n) \overset{\text{def}}{=} P_F([A_1^{x_1} \wedge A_2^{x_2} \wedge \cdots \wedge A_n^{x_n}]_F)$

However, we don't know how to compute a probability over a set of interpretations. Proposition 2.1 from [Sato, 1995] provides an answer to this problem. Let:

$\varphi_{DB}(x_1, x_2, \ldots, x_n) =< y_1, y_2, \ldots, y_k >$ iff

$\forall I \ (I \models A_1, \ldots, A_n \rightarrow LM(I) \models B_1, \ldots, B_k)$

Then the whole distribution is:

$P_{DB}(A_1, \ldots, A_n, B_1, \ldots, B_k) =$

$$\begin{cases} P_F(A_1, \ldots, A_n) & \text{if } \varphi_{DB}(x_1, x_2, \ldots, x_n) =< y_1, y_2, \ldots, y_k > \\ 0 & \text{otherwise} \end{cases}$$

The intuition behind this definition is that the probabilities of the interpretations over DB should be in fact the probabilities of the interpretations over F, but taking care that the rest of the atoms in the interpretations over DB should be assigned the correct values (obtained by induction of these atoms using the interpretations over F and the rules R).

The distribution over the head atoms is computed by summing (or marginalizing) over the probabilities of the interpretations that are consistent with $B_1, \ldots, B_k$:

$$P_{DB}(B_1, \ldots, B_k) = \sum_{\varphi_{DB}(x_1, x_2, \ldots, x_n) = <y_1, y_2, \ldots, y_k>} P_F(A_1, \ldots, A_n)$$

The easiest way to understand distribution semantics is via an example.

Example 2.1. *Let $DB = F \cup R$, $F = \{X, Y\}$, and $R = \{ \ A \leftarrow X,$*
$B \leftarrow X,$
$B \leftarrow Y,$
$C \leftarrow X, Y\}$

Assume P_F is defined as:

$\langle x_X, x_Y \rangle$	$P_F(x_X, x_Y)$
$\langle 0, 0 \rangle$	0.2
$\langle 1, 0 \rangle$	0.4
$\langle 0, 1 \rangle$	0.1
$\langle 1, 1 \rangle$	0.3
others	0

Then, P_{DB} will be:

$\langle x_X, x_Y, x_A, x_B, x_C \rangle$	$P_{DB}(x_X, x_Y, x_A, x_B, x_C)$
$\langle 0, 0, 0, 0, 0 \rangle$	0.2
$\langle 1, 0, 1, 1, 0 \rangle$	0.4
$\langle 0, 1, 0, 1, 0 \rangle$	0.1
$\langle 1, 1, 1, 1, 1 \rangle$	0.3
others	0

Suppose we now want to compute $P_{DB}(A = 1)$. This reduces to a simple marginalization over $P_F(x_X, x_Y, x_A, x_B, x_C)$:

$P_{DB}(A = 1) = P_{DB}(1, 0, 1, 1, 0) + P_{DB}(1, 1, 1, 1, 1) = 0.7$

In the same way:

$P_{DB}(B = 1) = P_{DB}(1, 0, 1, 1, 0) + P_{DB}(0, 1, 0, 1, 0) + P_{DB}(1, 1, 1, 1, 1) = 0.8$

Finally, we mention a probabilistic logic programming language called PRogramming In Statistical Modelling (PRISM[1]) described in [Sato and Kameya, 2001]. In this language we can specify logic programs which simulate the behaviour of Turing Machines, Bayes Networks, Markov Chains (e.g. Hidden Markov Models (HMMs)) etc. The authors also design an Expectation Maximization (EM) algorithm to evaluate $P(A_1, ..., A_n)$ if we know $P(B_1, ..., B_k)$, and which subsumes the specific version of EM for different models.

2.4 Stochastic Logic Programs and Their Semantics

Stochastic Logic Programs are introduced in [Muggleton, 1996], as a way to represent Stochastic (or Probabilistic) Context Free Grammars (SCFG or PCFG) in logic programming, and are defined as a set of *stochastic clauses.* A stochastic clause $p : C$ is a pair of a probability $p \in [0, 1]$ and a *range-restricted* clause C. A clause is range-restricted if every variable in the head of C is found in the body. Moreover, the set of stochastic clauses must satisfy the property that for all clauses containing q as the predicate of the head, the probabilities of these clauses must sum up to 1. It is easy to transform any logic program consisting of only range-restricted clauses annotated with probabilities into a SLP by normalization.

A *Stochastic SLD (SSLD) refutation* consists of the SLD refutation of the logic program (ignoring probabilities). Then, the probability of a derivation of a goal g is defined as the product of probabilities on the branches of the SLD tree, and the probability of goal g is the sum of the probabilities of all the derivations of g. Formally, we have:

Definition 2.1 (The Probability of a Derivation of a Goal)**.** *The probability of a derivation (in a successful case called a* proof *or* refutation*) proof of a goal g (given background knowledge B and hypothesis H) is :*

$$P(g|B \cup H, proof) = \prod_{p:c \in proof} p$$

Definition 2.2 (The (Total) Probability of a Goal)**.** *The total probability of a goal g (given background knowledge B and hypothesis H) is :*

$$\begin{aligned} P(g|B \cup H) &= \sum_{proof \in SSLD(g, B \cup H)} P(g|B \cup H, proof) \\ &= \sum_{proof \in SSLD(g, B \cup H)} \prod_{p:c \in proof} p \end{aligned}$$

Consider the following example from [Chen et al., 2008]:

Example 2.2. *SSLD derivation of s(X).*

0.4 : *s(X) :- p(X), p(X).*
0.6 : *s(X) :- q(X).*

0.3 : *p(a).*
0.7 : *p(b).*

0.2 : *q(a).*
0.8 : *q(b).*

[1] We won't use the abbreviation due to a name clash with the probabilistic model checker.

We reproduce the SSLD derivation tree with annotated probabilities in Figure 2.1[2]:

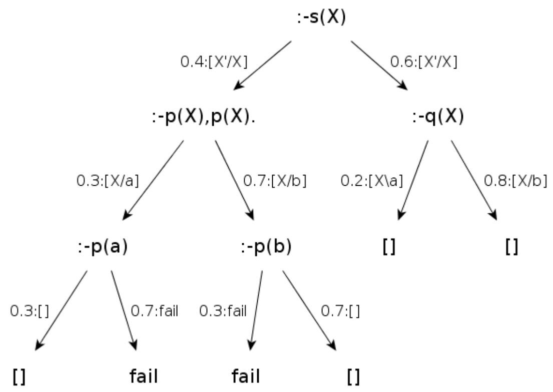


Figure 2.1: Example of SSLD derivation tree.

Notice that there are 6 derivations with 4 refutations and 2 fail-derivations. So, the (total) probability of goal $s(X)$:

$$\begin{aligned} P(s(X)) &= \sum_{r \in Refuations(s(X))} P(r) \\ &= 0.4 * 0.3 * 0.3 + 0.4 * 0.7 * 0.7 + 0.6 * 0.2 + 0.6 * 0.8 \\ &= 0.036 + 0.196 + 0.12 + 0.48 \\ &= 0.832 \end{aligned}$$

A nice property is that in the context of SLPs, the probability of all derivations, i.e. refutations as well as failed derivations, of any goal is 1, due to the fact that for any clauses with the same predicate in the head, the sum of the annotated probabilities must be 1.

We have mentioned that SLPs were designed for representing Probabilistic Context Free Grammars (PCFG) in a logic programming context. Let us define PCFGs formally and then provide a small example from Natural Language Processing (NLP), inspired by [Manning and Schütze, 1999].

Definition 2.3 (Probabilistic Context Free Grammar)**.** *A PCFG G consists of:*

- *A set of terminals,* $\{w^k\}$, $k = 1, \ldots, V$
- *A set of nonterminals,* $\{N^i\}$, $i = 1, \ldots, n$
- *A designated start symbol,* N^1
- *A set of rules,* $\{N^i \rightarrow \zeta^j\}$, *(where* ζ^j *is a sequence of nonterminals)*
- *A corresponding set of probabilities on rules such that:*
  $\forall i \sum_j P(N^i \rightarrow \zeta^j | N^i) = 1$

Consider the following PCFG[3]:

Example 2.3. *Simple PCFG example.*

$S \rightarrow NP\ VP$	1.0	$NP \rightarrow NP\ PP$	0.4
$PP \rightarrow P\ NP$	1.0	$NP \rightarrow astronomers$	0.1
$VP \rightarrow V\ NP$	0.7	$NP \rightarrow ears$	0.18
$VP \rightarrow VP\ PP$	0.3	$NP \rightarrow saw$	0.04
$P \rightarrow with$	1.0	$NP \rightarrow stars$	0.18
$V \rightarrow saw$	1.0	$NP \rightarrow telescopes$	0.1

[2] Figure 1(b) in [Chen et al., 2008].

[3] Table 11.2 from [Manning and Schütze, 1999].

We can directly model this PCFG into an SLP, by just considering probabilistic clauses of the form: $1 : s \leftarrow np, vp$ for the first rule in the PCFG, and so on. When considering (graphical) representations of PCFG we have the following result: for every PCFG there exists an equivalent Pushdown Automaton (PDA). However, it is convenient to represent a particular derivation of a string by the grammar using *parse trees*. Let us consider sentence S = [astronomers, saw, stars, with, ears], which can be parsed in two ways by the grammar in Example 2.3. The parse tree for the two derivations are shown in Figure 2.2.[4]

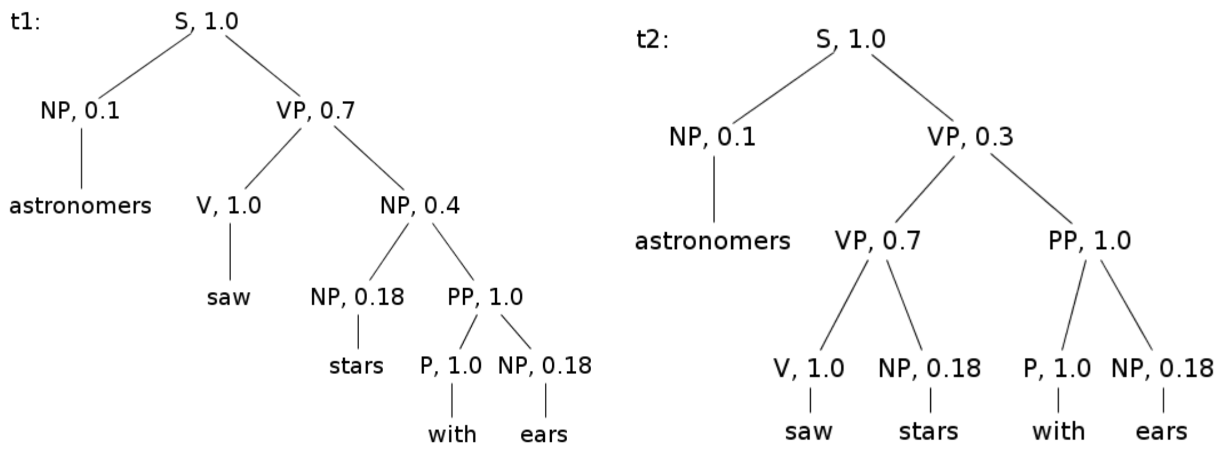


Figure 2.2: The parse trees for sentence [astronomers, saw, stars, with, ears] using the PCFG from Example 2.3.

The probability of a derivation is the product of all the probabilities in its parse tree, and is equivalent to the probability of a derivation of a goal in Definition 2.1, if we consider the goal to be the string that is parsed, and the logic program the representation of the grammar. In a similar manner, the probability of a string being derived by a PCFG is the sum over the probabilities of all derivations, which is again reflected in the (total) probability of a goal in Defintion 2.2, under the same circumstances as the ones mentioned above.

Note that this PCFG can in fact be represented in a logic programming language such as Prolog, as shown in Listing 2.1.[5] However, it is worthy to mention the problem of left recursion, which appears for rules of the form $R \rightarrow R...$. If we don't transform the parsing in an acceptable recursive paradigm (boundary condition and recursive case), we end up with an infinite loop in the execution.[6] By adequately querying the system after program compilation, we get the expected parse trees and the correct probabilities, as shown in Listing 2.2.

Listing 2.1: Prolog definition of PCFG in Example 2.3.

```
s(P0, s(NP,VP)) --> np(P1,NP), vp(P2,VP), { P0 is 1.0*P1*P2 }.

np(0.1, np(astronomers)) --> [astronomers].
np(P0, np(astronomers, Rest)) --> [astronomers], nptail(P1, Rest), {P0 is P1*0.1}.
np(0.18, np(ears)) --> [ears].
np(P0, np(np(ears), Rest)) --> [ears], nptail(P1, Rest), {P0 is P1*0.18}.
np(0.04, np(saw)) --> [saw].
np(P0, np(saw, Rest)) --> [saw], nptail(P1,  Rest), {P0 is P1*0.04}.
np(0.18, np(stars)) --> [stars].
np(P0, np(np(stars), Rest)) --> [stars], nptail(P1,  Rest), {P0 is P1*0.18}.
nptail(Acc, np(NP, PP)) -->  pp(P2, PP), np(Acc1, NP) , {Acc is 0.4*P2*Acc1}.
nptail(Acc,  PP) -->  pp(P2, PP) , {Acc is 0.4*P2}.
```

[4] Reproduced from figure 11.1 in [Manning and Schütze, 1999].

[5] The code is inspired by the simpler example at http://w3.msi.vxu.se/~nivre/teaching/statnlp/pdcg.html (accessed 01.09.2011).

[6] See http://www.cs.sfu.ca/~cameron/Teaching/383/DCG2.html(accessed 01.09.2011) for details and a more accessible example.

```
pp(P0, pp(P, NP)) --> p(P1, P), np(P2, NP), {P0 is 1.0*P1*P2}.

vp(P0, vp(V,NP)) --> v(P1,V), np(P2,NP), { P0 is 0.7*P1*P2 }.
vp(P0, vp(vp(V,NP), Rest)) --> v(P1,V), np(P2,NP), vtail(P3, Rest),{ P0 is 0.7*P1*P2*P3 }.
vtail(P0, PP) --> pp(P1, PP), {P0 is 0.3*P1}.
vtail(P0, VP) --> vp(P0, VP).

p(1.0, p(with)) --> [with].
v(1.0, v(saw)) --> [saw].
```

Listing 2.2: Results of the query to compute the parse trees and their probabilities.

```
?- s(P,T,[astronomers,saw,stars,with, ears],[]).
P = 0.0009072,
T = s(np(astronomers),vp(v(saw),np(np(stars),pp(p(with),np(ears))))) ? ;
P = 0.0006804,
T = s(np(astronomers),vp(vp(v(saw),np(stars)),pp(p(with),np(ears)))) ? ;
no
```

2.5 Polynomials

Polynomials are fundamental mathematical expressions which use constants and one (in which case we shall call them *univariate*) or more variables (in which case we shall call them *multivariate*). The constants, as well as the variables, are usually defined over the well known sets: $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$, or more generally $\mathbb{C}$.

For the moment, assume we discuss univariate polynomials over $\mathbb{C}$. A polynomial of this type is written as:

$p(x) = a_n x^n + a_{n-1}x^{n-1} + ... + a_1 x + a_0$, with $n \in \mathbb{N}$, $x, a_0, a_1, \ldots, a_n \in \mathbb{C}$ and $a_n \neq 0$.

n is the *degree* of the polynomial $p(x)$. If $n = 0$, then the polynomial is a constant function $p(x) = a_0$. A value x_0 is called a *root* of the polynomial $p(x)$ if $p(x_0) = 0$.

Property 2.1. *x_0 is a root of $p(x)$ of degree $n \geq 1$ i.f.f. there exists polynomial $q(x)$ of degree $n - 1$ and $p(x) = q(x)(x - x_0)$.*

The proof of the above property is straightforward:

$\Leftarrow$: If we have $p(x) = q(x)(x - x_0)$, then $p(x_0) = 0$, so, by definition x_0 is a root.

$\Rightarrow$: By the remainder theorem, we have: $p(x) = q(x)(x - x_0) + c$, $c \in \mathbb{C}$. However, x_0 is a root, so $p(x_0) = c = 0$.

Theorem 2.1 (Fundamental Theorem of Algebra)**.** *Let $p(x)$ be a polynomial of degree $n \geq 1$. Then, $p(x)$ always has a root $x_0 \in \mathbb{C}$.*

By using the fundamental theorem of algebra, and Property 2.1, we can inductively prove the following result:

Property 2.2. *Let $p(x)$ be a polynomial of degree $n \geq 1$. Then, $p(x)$ has at most n complex roots.*

The reason why we use "at most" and not "exactly" is due to multiple roots, e.g. $p(x) = (x-1)^2$, with root 1 being a multiple root (of order 2).

Among the many observations that can be made about polynomials, we shall mention mainly the following:

- if $p(x)$ has root $x_0 = a + ib$, then the complex conjugate of x_0, $\overline{x_0}$, is also a root. Informally, complex roots come in pairs.
- the derivative of a polynomial $p(x)$ of degree $n \geq 1$ is a polynomial $q(x)$ of degree $n - 1$.
- there exist efficient algorithms to determine the real roots of polynomials, among the most recent being [Rouillier and Zimmermann, 2004].

- we can find the minimum (or maximum) of a polynomial $p(x)$ in an interval $[a, b]$, by inspecting the real roots of the derivative in $[a, b]$, which are either local maxima or minima, and the values of $p(a)$ and $p(b)$, and keeping the minimum (or maximum).

We will also use multivariate polynomials, which have the general form:

$$p(x_1, ..., x_k) = \sum_{i=0}^{n} a_i x_1^{p(1,i)} x_2^{p(2,i)} \dots x_k^{p(k,i)}, \text{ with } n \in \mathbb{N},\ k \in \mathbb{N}^*,$$

$$a_i \in \mathbb{C}, \forall i = \overline{0, n},$$

$$x_j \in \mathbb{C}, \forall j = \overline{1, k}, \text{ and}$$

$$p(j, i) \in \mathbb{N},\ \forall i = \overline{0, n}, \forall j = \overline{1, k}$$

The degree of a multivariate polynomial is $\underset{i}{max} \sum_{j=1}^{k} p(j, i)$, and a root is a vector $[x_1^0, ..., x_k^0]$ such that $p(x_1^0, ..., x_k^0) = 0$.

The problem of finding the roots of a multivariate polynomial or of minimizing a multivariate polynomial (eventually, under certain constraints) is a difficult task in general. In some cases, the roots of multivariate polynomials can be found using Gröbner bases. The problem of solving sets of multivariate equations has been given attention due to the development of multivariate public key cryptography, which was motivated by the fact that quantum computers can perform integer factorization in polynomial time, and potentially break (univariate) public key cryptosystems such as the commonly used RSA.[7]

To minimize a multivariate polynomial under constraints, we use a special numerical method, implemented in MatLab, called GloptiPoly ([Henrion and Lasserre, 2002], [Henrion et al., 2007]). However, since it is not always reliable, and it is scalable only up to a certain extent, we can also treat this problem as a general function optimization problem, and use metaheuristics such as simulated annealing, genetic algorithms, ant colony optimization or particle swarm optimization. The disadvantage of metaheuristics is that we lose the guarantee of a global minimum, and the computation can be just as, or even more expensive.

To give an example, one of the well known functions used to test minimization problems is the Six Hump Camel Back function[8], which is a multivariate polynomial of 2 variables and of degree 6, given in Listing 2.3.

Listing 2.3: MatLab Definition of Six Hump Camel Back Function

```
function [ out ] = sixhump( x )
out = (4-2.1*x(1)^2+x(1)^4/3)*x(1)^2+x(1)*x(2)+(-4+4*x(2)^2)*x(2)^2;
end
```

If we consider the constraints: $-2 \leq x_1 \leq 2$ and $-1 \leq x_2 \leq 1$, then the function has two global minima (and 4 local minima), $f(x_1, x_2) = -1.0316$, for $(x_1, x_2) = (-0.0898, 0.7126)$, or $(0.0898, -0.7126)$. We illustrate the function in Figure 2.3. The two dark blue valleys correspond to the two global minima.

[7] See [Ding et al., 2006] for details.

[8] For example, it is mentioned in the GEATbx MatLab toolbox (http://www.geatbx.com/docu/fcnindex-01.html (accessed 01.09.2011)).

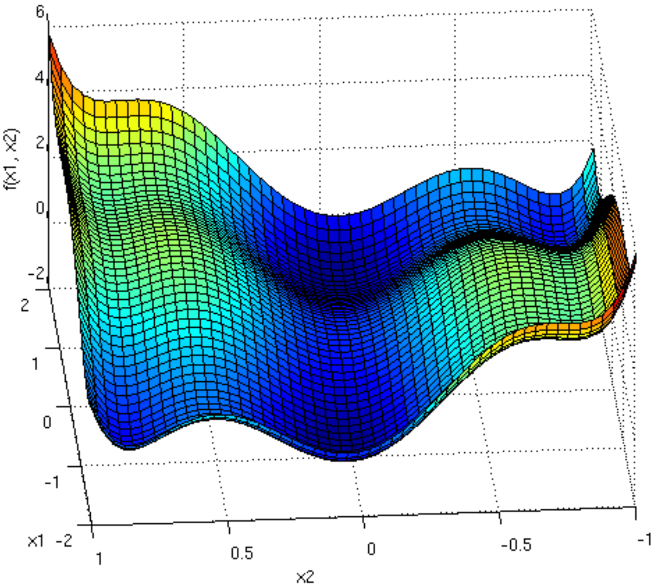


Figure 2.3: Plot of the Six Hump Camel Back Function.

Finally, we mention two theorems which allow us to compute polynomials raised to a power.

Theorem 2.2 (Binomial Theorem)**.** *The expansion of* $(x + y)^n$ *is:* $(x + y)^n = C(n, 0)x^n y^0 + C(n, 1)x^{n-1}y^1 + \cdots + C(n, n-1)x^1 y^{n-1} + C(n, n)x^0 y^n$*, with* $x, y \in \mathbb{R}$*, and* $n \in \mathbb{N}^*$

$C(n, k)$ are binomial coefficients, and:
$C(n, k) = \frac{n!}{k!(n-k)!}$, $n \in \mathbb{N}^*$, and $0 \le k \le n$
A more general result is the multinomial theorem.

Theorem 2.3 (Multinomial Theorem)**.** *The expansion of* $(x_1 + x_2 + \cdots + x_m)^n$ *is:* $(x_1 + x_2 + \cdots + x_m)^n = \sum\limits_{k_1+k_2+\cdots+k_m=n} (C(n; k_1, k_2, \ldots, k_m) \prod\limits_{t=1}^{m} x_t^{k_t})$

$C(n; k_1, k_2, \ldots, k_m)$ are multinomial coefficients, and:
$C(n; k_1, k_2, \ldots, k_m) = \frac{n!}{k_1!k_2!\ldots k_m!}$
Note that for binomial and multinomial coefficients although we give the usual definitions based on the factorials, the implementation uses very fast MatLab functions.

Part I

ProbPoly - a new PILP framework

Chapter 3

Related Work in (Probabilistic) Inductive Logic Programming

This chapter presents the (P)ILP frameworks which have inspired us to develop ProbPoly. In Section 3.1 we briefly introduce the idea of top directed hypothesis derivation, illustrated by TopLog. In the next chapter, we will adapt the TopLog scoring using probabilistic facts and examples, based on SLP semantics. In Section 3.2 we continue the discussion on SLPs, introduced in Section 2.4, with an emphasis on learning SLPs, which is also the main goal of ProbPoly. Finally, Section 3.3 succinctly describes a popular probabilistic logic programming language, ProbLog, based on distribution semantics, introduced in Section 2.3, and ProbFOIL, a PILP system implemented in ProbLog, based on the ILP FOIL system [Quinlan and Cameron-Jones, 1993].

3.1 TopLog

3.1.1 Top Directed Hypothesis Derivation

TopLog is an Inductive Logic Programming system which illustrates the theoretical framework of *Top-Directed Hypothesis Derivation* (TDHD), described in [Muggleton et al., 2008]. The key idea of TDHD is to build a *top theory* $\top$ from the background knowledge and based on a set of *non-terminal symbols* NT. The predicates in NT must be new predicates, different from the ones in the background knowledge B. The top theory $\top$ is a general logic program from which the hypotheses H is be derived. $\top$ must contain as the head of the first clause the target predicate, and as heads of the other clauses in $\top$, non terminal symbols in NT.

We reproduce Example 1 from [Muggleton et al., 2008]. Let:

$$\begin{aligned} NT &= \{ntBody\} \\ B &= b_1 = pet(lassy) \leftarrow \\ e &= nice(lassy) \leftarrow \end{aligned}$$

Then, $\top$ could be written as:

$$\begin{aligned} &\top_1 : nice(X) \leftarrow ntBody(X) \\ &\top_2 : ntBody(X) \leftarrow pet(X) \\ &\top_3 : ntBody(X) \leftarrow friend(X) \end{aligned}$$

3.1.2 Mode Declariations and Hypothesis Generation

The top theory can also be constructed from *mode declarations.*

Definition 3.1 (Mode Declaration)**.** *A mode declaration, as defined in [Corapi et al., 2010], is either a head (modeh(s)) or a body declaration (modeb(s)), where s is called a* schema*, which is a ground literal containing placemarkers.* Placemarkers *are either inputs (+type), outputs(−type), or ground terms (#type).*

Consider the following example[1]:

Example 3.1. *Learning the* uncle *predicate.*

$B = \{$

b_1 : *male(tom)*
b_2 : *male(bob)*
b_3 : *parent(tom, mary)*
b_4 : *parent(tom, bob)*
b_5 : *parent(tom, betty)*
b_6 : *parent(mary, ann)*
b_7 : *parent(joyce, susan)*

$\}$

$E = \{$

ID	*Example clause*	*Example weight*
e_1 :	*uncle(bob, ann)*	+10
e_2 :	*uncle(bob, susan)*	−10
e_3 :	*uncle(betty, ann)*	−10
e_4 :	*uncle(tom, betty)*	−10
e_5 :	*uncle(joyce, ann)*	−10
e_6 :	*uncle(tom, mary)*	−10

$\}$

Assume the following mode declarations:

$M = \{$

modeh(uncle(+person))
modeb(male(+person))
modeb(parent(+person, -person))
modeb(parent(-person, +person))

$\}$

We might build $\top$ as :

$\top = \{$

$\top_1 : uncle(X, Y) \leftarrow ntBody(X), ntBody(Y)$
$\top_2 : ntBody(X) \leftarrow$
$\top_3 : ntBody(X) \leftarrow male(X), ntBody(X)$
$\top_4 : ntBody(X) \leftarrow parent(X, Z), ntBody(X), ntBody(Z)$
$\top_5 : ntBody(X) \leftarrow parent(Z, X), ntBody(X), ntBody(Z)$

$\}$

Once we have built $\top$, the generation of the hypotheses set H is based on iterating on the positive examples E^+: starting with $H = \emptyset$ for each positive example $e \in E^+$, we consider the refutations r of e using B and $\top$, and based on the refutations we build the hypotheses H_e and merge them with H.

For the above example, we may have many hypotheses $h_i \in H$, such as:

$h_1 : uncle(X, Y).$
$(not(uncle(bob, ann)), \top_1, | \top_2, | \top_2)$

$h_2 : uncle(X, Y)$:- $male(X).$
$(not(uncle(bob, ann)), \top_1, | \top_3, b_2, \top_2, | \top_2)$

$h_3 : uncle(X, Y)$:- $parent(Z, X).$

[1] For the examples, a positive weight denotes a positive example, and a negative one – a negative example.

$(not(uncle(bob, ann)), \top_1, | \top_5, b_4, \top_2, \top_2, | \top_2)$

$h_4 : uncle(X, Y)$:- $parent(Z.Y)$.

$(not(uncle(bob, ann)), \top_1, | \top_2, | \top_5, b_6, \top_2, \top_2)$

...

In brackets we show the corresponding derivation of the (only) positive example e_1. We have used | to mark the beginning of the refutations for each argument of $\top_1$. Notice that for example the hypothesis $uncle(X, Y)$:- $parent(X, Z)$. cannot be generated, since *bob* isn't anyone's parent in B. Now let us consider more complex hypotheses, among which we will find the "correct" definition of the uncle predicate (h_7):

...

$h_5 : uncle(X, Y)$:- $male(X), parent(Z, Y)$.

$(not(uncle(bob, ann)), \top_1, | \top_3, b_2, \top_2, | \top_5, b_6, \top_2, \top_2)$

$h_6 : uncle(X, Y)$:- $male(X), parent(Z_1, X), parent(Z_2, Y)$.

$(not(uncle(bob, ann)), \top_1, | \top_3, b_2, \top_5, b_4, \top_2, \top_2 | \top_5, b_6, \top_2, \top_2)$

$h_7 : uncle(X, Y)$:- $male(X), parent(Z_1, X), parent(Z_1, Z_2), parent(Z_2, Y)$.

$(not(uncle(bob, ann)), \top_1, | \top_3, b_2, \top_5, b_4, \top_2, \top_4, b_3, \top_2, \top_4, b_6, \top_2, \top_2 | \top_2)$

$h_8 : uncle(X, Y)$:- $parent(Z_1, X), parent(Z_1, Z_2), parent(Z_2, Y)$.

$(not(uncle(bob, ann)), \top_1, | \top_3, \top_5, b_4, \top_2, \top_4, b_3, \top_2, \top_4, b_6, \top_2, \top_2 | \top_2)$

...

3.1.3 Obtaining a Final Theory

After the generation of all hypotheses, the TopLog system computes the final theory T, which is a subset of the derived hypotheses H. Initially, T is empty, and hypotheses from H are added according to a greedy heuristic, more specifically at each iteration hypothesis $h \in H$ is added to T if it gives the maximum score of $T \cup h$ (relative to the other existing hypotheses). The score of T is computed as:

$$S(T) = \sum_{e \in E_{CT}} weight(e) - \sum_{h \in T} |h|$$

We use E_{CT} to denote the set of examples covered by the hypotheses in T, and by $weight(e)$ the weight of an examples (a positive value for a positive example and a negative value otherwise). $|h|$ is the number of literals in hypothesis h. Thus, the two components of the score account for the maximization of the weights of the examples (the first sum), while penalizing complex rules (the second sum), which is an ILP way of implementing the Minimum Description Length (MDL) principle.

Again, consider the example of hypotheses h_5–h_8, and assume $T = \emptyset$. We compute the following scores, for $weight(e) = 10$ for a positive example and -10 otherwise:

$$\begin{aligned}
S(T \cup h_5) &= \overbrace{10}^{e_1} + \overbrace{(-10)}^{e_2} + \overbrace{(-10)}^{e_4} + \overbrace{(-10)}^{e_6} - \overbrace{3}^{\#\text{ literals}} &&= -23 \\
S(T \cup h_6) &= \overbrace{10}^{e_1} + \overbrace{(-10)}^{e_2} - \overbrace{4}^{\#\text{ literals}} &&= -4 \\
S(T \cup h_7) &= \overbrace{10}^{e_1} - \overbrace{5}^{\#\text{ literals}} &&= 5 \\
S(T \cup h_8) &= \overbrace{10}^{e_1} + \overbrace{(-10)}^{e_3} - \overbrace{4}^{\#\text{ literals}} &&= -4
\end{aligned}$$

It is clear that in the first iteration we will add h_7 to T. Also, since every other hypothesis will contribute a negative score, we can anticipate that in the second iteration no hypothesis will be added to T and the final theory will be $T = \{h_7\}$, as we would expect.

3.2 Stochastic Logic Program Learning

The problem of learning probabilities of an SLP is addressed in [Muggleton, 2002]. We must learn a set of probabilistic clauses H such that the SLP $S = B \cup H \models E$ (B is also an SLP and E is a set of ground positive non-probabilistic examples). Additionally, the probabilities of the clauses in the SLP S maximize $P(E \mid S)$. An ILP system which learns one (range-restricted) clause c at a time (like FOIL) is considered, and it is assumed that $B \cup H' \models E$, where $H' = H \cup c$. The problem reduces to finding the probability x of c. Then, the other clauses $y_1 : h_1, \ldots, y_n : h_n$ with the target predicate as head predicate can be normalized by multiplying each $y_i, \forall i = \overline{1,n}$ with $(1 - x)$.

Since we assume *structure learning* (i.e. clause c for each step, and in the end H without annotated probabilities) to be similar to ILP, we are left with *parameter learning* (i.e. probability X of clause c, and in the end the probabilities of each clause in H). The choice of x (such that $0 \leq x \leq 1$) is seen as a maximization problem of the function $P(E|B \cup H)$, defined as:

$$\begin{aligned}
P(E|B \cup H) &= \prod_{e \in E} P(e|B \cup H) \\
&= \prod_{e \in E} \sum_{proof \in SSLD(e, B \cup H)} P(proof) \\
&= \prod_{e \in E} \sum_{proof \in SSLD(e, B \cup H)} \prod_{p:c \in proof} p
\end{aligned}$$

By $SSLD(g, S)$ we denote the SSLD derivation of goal g using the SLP S, and $p : C$ is a probabilistic clause used in the SSLD derivation tree.

The author provides a solution for the case of non-recursive programs, by observing that any probability of a proof will either be:

- a constant *const*, if the example can be derived without using H (this is a trivial case – it means that the example can proven directly from the background knowledge),
- *const* $\times$ x if the derivation of the example uses the new clause C,
- *const* $\times$ $(1 - x)$ if the derivation uses a previously learned clause h_i.

This means that $P(e|B \cup H)$ will be of the form $x(c_1 + c_2 + \ldots) + (1 - x)(d_1 + d_2 + \ldots) + c(e)$. By computing $sum_c = c_1 + c_2 ...$, $sum_d = d_1 + d_2 ...$ and $k_1(e) = sum_c - sum_d$, $k_2(e) = sum_d + c(e)$, we get
$P(e|B \cup H) = k_1(e)x + k_2(e)$.

Instead of maximizing $P(E|B \cup H)$, the author considers the maximization of $ln(P(E|B \cup H))$, which is equivalent to the initial formulation due to the monotony of ln. The function becomes:

$$\begin{aligned}
ln(P(E|B \cup H)) &= ln(\prod_{e \in E} P(e|B \cup H)) \\
&= \sum_{e \in E} ln(P(e|B \cup H))) \\
&= \sum_{e \in E} ln(k_1(e)x + k_2(e))
\end{aligned}$$

The problem of finding x as $\underset{x}{argmax}\ P(E|B \cup H)$ is solved analytically, by setting the derivative of $ln(P(E|B \cup H))$ to 0:

$$\begin{aligned}\frac{\partial ln(P(E|B \cup H))}{\partial x} &= \sum_{e \in E} \frac{1}{k_1(e)x + k_2(e)} \frac{\partial(k_1(e)x + k_2(e))}{\partial x} \\ &= \sum_{e \in E} \frac{k_1(e)}{k_1(e)x + k_2(e)} \\ &= 0\end{aligned}$$

Thus, by considering $k(e) = \frac{k_2(e)}{k_1(e)}$, we get x by computing the solution of $\sum_{e \in E} \frac{1}{x + k(e)} = 0$. In the case of two examples, x is $-\frac{k(e_1)+k(e_2)}{2}$. The author also shows a numerical method to compute x for an arbitrary number of examples.

Let us consider a simple example, based on the one in [Muggleton, 2002], which corrects an error in the refutations, but doesn't keep the program normalized. However, in this particular context, it doesn't affect the purpose or validity of the example.

Example 3.2. *Un-normalized SLP to match refutations in [Muggleton, 2002] Program S:*

x	*: p(X,Y) :- q(X,Z), r(Z,Y).*	*[A]*
1-x	*: p(X,Y) :- r(X,Z), s(Y,Z).*	*[B]*
0.3	*: q(a,b).*	*[C]*
0.4	*: q(b,b).*	*[D]*
0.3	*: q(c,e).*	*[E]*
0.4	*: r(b,d).*	*[F]*
0.6	*: r(e,d).*	*[G]*
0.6	*: r(c,f).*	*[H]*
0.9	*: s(d,d).*	*[I]*
0.1	*: s(e,d).*	*[J]*
0.1	*: s(d,f).*	*[K]*

Examples E:

p(b,d).	$[e_1]$
p(c,d).	$[e_2]$

Then, the SSLD proofs are:

$$SSLD(e_1, S) = \{[A, D, F], [B, F, I]\}$$
$$SSLD(e_2, S) = \{[A, E, G], [B, H, K]\}$$

The equation for $p(E|S)$ is the polynomial:
$p(E|S) = [x(0.4)(0.4) + (1 - x)(0.4)(0.9)] \times [x(0.3)(0.6) + (1 - x)(0.6)(0.1)]$

For e_1 we get $c = c_1 = (0.4)(0.4)$ and $d = d_1 = (0.4)(0.9)$, and we can compute $k_1(e_1)$, $k_2(e_1)$ and $k(e_1)$. In a similar way, for e_2 we get $c = c_1 = (0.3)(0.6)$ and $d = d_1 = (0.6)(0.1)$, and again compute $k_1(e_2)$, $k_2(e_2)$ and $k(e_2)$. x will be $-\frac{k(e_1)+k(e_2)}{2}$, and in this instance, 0.65. The result can be verified visually by plotting $p(E|S)$ as a function of x (it is, in fact, a second degree polynomial).

In conclusion, in SLP learning, we are given as input an SLP ($B \cup H$) and we aim to learn the probability of a clause to be added to H, such that the probability of the examples, which can only be positive, is maximized. This is done in a Maximum-Likelihood (ML) fashion, by deriving an analytical expression, for non-recursive SLPs.

3.3 ProbFOIL - a PILP system in ProbLog

3.3.1 ProbLog

ProbLog is a probabilistic extension of Prolog, implemented in Yet Another Prolog (YAP, [Costa et al., 2011]). The theoretical and practical aspects of ProbLog are presented in the PhD report of A. Kimming [Kimming, 2010].

The formal foundation for extending logic programming with probabilistic elements is the *distribution semantics*, described in [Sato, 1995] and introduced in Section 2.3.

In ProbLog, each probabilistic fact f is treated as an independent random variable and is written as $p :: f$, where p is a probability of the standard Prolog fact f. *ProbLog rules* have the form: $h : -b_1, ..., b_n$, where h is a positive literal not unifying with any probabilistic fact and each b_i is either a probabilistic fact f, the negation of f with f ground, or a positive literal not unifying with any probabilistic fact. This allows the extension of each interpretation of the probabilistic facts into a unique minimal Herbrand model of the program.

A *ProbLog program* is defined as $T = \{p_1 :: f_1, ..., p_n :: f_n\} \cup BK$, where BK is a set of rules as defined above. The *logical facts* of T, L^T, consists of the set of all the possible groundings of all f_i. *Inference* in ProbLog implies two basic steps:

1. Computation of explanations of a query q using the logical part of the theory $(BK \cup L^T)$. The explanations are stored as a DNF formula.

2. Computation of the probability of the formula.

ProbLog relies on *binary decision diagrams* (BDDs) as data structures to represent a boolean formula (result of step 1 of the inference). BDDs are binary trees in which each node represents a propositional variable, and the edges from that node represent the assignment of true or false value to that variable. BDDs can be compressed by using *reduction operators* such as :

- *subgraph merging*, in which the edges going into a subgraph g_1 are redirected to an isomorphic subgraph g_2 and g_1 is deleted, and

- *node deletion*, in which a node n_1 whose edges go to the same node n_2 can be deleted and the edges going into n_1 are connected to n_2.

Maximal compression of a BDD depends on variable ordering, which was proven to be a coNP-complete problem.

One of the advantages of ProbLog is that it is the first probabilistic programming system using BDDs as a basic data structure for efficient probability calculation.

Let us consider a very small example, based on the ProbLog tutorial:[2].

Probability	::	Clause
0.9	::	edge(1, 2).
0.5	::	edge(2, 6).
0.7	::	edge(1, 6).

Assuming we have specified the definition of a *path*/2 predicate, we can query the system for probabilities related to paths in the graph, e.g.

Query to find the maximum probability:

```
?- problog_max(path(1,6),Prob,FactsUsed).
FactsUsed = [dir_edge(1,6)],
Prob = 0.7
```

Query to find the exact probability:

```
?- problog_exact(path(1,6),Prob,Status).
Prob = 0.835,
Status = ok
```

The value for the exact probability might seem strange at first, but recall that we are working in distribution semantics, so the logic behind the computation is:

[2] http://dtai.cs.kuleuven.be/problog/tutorial-inference.html (accessed 04.09.2011)

$\langle edge(1,2), edge(2,6), edge(1,6)\rangle$	$path(1,6)$
$\langle 0,0,1\rangle$	1
$\langle 0,1,1\rangle$	1
$\langle 1,0,1\rangle$	1
$\langle 1,1,1\rangle$	1
$\langle 1,1,0\rangle$	1
others	0

So we have:

$$\begin{aligned} P(path(1,6)) &= 0.1 * 0.5 * 0.7 \\ &+ 0.1 * 0.5 * 0.7 \\ &+ 0.9 * 0.5 * 0.7 \\ &+ 0.9 * 0.5 * 0.7 \\ &+ 0.9 * 0.5 * 0.3 \\ &= 0.835 \end{aligned}$$

ProbLog allows using k-best proofs, sampling, and even parameter learning. Learning aims at estimating parameters of ProbLog programs using examples with annotated probabilities and is accomplished by minimizing a mean square error (MSE) function.

3.3.2 ProbFOIL

ProbFOIL is described in [Raedt and Thon, 2010]. It uses ProbLog to evaluate probabilities of examples given a learned hypothesis $P(H \cup B \models e)$, based on probabilistic facts and non-probabilistic rules. $P(H \cup B \models e)$ has the same semantics as the probability of a goal (given a background knowledge and a hypothesis), written as $P(e|B \cup H)$, from Definition 2.2. The probabilities of the examples $P(e)$ represent the desires explanation probability, so the goal is to find a theory H such that $P(H \cup B \models e)$ are close to $P(e)$. Similar to a regression setting, the function used to evaluate the quality of learning is a *loss function*:
$Loss(H) = \sum_{e \in E} |P(e) - P(H \cup B \models e)|$

The goal is to find: $\underset{H}{argmin}\ Loss(H)$, so the learning continues until the inner loop of FOIL doesn't produce a clause c which can minimize $Loss(H \cup c)$. For the inner loop, the authors consider an adapted definition of the traditional Information Retrieval (IR) quantities: true/false positives, true/false negatives. Consider p_i the probability of an example, $n_i = 1 - p_i$ and $p_{h,i}$ the predicted probability, $n_{h,i} = 1 - p_{h,i}$. The following quantities are introduced, for each example e_i:

- the true positive part $tp_i = min(p_i, p_{h,i})$
- the true negative part $tn_i = min(n_i, n_{h,i})$
- the false positive part $fp_i = max(0, n_i - tn_i)$
- the false negative part $fn_i = max(0, p_i - tp_i)$

Consider also the definitions of true/false positive/negative parts over the whole set of examples:

$$TP = \sum_{e_i \in E} tp_i,\ TN = \sum_{e_i \in E} tn_i,\ FP = \sum_{e_i \in E} fp_i,\ FN = \sum_{e_i \in E} fn_i,$$

Let us consider a small example where $p_i = 0.5$ and $p_{h,i} = 0.3$. We illustrate the above-defined quantities in Figure 3.1. Note that the false positive part is in this case 0.

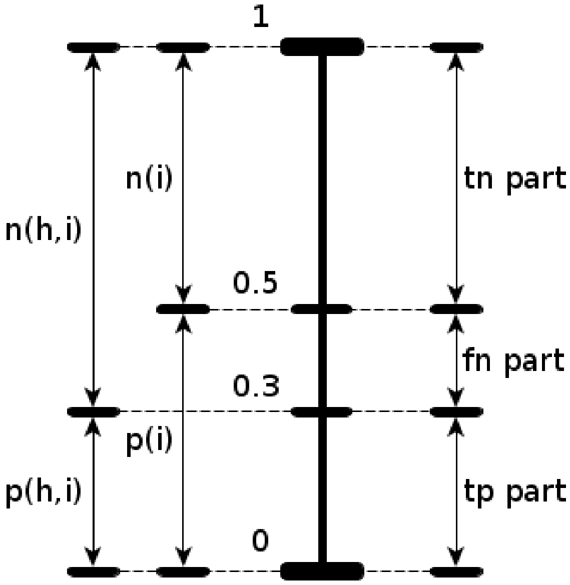


Figure 3.1: True/false positive/negative parts for $p_i = 0.5$ and $p_{h,i} = 0.3$.

Based on these quantities, the authors define generalized *information retrieval* (IR) measures such as precision, recall, accuracy etc. Such a measure (called *m-measure*, similar but more robust than precision) is used to define a local score in the process of generating clause c to be added to H:

$localscore(H, c) = m - estimate(H \cup \{c\}) - m - estimate(H)$

Finally, the stopping criterion is defined as:

$localstop(H, c) = (TP(H \cup \{c\}) - TP(H) = 0) \vee (FP(\{c\}) = 0)$

Informally, this means that the search stops when there are no more false positives, or if the refined clauses doesn't contribute to the increase in the true positive part. The authors also propose a post-pruning of the refined rules.

To summarise, ProbFOIL performs learning of ProbLog programs based on probabilistic examples, in the context of distribution semantics, using a FOIL-like approach, which aims at minimizing the error between the observed and predicted probabilities of the examples, while guiding the refinement process according to generalized IR inspired measures.

Chapter 4

ProbPoly

After establishing a background in PILP and reviewing some of the most significant systems in the field, we now focus on developing a new framework for PILP. This is the main theoretical contribution of our thesis. We extend learning SLPs to learning with probabilistic examples, overcome the important limitation of learning only non-recursive programs, establish a way to interpret negation as failure, and extend learning to multiple clauses. Recall that SLPs were initially meant to represent PCFGs, which are recursive in almost all real-world applications. Learning probabilities for multiple clauses is crucial as well, since PCFGs are defined by a large number of parameters. Due to the powerful optimization mechanisms behind ProbPoly, we almost always manage to obtain very good learning results, given an adequate structure.

This chapter is organized as follows: Section 4.1 defines a probabilistic score for the TopLog system, which is not actually part of the ProbPoly idea, and hasn't been implemented, but is an introductory experiment in PILP. The meaning of the probabilities are similar to the usual weights: they are meant to quantify the quality of the examples, i.e. the trust we place in them. In Section 4.2 we describe the adaptation of a simple iterative deepening Prolog meta-interpreter to the probabilistic context of SLPs. The focus is on clarity of ideas rather than efficiency. In the concluding Section 4.3 we present the main theoretical results behind ProbPoly accompanied by small, but hopefully illustrative examples.

4.1 A Simple Score for Probabilistic Facts in TopLog

This section is a direct continuation of Section 3.1, so we assume implicit the notation convention and the context of TopLog learning.

In the context of probabilistic facts in B and probabilistic examples in E, we could use the derivation of H, but adapt the score used for generating the final theory to take into account probabilities. A simple solution for such a score would be:

$$S_{prob}(T) = \sum_{e \in E_{CT}} weight(e) L(e|T) P(e).$$

We will consider the weight of the examples 1 for a positive example and -1 otherwise, and for simplicity, we assume complete certainty for all examples ($P(e) = 1, \forall e \in E$). $L(e|T)$ is the likelihood that example e is derived from theory T, and is computed as the sum of an example being derived from any hypothesis:

$$P(e|T) = \sum_{h \in T} P(e|h).$$

Let $RF(e,h)$ (RF – Refutation Facts) be the set of probabilistic facts in the refutation of e by h, denoted b (of course, taken from the background knowledge B). $P(e|h)$ is computed as the product of the probabilities of the facts appearing in the refutation of e by h:

$$P(e|h) = \prod_{b \in RF(e,h)} P(b).$$

Under this assumption, we consider that each example has a single refutation by any one hypothesis. To generalize to multiple refutations, we can assume $P(e|h)$ is equivalent to $P(e|B \cup$

$\{h\}$), and $P(e|T)$ to $P(e|B \cup \{T\})$ from Definition 2.2.

We illustrate the computation of this score on the *uncle* toy dataset, by annotating the background knowledge facts with probabilities:

Example 4.1. *Learning the* uncle *predicate with probabilistic facts.*

$$
\begin{array}{llll}
B = \{ & & & \\
 & b_1: & male(tom). & :: 0.8 \\
 & b_2: & male(bob). & :: 0.3 \\
 & & & \\
 & b_3: & parent(tom, mary). & :: 0.7 \\
 & b_4: & parent(tom, bob). & :: 0.5 \\
 & b_5: & parent(tom, betty). & :: 0.9 \\
 & b_6: & parent(mary, ann). & :: 0.6 \\
 & b_7: & parent(joyce, susan). & :: 0.4 \\
\} & & &
\end{array}
$$

The probabilistic scores for hypotheses h_5–h_8 will be:

$$
\begin{aligned}
S_{prob}(T \cup h_5) &= \overbrace{\underbrace{0.3}_{P(male(bob))} * \underbrace{0.6}_{P(parent(mary,ann))} * \underbrace{1}_{P(e_1)} * \underbrace{1}_{weight(e_1)}}^{e1} \\
&+ \overbrace{0.3 * 0.4 * -1}^{e2} + \overbrace{0.8 * 0.9 * -1}^{e4} + \overbrace{0.8 * 0.7 * -1}^{e6} \\
&= -1.22 \\
S_{prob}(T \cup h_6) &= \overbrace{0.3 * 0.5 * 0.6 * 1}^{e1} + \overbrace{0.3 * 0.5 * 0.4 * -1}^{e2} \\
&= 0.03 \\
S_{prob}(T \cup h_7) &= \overbrace{0.3 * 0.5 * 0.7 * 0.6 * 1}^{e1} \\
&= 0.063 \\
S_{prob}(T \cup h_8) &= \overbrace{0.5 * 0.7 * 0.6 * 1}^{e1} + \overbrace{0.9 * 0.7 * 0.6 * -1}^{e3} \\
&= -0.168
\end{aligned}
$$

Although the results are very much influenced by the probabilities we have chosen, we notice that h_7 is still the hypothesis with maximum score. However, we also find a positive hypothesis in h_6, because we are more certain on *mary* being *ann*'s parent (0.6), then on *joyce* being *susan*'s parent(0.4), which clearly gives more weight to the positive example. Another important aspect of this choice of score is that, although simple, it implicitly incorporates the Minimum Description Length (MDL) principle, because a longer hypothesis will need additional facts in the refutation, and consequently there will be more probabilities to compute in the product of $P(e|h)$, and since all $P(b)$ are real probabilities ($P(b) \in (0, 1]$), they will diminish the score of the particular derivation.

4.2 A Probabilistic Hypothesis Interpreter

The core of a probabilistic logic programming environment is the *hypothesis interpreter* (we will also refer to it as *prover*). Unlike the sophisticated prover of ProbLog (briefly described in Section 3.3), we have developed one based on the prover for the (mini)HYPER ILP system described in [Bratko, 2000], shown in Listing 4.1.[1]

HYPER (Hypothesis Refiner) and miniHYPER are two small ILP systems presented in the ILP chapter of [Bratko, 2000]. HYPER uses the best first strategy to search a refinement graph according to the cost of hypothesis H:

[1] The code is available online at the book's website.

$Cost(H) = w_1 * Size(H) + w_2 * NegCover(H)$,

where w_1 and w_2 are tunable weights, $NegCover(H)$ is the number of negative examples covered by H and $Size(H)$ is:

$Size(H) = k_1 * \#literals(H) + k2 * \#variables(H)$

HYPER is able to learn multiple clauses in H, thus actually searching over a refinement forest. The search is performed in a top-down manner, usually starting from the clause with the head predicate as a fact and applying a range of refinements. Variables can be typed and language bias is specified via predicates such as *backliteral*, *prolog_predicate* and *start_clause*.

Examples of predicates that can be learned using HYPER include the definition of a path in a graph and insert sort.

Listing 4.1: Hypothesis Interpreter from [Bratko, 2000].

```
% Figure 19.3  A loop-avoiding interpreter for hypotheses.


% Interpreter for hypotheses
% prove( Goal, Hypo, Answ):
%   Answ = yes, if Goal derivable from Hypo in at most D steps
%   Answ = no, if Goal not derivable
%   Answ = maybe, if search terminated after D steps inconclusively

prove( Goal, Hypo, Answer)  :-
  max_proof_length( D),
  prove( Goal, Hypo, D, RestD),
  (RestD >= 0, Answer = yes          % Proved
   ;
   RestD < 0, !, Answer = maybe      % Maybe, but it looks like
                                     % inf. loop
  ).

prove( Goal, _, no).       % Otherwise Goal definitely cannot be proved

% prove( Goal, Hyp, MaxD, RestD):
%   MaxD allowed proof length, RestD 'remaining length' after proof;
%   Count only proof steps using Hyp

prove( G, H, D, D)  :-
  D < 0, !.                % Proof length overstepped

prove( [], _, D, D)  :-  !.

prove( [G1 | Gs], Hypo, D0, D)  :-  !,
  prove( G1, Hypo, D0, D1),
  prove( Gs, Hypo, D1, D).

prove( G, _, D, D)  :-
  prolog_predicate( G),                % Background predicate in Prolog?
  call( G).                            % Call of background predicate

prove( G, Hyp, D0, D)  :-
  D0 =< 0, !, D is D0-1                % Proof too long
  ;
  D1 is D0-1,                          % Remaining proof length
  member( Clause/Vars, Hyp),           % A clause in Hyp
  copy_term( Clause, [Head | Body] ),  % Rename variables in clause
  G = Head,                            % Match clause's head with goal
  prove( Body, Hyp, D1, D).            % Prove G using Clause
```

As shown in the listing, the prover is simply a Prolog predicate which, given a goal *Goal* and an hypothesis *Hypo*, should always succeed and bind answer *Answ* to 'yes' if *Goal* is provable with *Hypo* (and the Background Knowledge – whose predicates are encoded via *prolog_predicate*/1), 'maybe' if after D steps the search hasn't terminated (where D is a parameter specified by $max\_proof\_length(D)$, and the steps are counted by the use of predicates in *Hypo*), or 'no' if *Goal* is not derivable.

In an ILP context, the top goal of the prover will always be an example. In HYPER, the prover is used to test if all positive examples are covered and if none of the negative examples are covered. An example is covered by a hypothesis H if we $B \cup H \models e$, and in HYPER this relation can be verified by calling $prove(Example, H, Answer)$, assuming $Example$ is bound to e. The quantities of covered positive examples and negative examples are used in virtually all ILP systems either in the definition of termination criteria or in the design of scores to evaluate the quality of (partial) hypotheses.

The hypothesis $Hypo$ is encoded as a list of clauses, and each clause is encoded as a $Clause/Var$ pair, where $Clause$ is a ordered list of predicates (e.g. the first one is the head of the rule), and Var is the list of variables in $Clause$. For example, the clause $p(X,Y) :- r(X,Z), s(Y,Z)$ is encoded as: $[p(\_X,\_Y), r(\_X,\_Z), s(\_Y,\_Z)]/[\_X,\_Y,\_Z]$.

Our probabilistic hypothesis interpreter should function the same as the prover discussed, however the clauses, both in the background knowledge and in the hypothesis $Hypo$ will have probabilities, and the answer $Answ$ will be the probability of a proof (or refutation) of $Goal$, as defined in Definition 2.1, or 0 in the case of a failed derivation or incomplete search. In fact, in the case of an incomplete search, the probability is not necessarily 0 (which is a worst case assumption), rather it could be in the interval $[0, P(Goal|B \cup H, proof))$, where $proof$ is the partial derivation where the search has stopped. We will explore this idea further in Section 4.3.

The stochastic clauses $p : C$ are specified in the background knowledge as: $pc(p, C)$, and that the hypothesis $Hypo$ consists of clauses of the form: $Clause/Var$, where $Clause = [p,\ head,\ body_1, \ldots, body_n]$, and p denotes the probability of the clause. A probabilistic prover is given in Listing 4.2.

Listing 4.2: A simple Probabilistic Hypothesis Interpreter

```
% === prove/3
% === prove(Goal, Hypo, Answ) - unlike the usual prover,
% in which Answ is yes if a proof of Goal using Hypo is found,
% no if the derivation fails,
% or maybe if it isn't found at depth<=D (from max_proof_length(D)),
% Answ is a probability:
% the probability of the proof found instead of yes
% 0 instead of maybe
% 0 instead of no
prove( Goal, Hypo, Answ)  :-
  max_proof_length( D),
  prove( Goal, Hypo, D, _RestD, Answ, 1).

prove( _Goal, _, 0).        % Otherwise Goal definitely cannot be proved

% prove( Goal, Hyp, MaxD, RestD):
%   MaxD allowed proof length, RestD 'remaining length' after proof;
%   Count only proof steps using Hyp

prove( _G, _H, D, D, 0, _PAcc)  :-
  D < 0, !.                 % Proof length overstepped

prove( [], _Hyp, D, D, P, P)  :-  !.

prove( [G1 | Gs], Hypo, D0, D, P, PAcc)  :-  !,
  prove( G1, Hypo, D0, D1, P1, PAcc),
  prove( Gs, Hypo, D1, D, P, P1).

prove( G, _Hyp, D, D, P, PAcc)  :-
  prolog_predicate( G),                 % Background predicate in Prolog?
  pc(Prob, G),
  P is PAcc*Prob,
  call(pc(_P, G)).                      % Call of background predicate

prove( G, Hyp, D0, D, P, PAcc)  :-
  D0 =< 0, !, D is D0-1                 % Proof too long
  ;
  D1 is D0-1,                           % Remaining proof length
  member( Clause/_Vars, Hyp),           % A clause in Hyp
  copy_term( Clause, [Prob, Head | Body] ), % Rename variables in clause
  G = Head,                             % Match clause's head with goal
  PAcc1 is PAcc*Prob,
  prove( Body, Hyp, D1, D, P, PAcc1). % Prove G using Clause
```

In the probabilistic case, it is useful to also compute the total probability of a goal, defined in

Definition 2.2, and this is easily accomplished by considering the sum of all the solutions to the probabilistic prover, as illustrated in Listing 4.3

Listing 4.3: Predicate for Computing the Total Probability of a Goal

```
% === total_prob/3
% === total_prob(G, Hyp, Prob) -
% for goal G and thoery Hyp, find the total probability
% of G being proved by Hyp by summing all the proofs
total_prob(G, Hypo, Prob) :-
    findall(P,prove(G,Hypo,P),Probs),
    sum_list(Probs, Prob).
```

In the next section we will explore learning probabilities, and consequently the representation for the prover and its result will change (due to unknown probabilities), however the general structure will be identical with that of the probabilistic prover in Listing 4.2.

4.3 Learning SLPs in ProbPoly

The learning framework proposed in this section combines the two PILP approaches discussed in Sections 3.2 and 3.3. Our aim is to learn SLPs based on probabilistic examples

After reviewing the methods in Sections 3.3 and 3.2, the idea of combining the two methods seemed a reasonable improvement, that is to develop a framework in which SLPs are learned based on probabilistic examples, and the aim of the learning process is to have an SLP that explains the examples optimally.

ProbPoly has as input: an SLP B, a set of probabilistic clauses H, one (or in the multi-clause case more) clauses with unknown probability/probabilities with the same head as the clauses in H and a set of probabilistic examples. The goal is to learn the probabilities of the clauses, thus obtaining H' such that the SLP given by $B \cup H'$ proves examples E with a probability close to that observed.

But what does the probability of an example mean in this situation? If we recall the initial use of SLPs as PCFGs, then if the example is a string produced from the grammar, the probability of the example is the probability of the string being generated by the grammar, and in fact it is the probability of a goal, where the goal is the example itself, from Definition 2.2.

Initially, the setting will be the same as in Section 3.2, but we will change the way we perform parameter learning. Rather than maximizing $p(E|S)$, we will aim at minimizing an error function $Err(E|S)$, which acts like a loss function in the global score of ProbFOIL. However, instead of:

$$\sum_{e \in E} |P(e) - P(H \cup B \models e)|,$$

we will choose the least squares error function, commonly used in machine learning when training neural networks (e.g. [Mitchell, 1997], Chapter 4, equation (4.2)) or solving regression tasks (e.g. [Bishop, 2007], Chapter 1, equation (1.2)):

$$Err(E|S) = \frac{1}{2} \sum_{e \in E} (P(e) - P(e|S))^2$$

$P(e)$ are the probabilities of the examples, and $P(e|S)$ is the total probability of explaining example e with the SLP S, and it is the same as $P(B \cup H \models e)$, used in Section 3.2, if we consider $S = B \cup H$.

A natural question is: what happens to *negative examples*? The approach for learning SLPs presented in Section 3.2 considers only positive examples, and in distribution semantics the negative equivalent of $p :: e$ is simply $1 - p :: e$. Nevertheless, the answer isn't hard to find, because by going back to the PCFG analogy, a counterexample for a grammar should be a string that can't be produced by the grammar, so a negative example should be one that can't be derived by an SSLD derivation, and in this case it's probability is 0.

Although the fact that negative examples are, in our formalism, examples annotated with 0 probability, seems just a special case, it is in fact important when we consider the optimization of the error function over the examples. It is clear that the difference between an example having

0.8 observed and 0.9 predicted probability and an example having 0.0 observed and 0.1 predicted probability (or viceversa) is crucial. However our error function doesn't distinguish this case, since $(0.8-0.9)^2=(0.0-0.1)^2=0.1$. We will see in Chapter 5 how we can enforce the constraint that for negative examples $P(e)=P(e|S)=0$.

Moving on from the special case of negative examples, we now proceed to gradually generalize our approach, starting from non-recursive SLPs, then making an important leap to recursive SLPs, after which we deal with the minor but noteworthy subject of integrating negation as failure, and finally we describe the problem in which we want to learn probabilities for multiple clauses in one step of the learning process.

4.3.1 Non-recursive SLPs

A non-recursive SLP is a program S consisting of a set of probabilistic clauses forming the background knowledge B and a set of probabilistic clauses with the same head predicate h the hypothesis H such that any clause in the hypothesis cannot have in its body predicate h. We usually don't allow h to be the head of a clause in B, because in every SLP clauses with the same head need to be normalized (i.e. their probabilities must sum up to 1). Moreover, non-recursive SLPs cannot have a clause in B whose body contains h.

Due to the constraints mentioned above, we have $P(e|S)=k_1(e)x+k_2(e)$, with the same meaning from Section 3.2. We can prove there is an analytical solution to:

$$(arg)\underset{x}{min}\frac{1}{2}\sum_{e\in E}(P(e)-k_1(e)x-k_2(e))^2$$

Note that we use $(arg)min$ to denote the fact that we are interested in both the solution x and in the minimum value of the error function.

Taking the derivative we get:

$$\begin{aligned}\frac{\partial Err}{\partial x} &= \frac{1}{\partial x}\frac{1}{2}\sum_{e\in E}(P(e)-k_1(e)x-k_2(e))^2\\ &= \sum_{e\in E}(P(e)-k_1(e)x+k_2(e))\frac{1}{\partial x}(P(e)-k_1(e)x-k_2(e))\\ &= \sum_{e\in E}(P(e)-k_1(e)x-k_2(e))\cdot(-k_1(e))\\ &= \sum_{e\in E}(k_1(e)^2x+k_1(e)k_2(e)-k_1(e)P(e))\end{aligned}$$

Setting it to zero yields:

$$x=\frac{\sum\limits_{e\in E}k_1(e)P(e)-\sum\limits_{e\in E}k_1(e)k_2(e)}{\sum\limits_{e\in E}k_1(e)^2}$$

In a simpler vector notation $k_1=[k_1(e_1),\ldots,k_1(e_n)]$, and similarly $k_2=[k_2(e_1),\ldots,k_2(e_n)]$ and $P=[P(e_1),\ldots,P(e_n)]$, we get the solution:

$x=\frac{P\cdot k_1-k_1\cdot k_2}{k_1\cdot k_1}$

Example 4.2. *Un-normalized SLP example Program S:*

	Hypothesis H	
X	*: p(X,Y) :- q(X,Z), r(Z,Y).*	*[A]*
1-X	*: p(X,Y) :- r(X,Z), s(Y,Z).*	*[B]*

	Background Knowledge B	
0.5	*: q(a,b).*	*[C]*
0.3	*: q(b,b).*	*[D]*
0.4	*: q(c,e).*	*[E]*
0.1	*: r(b,d).*	*[F]*
0.8	*: r(e,d).*	*[G]*
0.5	*: r(c,f).*	*[H]*
0.4	*: s(d,d).*	*[I]*
0.5	*: s(e,d).*	*[J]*
0.3	*: s(d,f).*	*[K]*

	Examples E	
0.6	*: p(b,d).*	$[e_1]$
0.3	*: p(c,d).*	$[e_2]$

The example is similar to Example 3.2, however notice that the examples have probabilities. The refutations are the same:

$$SSLD(e_1, S) = \{[A, D, F], [B, F, I]\}$$
$$SSLD(e_2, S) = \{[A, E, G], [B, H, K]\}$$

We will have:

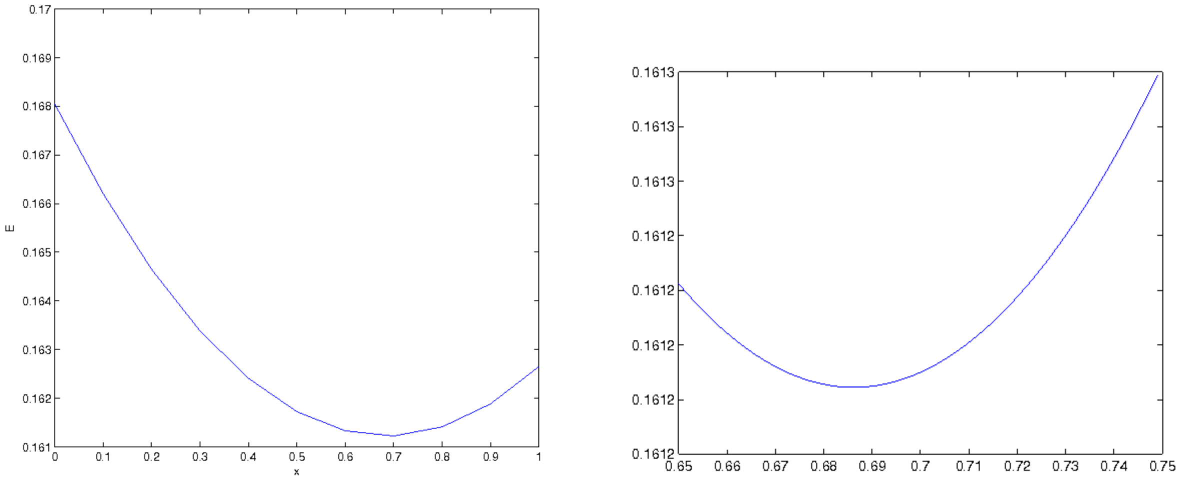
- for e_1 – $X(0.3)(0.1) + (1 - X)(0.1)(0.4) = -0.01 * X + 0.04$, so $k_1(e_1) = -0.01, k_2(e_1) = 0.04$;
- for e_2 – $X(0.4)(0.8) + (1 - X)(0.5)(0.3) = 0.17 * X + 0.15$, so $k_1(e_2) = 0.17, k_2(e_2) = 0.15$.

A simple computation reveals that X is $\simeq 0.6862$. We present in Figure 4.1 the error function plotted against X. Our method returns:

$X = 0.6862068965517237$,

$Err = 0.1612222413793103$,

which seems to be a good result. Note that the results presented in the next subsections are more general, so we get the same solution using techniques for recursive or non-recursive programs.



(a) Complete plot.

(b) Detail of the same plot.

Figure 4.1: Example plot of the error function against probability X for Example 4.2.

4.3.2 Recursive SLPs

Due to the fact that limiting ourselves to non-recursive programs is a very strong constraint, we find it highly desirable to overcome this problem. In the general case, a proof of an example using an SLP is of the form:

$cx^n(1-x)^m$, $c \in \mathbb{R}^+$ and $n, m \in \mathbb{N}$,

where $n+m$ is the total recursive depth of the target predicate.

To compute the total probability of an example $P(e|S)$, we need to sum up polynomials of the form $cx^n(1-x)^m$, which yields another polynomial[2], of maximal degree $n+m$, denoted by $Poly(e,S)$. Consider $DPoly(e,S)$ the polynomial equal to the (first order) derivative of $Poly(e,S)$ with respect to x. The minimization problem is the same:

$\underset{x}{argmin}\frac{1}{2}\sum\limits_{e\in E}(P(e) - Poly(e,S))^2$,

but $Err = \frac{1}{2}\sum\limits_{e\in E}(P(e) - Poly(e,S))^2$ will be a polynomial in x, $Err(x)$.

Taking the derivative:

$$\begin{aligned}
\frac{\partial Err}{\partial x} &= \frac{1}{\partial x}\frac{1}{2}\sum_{e\in E}(P(e) - Poly(e,S))^2 \\
&= \sum_{e\in E}(P(e) - Poly(e,S))\frac{1}{\partial x}(P(e) - Poly(e,S)) \\
&= \sum_{e\in E}(P(e) - Poly(e,S)) \cdot (DPoly(e,S)) \\
&= \sum_{e\in E}(DPoly(e,S)P(e) - Poly(e,S)DPoly(e,S))
\end{aligned}$$

However, $DPoly(e,S)P(e) - Poly(e,S)DPoly(e,S)$ is also a polynomial, because it is the difference of two polynomials: $DPoly(e,S)$ multiplied by a scalar $P(e)$ and $DPoly(e,S)$ multiplied with $Poly(e,S)$, which yields a polynomial. Furthermore, by summing up all the polynomials we get a polynomial, of maximal degree $(n+m)*(n+m-1)$, so we can assert that: $\frac{\partial Err}{\partial x} = SolPoly(x)$, with $SolPoly$ a polynomial. Setting the derivative to zero is equivalent to finding the roots of polynomial $SolPoly$. Having found the roots of the derivative of the polynomial, and recalling that our objective is to minimize $Err(x)$, for $x \in [0,1]$, all we need to do to find the optimum is:

1. discard complex roots and real roots not in $[0,1]$;
2. check for each root r $Err(r)$ against the current minimum and keep the lowest value for $Err(x)$ together with the solution x.
3. check $Err(0)$ and $Err(1)$, the limits of the interval, against the current minimum and keep the lowest value for $Err(x)$ together with the solution x.

Optionally, we could check for convexity, that is evaluate the second order derivative of $Err(X)$, in order to discard local maxima.

Due to uncertainty in the size of the list of roots, we also consider direct minimization of $Err(x)$ via the free MatLab solution GloptiPoly (versions 2 and 3, [Henrion and Lasserre, 2002] and [Henrion et al., 2007]), which specializes in finding the global optimum of (multivariate) polynomials.

Let us again consider an example:

Example 4.3. *Un-normalized recursive SLP example Program S:*

[2] We use the binomial theorem to compute $(1-x)^m$, when $m > 0$.

	Hypothesis H	
X	*: path(X,X).*	*[A]*
1-X	*: path(X,Y) :- link(X,Z), path(Z,Y).*	*[B]*
	Background Knowledge B	
0.5848	*: link(a,b).*	*[C]*
0.5848	*: link(b,c).*	*[D]*
0.5848	*: link(c,d).*	*[E]*
	Examples E	
0.2	*: p(a,d).*	$[e_1]$

We have a single refutation:

$$\overbrace{(1-X)*0.5848}^{\text{link(a,b)}}*\overbrace{(1-X)*0.5848}^{\text{link(b,c)}}*\overbrace{(1-X)*0.5848}^{\text{link(c,d)}}*\overbrace{\quad X\quad}^{\text{path(d,d)}}.$$

Thus, we get the polynomial:

$$\begin{aligned} Poly(e_1,S) &= (1-X)^3 * X * 0.5848^3 \\ &= (-X^4 + 3*X^3 - 3*X^2 + X)*0.5848^3 \end{aligned}$$

The error function will be:

$$Err = (0.2 - Poly(e_1,S))^2$$

And the derivative:

$\frac{\partial Err}{\partial x} = ((X^4 - 3*X^3 + 3*X^2 - X)*0.5848^3 + 0.2)*$
$*(4*X^3 - 9*X^2 + 6*X - 1)*0.5848^3$

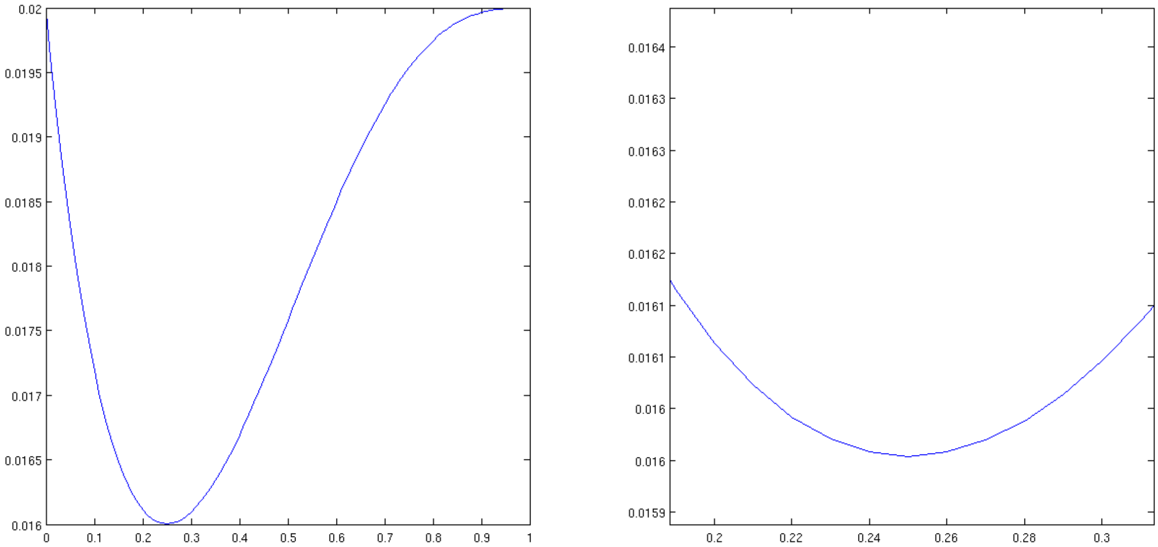
The first term has only complex roots, and the second one $((4*X^3 - 9*X^2 + 6*X - 1)*0.5848^3)$ has complex roots and the real root 0.25.

Running our algorithm we get:

$X = 0.25$,

$Err = 0.0160037918242096$.

The graphical illustration of the error function in Figure 4.2 seems to confirm this theoretical result. The example chosen is quite simple, because otherwise the computation of polynomials and of the error function would have been much more cumbersome. However, the method has been tested on more complex examples, and although it is hard to manually compute the error polynomial and its derivative, the graphical illustration has confirmed the validity of our approach.



(a) Complete plot.

(b) Detail of the same plot.

Figure 4.2: Example plot of the error function against probability X for Example 4.3.

4.3.3 Negation as Failure

Negation as failure (NaF) is treated based on the definition of the *not* predicate in Prolog [Bratko, 2000]:

$not(G)$:- $G, !, fail$.
$not(G)$.

In the standard hypothesis interpreter G either succeeds or fails (or the computation is infinite). In a probabilistic hypothesis interpreter, G succeeds if it has a probability greater than 0. Since we can always normalize an SLP, we ensure this condition to be able to conclude that the sum of the probabilities of all the derivations (irrespective of whether they are refutations or failed derivations) is 1. Since the total probability of G takes into account only derivations, it is natural to consider:

$not(G, NP)$:- $totalProbability(G, P)$, NP is $1 - P$.

The value NP represents the sum of all the failed derivations of G. There is still the case of computation not finishing under certain thresholds which needs to be integrated. In this circumstance, we can only evaluate an upper-bound for the probability.

In the case of computing the probabilities using new clauses, P becomes a polynomial, and the probability NP of $not(G)$ will be the polynomial $1 - P$.

Let us consider an artificial example:

Example 4.4. *Un-normalized recursive SLP example Program S:*

	Hypothesis H	
X	*: p(X) :- q(X).*	*[A]*
1-X	*: p(X) :- r(X), not(s(X)).*	*[B]*

	Background Knowledge B	
0.4	*: s(X) :- t(X, Y), u(X, Y).*	*[C]*
0.1	*: s(X) :- v(X).*	*[D]*
0.3	*: q(a).*	*[E]*
0.5	*: r(a).*	*[F]*
0.7	*: t(a, x).*	*[G]*
0.9	*: u(a, x).*	*[H]*
0.2	*: v(a).*	*[I]*

The total probability of the goal $p(a)$ is computed as follows:

$$\begin{aligned} P(p(a)|S) &= \overbrace{X * 0.3}^{\text{p(X):-q(X)}} + \overbrace{(1-X) * 0.1 * (1 - (\underbrace{0.4 * 0.7 * 0.9}_{\text{s(X) :- t(X, Y), u(X, Y)}} + \underbrace{0.1 * 0.2}_{\text{s(X) :- v(X)}}))}^{\text{p(X) :- r(X), not(s(X))}} \\ &= ... \end{aligned}$$

Integrating negation as failure into the probabilistic prover has consequences on the representation of the polynomials, but such implementation details will be further discussed in Chapter 5. The final version of the probabilistic prover for single-clause learning which takes into account negation as failure and rules in the background knowledge is shown in Listing A.2.

4.3.4 Multiple Clauses

Learning multiple clauses is the final generalization step we apply to our framework. Although we may simulate multiple clause learning by just learning one clause at a time, we find it desirable to be able to learn multiple probabilities simultaneously. The task is much more difficult than learning a single probability. However, the advantage is that we may overcome local minima in the error function evaluated over all clauses. Note that due to normalization of the old clauses, we can basically tune the old clauses in any way we want. This way, the whole theory H "adapts" to the new learned clause. The problem that still remains is that in incremental single clause learning, once we have learned at least 2 clauses out of n, $n > 2$, with probabilities p_i and p_{i+1}, then $\frac{p_i}{p_{i+1}}$ will be the same until the end of the learning, because all "old" clauses are normalized by the same quantity (in the case of single clause learning $(1-x)$). In this circumstance, the proportion $\frac{p_i}{p_{i+1}}$ is determined by the error on clauses $c_1, c_2, \dots, c_{i+1}$, but there is no reason to assume that clauses $c_{i+1}, c_{i+2}, \dots, c_n$ don't influence the dynamics of p_i and p_{i+1}. Multiple clause learning takes advantage of the clauses learned and computes a global minimum over the probabilities of all the clauses.

The problem we aim to solve is:

Given background knowledge SLP B, probabilistic examples E, and

$$\begin{array}{ll} H = \{ & \\ p_1 & : c_1 \\ p_2 & : c_2 \\ \dots & : \dots \\ p_n & : c_n \\ \hline x_1 & : c_{n+1} \\ x_2 & : c_{n+2} \\ \dots & : \dots \\ x_k & : c_{n+k} \\ \} & \end{array}$$

We know $c_1, \dots, c_{n+k}$ and $p_1, \dots, p_n$, and we want to find $[x_1, x_2, \dots, x_k]$ such that $Err(E|B \cup H)$ is minimized. We may refer to the error function also as $Err(x_1, x_2, \dots, x_k)$, to emphasize the parameters that we have to learn.

As usual, we ask ourselves how does the probability of the derivation of an example might look like, and the answer is easy to find, considering H. We will have $P(e|B \cup H, proof)$, according to Definition 2.1, $\forall e \in E$, of the form:

$c \cdot x_1^{m_1} \cdot x_2^{m_2} \cdot \dots \cdot x_k^{m_k}$, where $c \in \mathbb{R}^+$, $m_1, m_2, \dots, m_k \in N$

This is a multivariate polynomial with one term, at most k variables, and of degree $\sum\limits_{i=1}^{k} m_i$.

Then, the total probability of example e, $P(e|B \cup H)$ will be, according to Definition 2.2 and considering that $proofs$ are the SSLD refutations of e by $B \cup H$, and $|proofs|$ is the size of proofs:

$$\sum_{i \in |proofs|} c_i \cdot x_1^{m_{i1}} \cdot x_2^{m_{i2}} \cdot \dots \cdot x_k^{m_{ik}}$$

This is also a multivariate polynomial, of degree $max\_deg\_goal = \max_{i \in |proofs|} m_{i1} + m_{i2} + \cdots + m_{ik}$.

Let $S = B \cup H$ and $MPoly(e, S) = P(e|B \cup H)$, then the error function becomes:
$Err(x_1, x_2, \ldots, x_k) = \frac{1}{2} \sum_{e \in E} (P(e) - MPoly(e, S))^2$

The error function is a multivariate polynomial of degree $2 \cdot max\_deg\_goal$. At this point, we will no longer adopt the gradient descent methodology, instead we will minimize the error function directly, using Gloptipoly.

As usual, after learning, we normalize the learned probabilities by multiplying each of them with $(1 - x_1 - \cdots - x_k)$.

An important remark is that this is a case of constrained optimization, because we need to ensure the following inequalities hold:

$$\begin{gathered} 0 \leq x_1 \leq 1 \\ 0 \leq x_2 \leq 1 \\ \ldots \\ 0 \leq x_k \leq 1 \\ x_1 + x_2 + \cdots + x_k \leq 1 \end{gathered}$$

Ideally we don't allow non-strict inequalities. Consider the following situations:

- in the case that $x_i = 0$, $1 \leq i \leq k$, the clause c_{n+i} is of no use and should be discarded, i.e. we have learned an irrelevant clause.
- in the case that $x_i = 1$, $1 \leq i \leq k$, the clauses $c_1, \ldots, c_n$ and c_{n+j}, $1 \leq i \leq k$, and $j \neq i$ are of no use and should be discarded, i.e. clause c_{n+i} is the only relevant clause.
- in the case that $x_1 + x_2 + \cdots + x_k = 1$, the clauses $c_1, \ldots, c_n$ are of no use and should be discarded, i.e. the old clauses are irrelevant.

However, since, as in the case of single clause learning, we may perform multiple clause learning incrementally (learn k clauses at a time), the clauses which are assigned 0 probability might be discarded later. Even if we would delete the clauses and rely on the ILP system to recover them and learn new, hopefully non-zero or one probabilities, this seems extremely wasteful in terms of time complexity. For this reason, it is desirable to enforce strict inequalities.

Let us consider a small example to illustrate the technique. We will learn only two clauses, so that we can represent the error function graphically. Unlike in the previous sections, we now consider a very simple grammar, which generates a list of a symbols of arbitrary length, shown in Figure 4.3. The text on the edges is a pair of p, s, where p is the transition probability and s is the symbol generated. We assume [] to be the null symbol (because it is the Prolog notation for the empty list).

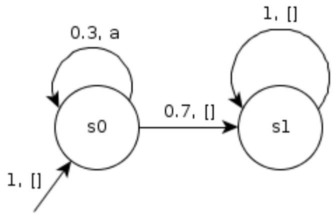


Figure 4.3: Graphical illustration of the simple grammar we aim to learn.

The experiment is the following:

Example 4.5. *Simple grammar for multiple clause learning*
Program S:

	Hypothesis H	
x_1	*: p([]).*	*[A]*
x_2	*: p([a — T]) :- p(T).*	*[B]*

	Background B	
	$\emptyset$	

	Examples E	
0.7	*: p([]).*	e_1
0.063	*: p([a,a]).*	e_2
0.0057	*: p([a, a, a, a]).*	e_3

Note that the probabilities of the examples are consistent with the grammar from Figure 4.3. The output of ProbPoly is:

$Err = 2.56837472178972e - 07,$

$NewH = [[0.700033877366904, p([])]/[], [0.29996612246858, p([a|\_T]), p(\_T)]/[]]$

As expected, the learning was successful, the error is very low and the clauses have the correct probabilities. However, we feel that introducing the first example is almost like cheating, so we test it using only e_2 and e_3. The output of ProbPoly is:

$Err = 2.5580553827129e - 07,$

$NewH = [[0.700044460210975, p([])]/[], [0.299955539764937, p([a|\_T]), p(\_T)]/[]]$

This is a reassuring result, confirming that the prover correctly computes the polynomials and that the correct constraints are generated. Now consider an even more extreme case, in which we learn based only on e_2. The output of ProbPoly is:

$Err = 5.00295134005409e - 07,$

$NewH = [[0.700036139128606, p([])]/[], [0.299963860847507, p([a|\_T]), p(\_T)]/[]]$

Due to the simple structure of the grammar, one example is sufficient to learn the correct probabilities. An interesting phenomenon is that we actually get 2 global solutions from GloptiPoly, the second one is:

$[0.0733568737802477, 0.926643125628242]$

To verify this, we plot the error function and the line $x_1 + x_2 - 1 = 0$ in Figure 4.4. Due to the shape of the polynomial, our best effort is to plot the area where the error is less than 10^{-5}. It is easy to observe that the intersections of this line with the low error area are around $[x_1,\ x_2] = [0.7,\ 0.3]$ and $[x_1,\ x_2] = [0.07,\ 0.93]$. This is a clear confirmation that our program has the correct behaviour.

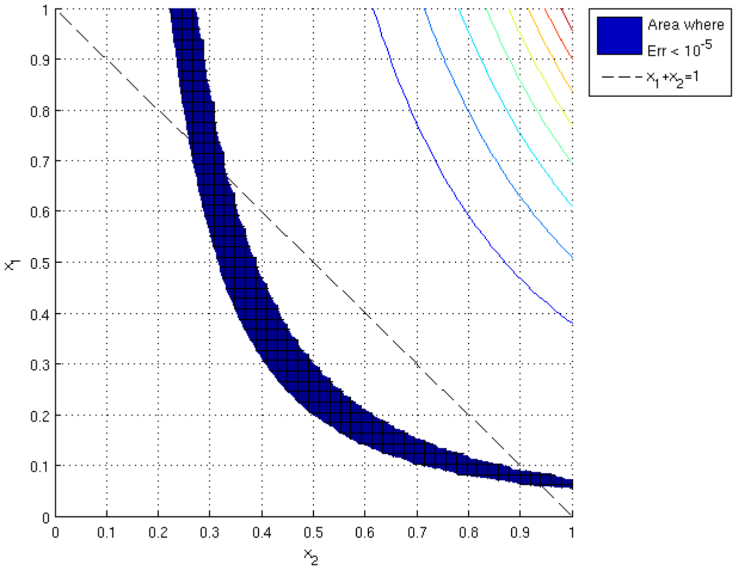


Figure 4.4: Example plot of the error function against probabilities $[x_1, x_2]$ for Example 4.5.

4.3.5 True/False Positive/Negatives in SLPs

Recall the measures defined in Section 3.3 in the context of ProbFOIL. We adapt them to the context of SLP semantic, in the search of a suitable local score function for learning SLPs.

Let us assume that $p(e_i)$ is the observed probability of an example and $p(e_i|S)$ is the predicted probability of an example. We distinguish between the following cases, keeping in mind that negative examples have 0 probability:

- $p(e_i) \neq 0$, $p(e_i|S) \neq 0$, then the *over-explained part* of e_i is $max(0, p(e_i|S) - p(e_i))$, the *under-explained part* of e_i is $max(0, p(e_i) - p(e_i|S))$, and the *true positive part* of e_i is $min(p(e_i), p(e_i|S))$.
- $p(e_i) \neq 0$, $p(e_i|S) = 0$, then the *false negative part* of e_i is $p(e_i)$.
- $p(e_i) = 0$, $p(e_i|S) \neq 0$, then the *false positive part* of e_i is $p(e_i|S)$.
- $p(e_i) = 0$, $p(e_i|S) = 0$, then the *true negative part* of e_i is 1.

We define these measures over the whole example set E as the sum of the corresponding measure for each example. Due to time constraints, we haven't developed any IR scores such as precision or recall, but we believe that further investigation in this direction might lead to useful statistics about the learned hypothesis, which may be used to guide the ILP search process, thus combining structure and parameter learning.

4.4 Conclusions

It would be valuable to evaluate our system in comparison with the SLP learning framework [Muggleton, 2002] and with ProbFOIL [Raedt and Thon, 2010]. Unfortunately, it is impossible to compare ProbPoly with ProbFOIL due to different semantics: the SLP semantics and the distribution semantics. In the current implementation of ProbPoly, it is also impossible to compare our results with those obtained in [Muggleton, 2002]. Recall from Section 3.2 that the approach used there aims to maximize the product of the examples, while our approach minimizes the difference between the observed and predicted probabilities over all the examples. Even if we set all the probabilities of the non-probabilistic examples in SLP learning to 1, this cannot ensure that the product of the learn probabilities is maximal. We could consider our error function of the form $-p(E|S)$ ($p(E|S)$ is the score defined in Section 3.2), but this would lead to an increase of the degree of the error function polynomial to a maximum of $|E| \cdot 2 \cdot (\max_i max\_deg\_goal(goal_i))$, where $i \in 1, \ldots, |E|$. In fact, this is the advantage of ProbPoly over SLP learning: in SLP learning the complexity increases with the number of examples, while in ProbPoly the complexity increases with the number of parameters learned and with the recursion depth, keeping in mind the fact that SLP learning always learns one probability, and doesn't allow recursion.

To conclude, in this chapter we have introduced a new method for learning SLPs with probabilistic examples. Starting from the non-recursive case considered in [Muggleton, 2002] with non-probabilistic examples, we have adapted it to probabilistic examples, and further extended our approach to recursive SLPs, SLPs with negation as failure, and finally, we have succeeded in learning probabilities for the context in which we learn more than one clause at a time. This completes the theoretical presentation of ProbPoly. In Chapter 5 we will consider implementation issues, and in Chapter 8 we discuss essential directions for improving and/or extending the framework.

Chapter 5

Implementation

5.1 The MatLab Package Interface

The need for numerical solutions in our approach implies using both Prolog and MatLab. Unfortunately, this is feasible only in the YAP system, where we have the MATLAB Package Interface. In this manner, we can perform numerical tasks in MatLab, without thinking of how to implement them efficiently in Prolog. For example, consider the problem of finding the value of binomial coefficients $C(n, k)$ for a given n and k, which is necessary when computing $(1 - x)^n$.

Instead of computing the binomial coefficients in Prolog, via the predicate given in Listing A.4, we simply call the (more efficient) MatLab function, via the MATLAB Package Interface, as shown in Listing A.5. Note that *ml_init*/0, given in Listing A.3, simply initializes MatLab, and the operator $< --$ denotes a call to MatLab function on the right-hand-side, with the result in the left-hand-side.

Unfortunately, due to the limitations of the MatLab Package Interface, it is a little difficult to pass complex structures from Prolog, or evaluate MatLab commands of great length. We overcome these difficulties by generating MatLab code in Prolog, writing it into MatLab files (scripts or functions), and then calling the scripts or functions.

5.2 PSOpt

PSOpt is a MatLab toolbox for Particle Swarm Optimization, and its main advantages over other variants is that it allows constraints to be specified in a format compatible with the rest of the functions in the Optimization toolbox.

The reason we need Particle Swarm Optimization is that GloptiPoly cannot always ensure global optimality, and worse than that, in particular circumstances it will not return any solution. We find it unacceptable to simply give up in such a circumstance, so we trade-off the global optimality condition and the efficient methods of GloptiPoly for the certainty of finding a solution.

Particle Swarm Optimization (PSO), proposed in [Kennedy and Eberhart, 2002], is a meta-heuristic, or a method of evolutionary computation, which aims at minimizing a parametrized function. Unlike algorithms such as randomized search, gradient descent with repeated restarts, simulated annealing, or even genetic algorithms, PSO uses a strategy which often leads to very fast convergence.

The Basic PSO Algorithm is:

Start with a population of random parameters (or generated using a heuristic). Each individual or particle of the population is characterized by a position vector (which is given by the values of the parameters) and a velocity vector. Both are updated over all individuals in the population, and a discrete time steps are considered in the evolution of the population. The algorithm terminates based on a convergence criterion, e.g. the value of the best solution hasn't improved with more than ε in the last x time steps.

The update rules for each individual i are:

$v_i(k + 1) = v_i(k) + \gamma_{i1}(p_i - x_i(k) + \gamma_{i2}(G - x_i(k))$, and

$x_i(k+1) = x_i(k) + v_i(k+1)$,
where k is a discrete time index, x is the position of the particle, v is the velocity, p_i is the best value found by particle i in its history, G is the global best (i.e. over the whole population) and γ_{i1}, γ_{i12} are random numbers between $[0,1]$ generated for each particle.

The intuition behind the update rule for velocity is that $v_i(k)$ represents inertia, which may enable the particle to overcome local minima (the same idea is used in some versions of weight updates in artificial neural networks), the second term, $\gamma_{i1}(p_i - x_i(k)$, ensures local exploitation, meaning that particles are attracted to the best value of the objective function they have found, while the third term, $\gamma_{i2}(G - x_i(k)))$, ensures exploration (it may draw particles away from being stuck in a personal best minimum) as well as exploitation, since particles are attracted to the best solution found in the entire population.

To conclude, even though Particle Swarm Optimization doesn't guarantee global optimality, it will always produce a solution, and it is believed to converge faster than other metaheuristics.

5.3 Architecture

In this section we give an overview of the current architecture of the ProbPoly implementation. In Figure 5.1 we show the main components used by our application, and their interaction.

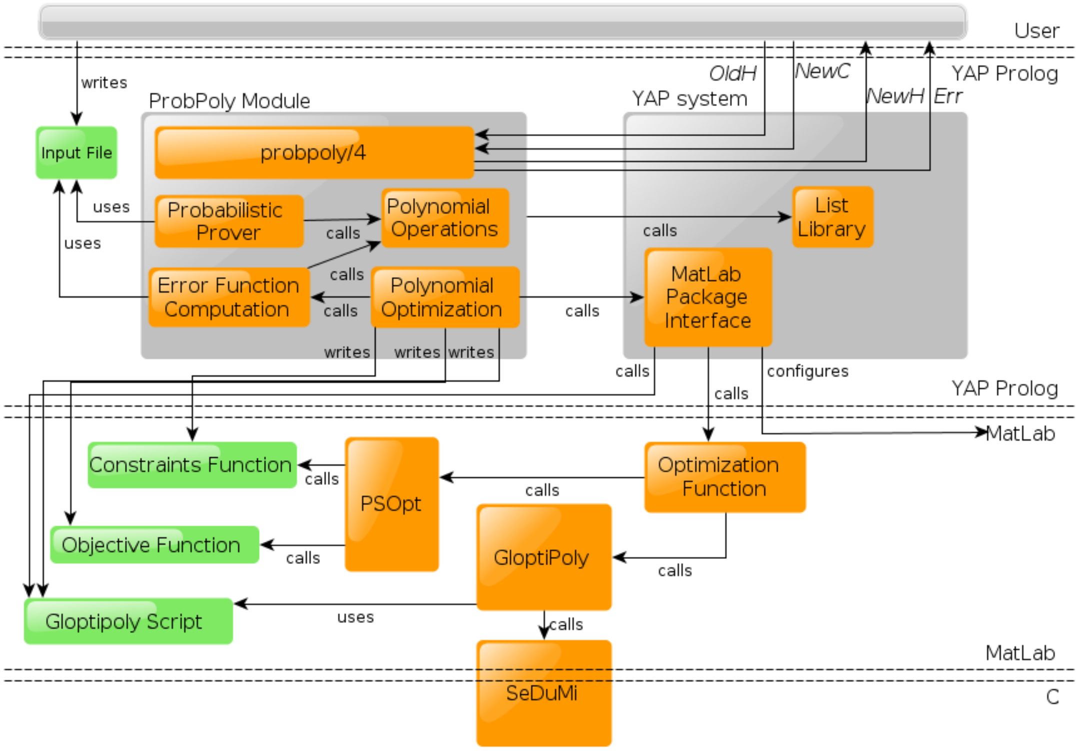


Figure 5.1: General overview of the ProbPoly architecture.

The most intuitive way to understand the logic of the architecture is to analyse it from the top-level, that is from the user's perspective. The user provides an input file to the system, which is a Prolog file containing the background knowledge and the examples. The main predicate, *probpoly*/4, takes as input a set of probabilistic clauses, either learned from a previous run or known a-priori, $OldH$, and a set of non-probabilistic clauses, $NewC$, whose probabilities must be learned. The output presented to the user is the new theory consisting of $NewH = OldH \cup NewC'$, where $NewC'$ are the clauses $NewC$ annotated with probabilities, and the value Err of the error function of $NewH$. The predicate code is shown in Listing A.1.

The ProbPoly module computes a polynomial of total probability for each example using the prover and predicates defining operations on polynomials. The polynomials, together with the probabilities of the examples are then used to compute the error function, which is also a polynomial. Then, this polynomial, and the constraints discussed in Section 4.3 when learning multiple clauses are sent to MatLab. For Gloptipoly, we generate a MatLab script which creates the Gloptipoly objects needed as input for the solver:

- the polynomial object,
- the objective function, i.e. the error function
- the list of constraints

An example is given in Listing A.7.

For PSOpt, we generate two MatLab function files, specifying the objective function, e.g. Listing A.8 and the list of constraints A.9.

After the generation of these files, we start and set up MatLab[1] using the MatLab Package Interface available in YAP. Then, we call a function which tries to find the global minima with GloptiPoly, and in case of failure, uses the particle swarm optimization toolbox PSOpt to search for global minima. We show the function in Listing A.10.

GloptiPoly uses SeDuMi, "a software package to solve optimization problems over symmetric cones. This includes linear, quadratic, second order conic and semidefinite optimization, and any combination of these"[2], written in MatLab and C.

We believe that the current architecture is a good design choice since we can solve tasks of symbolical computation in Prolog, and we can use MatLab to handle numerical problems.

5.4 The Impact of Negation as Failure

Allowing negation as failure means that in the derivation of a goal, we may have to compute the total probability of a negated goal. This forces the representation of the probability of a goal as a polynomial.

If we don't incorporate negation as failure, and consider single clause learning, we can represent the probability of a goal as a tuple:
$[c, n, m]$, corresponding to
$c * x^n * (1 - x)^m$.

This way, during the proof we simply collect $[c, n, m]$ and when we compute the total probability of a goal we use Theorem 2.2 for $(1 - x)^m$.

A similar situation occurs in the case of multiple clause learning. We can represent the probability of a goal as a tuple:
$[c, p_1, \ldots, p_k, p_{k+1}]$, corresponding to
$c * x_1^{p_1} \ldots x_k^{p_k} * (1 - x_1 - ... - x_k)^{p_{k+1}}$.

Again, during the proof we collect only $[c, p_1, \ldots, p_k, p_{k+1}]$ and when evaluating the total probability of a goal we use Theorem 2.3 to compute $(1 - x_1 - ... - x_k)^{p_{k+1}}$.

[1] The most important operation is to add the MatLab folder and its subfolders to the MatLab path.

[2] Quote from the FAQ of SeDuMi (http://sedumi.ie.lehigh.edu/ – accessed 08.09.2011)

Part II

Towards Learning Probabilistic Requirements

Chapter 6

Related Work in Learning Requirements

It is essential to have an overview of the state-of-the-art systems used in modelling and learning requirements, in order to be able to develop a solution for models which incorporate probabilities. In Section 6.1 we briefly present a system which uses ILP and a model checker in a learner-teacher paradigm. A similar approach, which uses artificial neural networks adapted for linear temporal logic instead of ILP, is described in Section 6.2. Automated verification and learning assumptions for models is the topic of Section 6.3, and the studied system uses the L* algorithm as a learner, with counterexamples provided by a model checker. Finally, we review two robust frameworks which have been developed for a long time, the KAOS and i* approaches to modelling and learning requirements in Sections 6.4 and 6.5. KAOS and i* don't take into account uncertainty in the form of probabilities, so it is important to study them to research the possibility of integrating probabilistic reasoning in their models and methods.

6.1 Learning Requirements using ILP and Model Checking

In [Alrajeh et al., 2006], a novel method of learning requirements from scenarios is presented. The problem is to automate the process of requirement elicitation, which is a crucial part of the requirements engineering process. The raw input represents a narrative which partially describes the requirements of the *system-to-be.* This is processed into a formal representation, which relies on Linear Temporal Logic (LTL) to define *requirements specification Spec* as a LTL theory consisting of axioms(two initial state axioms, two persistence axioms and two change axioms) and *event precondition axioms.* A *scenario* is a specific type of an LTL formula. Scenarios can be desirable or undesirable. The solution to the problem become finding a set of event precondition axioms Pre such that all desirable scenarios are true and all undesirable scenarios are false:

- $Spec \cup Pre \models_M \neg P_u$, for each undesirable scenario $P_u \in Und$.
- $Spec \cup Pre \not\models_M \neg P_d$, for each desirable scenario $P_d \in Des$.

Und and Des are the sets of undesirable and desirable scenarios. M is an LTL model of the form $\langle T, V \rangle$, where T is a *Label Transitioning System (LTS)*, and V is a *valuation function.*

In order to accomplish this, specifications and scenarios are translated using an Event Calculus (EC) framework into normal logic programs. Two transformation methods are presented: one for the specification, and one for the specification together with the sets of desirable and undesirable scenarios. The result is a background knowledge B (which includes four EC core axioms), and a set of positive and negative examples E. This is the standard input to an ILP algorithm, and in the system described XHAIL (Extended Hybrid Abductive and Inductive Learning) is applied in order to learn hypotheses H. These are shown, for a case study of a mine pump control system, to correspond to the correct event precondition axioms.

The work in [Alrajeh et al., 2009b] extends the framework presented in the previous subsection (based on [Alrajeh et al., 2006]). The main difference represents the fact that the latter is able to handle trigger axioms, as well as precondition axioms in the LTL formalism of defining a specification, and thus the newer system is able to learn trigger conditions. This is not a trivial extension of the previous system, and in what follows we will briefly present the changes this extension implies. First of all, since we need a new predicate, called "triggered", we must define additional EC core axioms which describe properties of triggered events, e.g. if an event is triggered at one point in time, then all other events must be impossible at that point[1]. The specification LTL theory is enriched with *trigger-condition axioms*, similar in structure with the *pre-condition axioms*, denoted in [Alrajeh et al., 2006] as event precondition axioms. The way to define positive and negative scenarios is also slightly different.

Since the solution will consist of trigger-condition axioms, in addition to pre-condition axioms, we will have to find the sets of axioms Pre and $Trig$ such that $Spec \cup Pre \cup Trig \models_M \neg P_u$ and $Spec \cup Pre \cup Trig \not\models_M \neg P_d$. The translation into an EC normal logic program is naturally extended to $Trig$ axioms. Some theoretical results are also presented (Theorems 3.1 and 3.2), one (3.2) suggesting that we may perform the inverse translation of H into LTL formulas corresponding to axioms (due to the inverse of the translation function τ^{-1}). Also, the examples are fewer than in [Alrajeh et al., 2006], and more hypotheses are generated, both correct with respect to the learning task. It is up to the engineer to choose one more appropriate, or more easily to interpret.

The system proposed in [Alrajeh et al., 2009a] represents a further improvement of the previous work in [Alrajeh et al., 2006] and [Alrajeh et al., 2009b]. The essential novelty is the fact that counterexamples are generated by using the LTSA model checker (in a semi-automated process). First of all, the input changes: we now have only a partial specification which needs to be refined, and a set of goals such that the learned specification will be able to satisfy all the given goals. This is only a semi-automated procedure because, based on the trace given by the model checker, we still have to manually develop a negative scenario. In the same manner, positive scenarios are elaborated, in the case that the model checker doesn't return a goal violation. These, as usual, will be translated into a normal logic program, which is fed to the ILP learner. The result of the ILP module is a set of hypotheses which are finally translated back into requirements from which the engineer selects the adequate ones (although all are formally correct from the ILP perspective) to be added to the specification.

6.2 Connectionist Systems for Learning Requirements

In [Borges et al., 2010b], the authors propose a new framework for knowledge representation, reasoning and learning, which is applied to the task of learning requirements from scenarios, inspired from [Alrajeh et al., 2009b]. The premises and goals are essentially the same, and the case study and main example of the article is the mine pump system, but the ILP part is replaced with a connectionist system. In this connectionist system, illustrated in Figure 6.1 (reproduced from figure 1 in [Borges et al., 2010b]), the *model description* is translated into a neural network, which is trained using examples from the *observed system* and from the *specified properties*. The next step involves extracting knowledge from the trained network in the form of *state diagrams*, which can be translated into logic programs.

[1] EC core axiom (6) in [Alrajeh et al., 2009b].

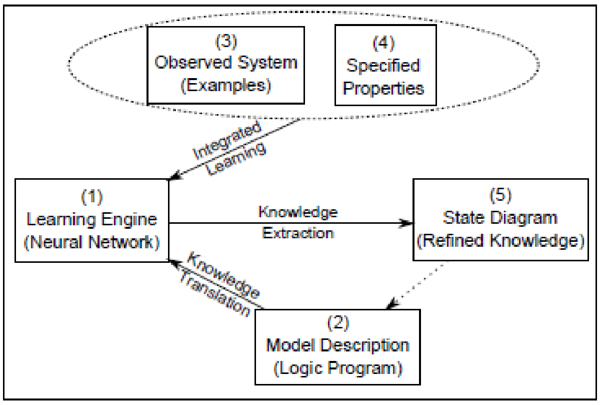


Figure 6.1: Learning framework for the connectionist system.

The representation of knowledge is based on Linear Temporal Logic (without event calculus, unlike [Alrajeh et al., 2009b]). The inputs of the network are states or inputs of the system, and the main role of the network is to learn given states $s_1, s_2, ..., s_n$ and inputs $i_1, i_2, ..., i_m$, the temporal logic next state operator $\bigcirc$, i.e. $\bigcirc s_1, \bigcirc s_2, ..., \bigcirc s_n$. The input nodes of the network consist of a corresponding node for each state and input, respectively, and the output nodes represent $\bigcirc s_1, \bigcirc s_2, ..., \bigcirc s_n$. If the model description consists of rules of the type: $\bigcirc s_i \leftarrow s_1, s_2, ..., s_k, i_1, i_2, ..., i_l$, where the states or inputs may be negated, then for each such rule a hidden neuron is added to the network, with inputs connections from $s_1, s_2, ..., s_k, i_1, i_2, ..., i_l$ and output connection to $\bigcirc s_i$. The idea resembles the core method translation algorithm for networks simulating propositional logic programs (the algorithm can be found in the proof of Theorem 3.2 in [Hitzler et al., 2004]).

The training using input/output patterns is simple, however in the case where learning is accomplished by presenting properties to the system, the authors propose a method of defining the output of the network based on a list of active properties, which is in turn determined by the values of the current states and inputs. The example given for this step however is not completely clear. After training, knowledge is extracted in a greedy manner: each tuple $\langle s_1, s_2, ..., s_n; i_1, i_2, ..., i_m; \bigcirc s_1, \bigcirc s_2, ..., \bigcirc s_n \rangle$ appearing in the network increases the count of the same transition, and transitions are filtered according to their counts. Each transition can be then rewritten into a set of rules to obtain a logic program.

The experiments carried out on the mine pump example suggest that the framework is successful in learning in the absence of background knowledge, and the authors argue that their method is a viable alternative to the one proposed in [Alrajeh et al., 2009b] because the connectionist system is able to correct eventual errors in the initial description.

An extension of this system, described in [Borges et al., 2010a] incorporates the NuSMV model checker, probably inspired by the framework in [Alrajeh et al., 2009a], An important idea is that the initial knowledge may or may not be incorporated in the adaptation process. In the case when initial knowledge is given, it is represented as a NuSMV model, which is translated into a logic program, and finally, the temporal logic program is used to create a neural network, via the Sequential Connectionist Temporal Logic tool. Otherwise, the system will adapt based on the observed system behaviour, only through examples fed into the network. After the network is trained, its output representing a symbolic model is verified against the model checker. If there are any properties not satisfied by the model, at least one counterexample is generated, and this can be used in a similar way to examples for the neural network to adapt. The framework is shown in Figure 6.2, which is reproduced from figure 1 in [Borges et al., 2010a].

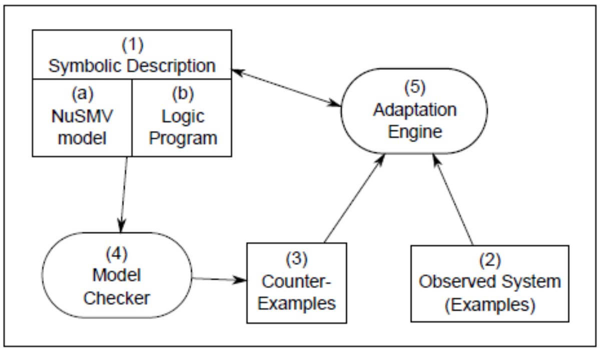


Figure 6.2: Learning framework for the connectionist system incorporating the NuSMV model checker.

The process of adaptation to counterexamples is very similar to learning from properties: based on sequences of states and inputs, the system determines which inputs would generate property violations at the next state, and doesn't allow these inputs to activate in the network. Knowledge extraction is performed in a similar way as in the previous framework. The main example remains the mine pump system, and it is reported that even without initial knowledge, the connectionist system can be successfully trained and adapted to learn the correct model.

6.3 Automated Verification of Systems using L*

6.3.1 The L* Algorithm

In [Angluin, 1987], the L^* algorithm for learning regular sets is proposed. The algorithm is the Learner in a Teacher-Learner framework, because it depends on information given by a (minimally adequate) teacher regarding membership queries and conjectures. The teacher must be able to answer whether a string is a member of the set or not (i.e. answer to a membership query), and must be able to compare the learned set S (i.e. make a conjecture) with the unknown language, and its answer must be yes in the case they are equal, or no otherwise. In the latter case, the teacher must also give a counterexample, that is a string which is in S but not in the unknown language.

L^* relies on an *observation table* (S, E, T), consisting of a set of candidate strings S, a set of distinguishing experiments E and a mapping $T : (S \cup S \cdot A) \cdot E \to \{0, 1\}$, where A is the alphabet over which the strings are defined. By $row(s)$ we will denote $T(s \cdot e)$. Based on an observation table (S, E, T) we define a deterministic finite-state acceptor $M(S, E, T)$:

$$Q = \{\mathrm{row}(s) \colon s \in S\},$$

$$q_0 = \mathrm{row}(\lambda),$$

$$F = \{\mathrm{row}(s) \colon s \in S \text{ and } T(s) = 1\},$$

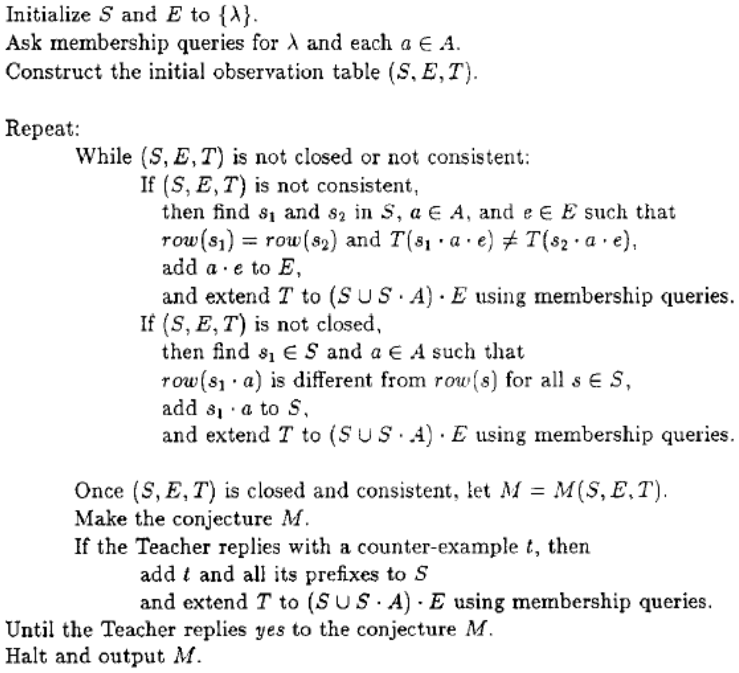
$$\delta(\mathrm{row}(s), a) = \mathrm{row}(s \cdot a).$$

, where Q is the state set, q_0 is the initial state, F are the accepting states, and δ is the transition function.

Two properties of observation tables are relevant: closure and consistency. An observation table is closed if for each t in $S \cdot A$ there exists a state s in S such that $row(s) = row(t)$. An observation table is consistent if for states s_1 and s_2 in S, and $row(s_1) = row(s_2)$, then for all a in A we have $row(s_1 \cdot a) = row(s_2 \cdot a)$.

We now present the L^* algorithm in Figure 6.3 (Figure 1 in [Angluin, 1987]). After initialization, in which the candidate states S and the distinguishing strings E are both the null string λ, the main loop of the algorithm has three important parts:

- assure that the observation table (S, E, T) is closed consistent, which is performed using membership queries.
- define, as described above, the deterministic finite-state acceptor $M = M(S, E, T)$.
- make the conjecture M, and if the teacher replies with no, update the observation table according to the counterexample provided, otherwise halt and return M.



```
Initialize $S$ and $E$ to $\{\lambda\}$.
Ask membership queries for $\lambda$ and each $a \in A$.
Construct the initial observation table $(S, E, T)$.

Repeat:
    While $(S, E, T)$ is not closed or not consistent:
        If $(S, E, T)$ is not consistent,
          then find $s_1$ and $s_2$ in $S$, $a \in A$, and $e \in E$ such that
          $row(s_1) = row(s_2)$ and $T(s_1 \cdot a \cdot e) \neq T(s_2 \cdot a \cdot e)$,
          add $a \cdot e$ to $E$,
          and extend $T$ to $(S \cup S \cdot A) \cdot E$ using membership queries.
        If $(S, E, T)$ is not closed,
          then find $s_1 \in S$ and $a \in A$ such that
          $row(s_1 \cdot a)$ is different from $row(s)$ for all $s \in S$,
          add $s_1 \cdot a$ to $S$,
          and extend $T$ to $(S \cup S \cdot A) \cdot E$ using membership queries.

    Once $(S, E, T)$ is closed and consistent, let $M = M(S, E, T)$.
    Make the conjecture $M$.
    If the Teacher replies with a counter-example $t$, then
        add $t$ and all its prefixes to $S$
        and extend $T$ to $(S \cup S \cdot A) \cdot E$ using membership queries.
Until the Teacher replies yes to the conjecture $M$.
Halt and output $M$.
```

Figure 6.3: The L^* algorithm.

An important practical issue with the algorithm is the verification of the conjecture. This query might prove infeasible in general languages. To overcome this problem, the author proposes a different method to perform the conjecture query: assuming there is a probability P on the set of all strings of alphabet A, then the query will consist of sampling a number of strings from A according to P, and if any of the sampled strings are in the unknown language U and not accepted by $M(S, E, T)$ or vice versa, then the answer is no, and the counterexample is the corresponding string.

We will briefly cover an improvement of L^* for the case in which we are not allowed to "reset" the finite-state-machine, i.e. we are not allowed to execute sequences from the initial state, but only from the current state. The algorithm is described in [Rivest and Schapire, 1989] and relies on two important ideas. The first one is that the finite state automaton has a finite number of equivalence classes, defined by the equivalence relation $t1 \equiv t2$, which means sequences $t1$ and $t2$ have the same value at every state.

The second idea is to define a *homing sequence* as "an action sequence h for which the state reached by executing h is uniquely determined by the output produced"[2]. The output produced means the vector of values with the same length as h, where the i-th component means the value after executing the prefix of length i of h. The algorithm will only execute homing sequences, and based on their output, it will create a separate L^* learner, or if one already exists, it will execute the next query. By using a different learner for each output, we essentially simulate L^* for multiple starting states (corresponding to the ones defined by the output of executing the homing sequence).

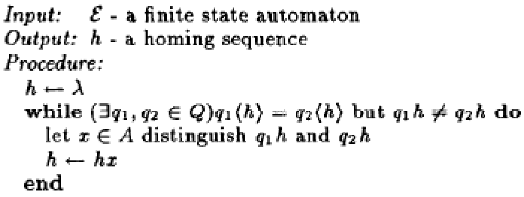
In Figure 6.4[3] we reproduce the algorithm for finding a homing sequence given a finite state automaton, and in Figure 6.5[4] we show the algorithm for inferring the finite state automaton given

[2] Definition 2 in [Rivest and Schapire, 1989].

[3] Figure 3 in [Rivest and Schapire, 1989].

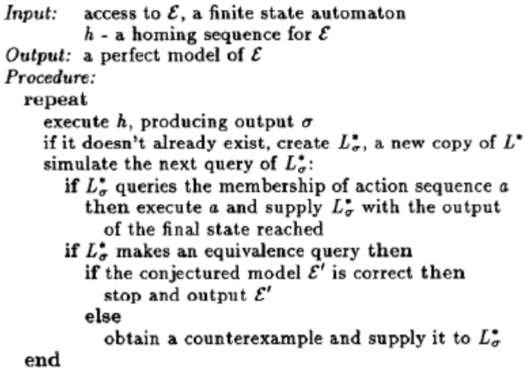
[4] Figure 5 in [Rivest and Schapire, 1989].

access to it, and a homing sequence. The authors also provide a slightly more complicated variant of the algorithm, in which the homing sequence is constructed simultaneously with the inference procedure. They also propose algorithms for stochastic inference (infer U with probability $1 - \delta$), algorithms which use explicitly the equivalence classes of the finite state automaton and similar inference algorithms for permutation automata.



```
Input:    $\mathcal{E}$ - a finite state automaton
Output:   $h$ - a homing sequence
Procedure:
  $h \leftarrow \lambda$
  while $(\exists q_1, q_2 \in Q) q_1\langle h \rangle = q_2\langle h \rangle$ but $q_1 h \neq q_2 h$ do
    let $x \in A$ distinguish $q_1 h$ and $q_2 h$
    $h \leftarrow hx$
  end
```

Figure 6.4: Algorithm for finding a homing sequence.



```
Input:    access to $\mathcal{E}$, a finite state automaton
          $h$ - a homing sequence for $\mathcal{E}$
Output:   a perfect model of $\mathcal{E}$
Procedure:
  repeat
    execute $h$, producing output $\sigma$
    if it doesn't already exist, create $L^*_\sigma$, a new copy of $L^*$
    simulate the next query of $L^*_\sigma$:
      if $L^*_\sigma$ queries the membership of action sequence $a$
        then execute $a$ and supply $L^*_\sigma$ with the output
          of the final state reached
      if $L^*_\sigma$ makes an equivalence query then
        if the conjectured model $\mathcal{E}'$ is correct then
          stop and output $\mathcal{E}'$
        else
          obtain a counterexample and supply it to $L^*_\sigma$
  end
```

Figure 6.5: Algorithm for inferring a finite state automaton using a homing sequence.

6.3.2 The Original Framework

In this subsection we will discuss the framework in [Păsăreanu et al., 2008] which aims at verifying systems in an automatic way, using L^* as a learner and the LTSA model checker as part of the teacher. The fundamental principle is to verify that all the system components satisfy some properties under certain assumptions, and use parallel composition to prove that the safety property holds for the whole system.

Let us introduce a few notions. A *labelled transition system* (LTS) is a four-tuple $M = \langle Q, \alpha M, \delta, q_0 \rangle$, where Q is the set of states, αM is the alphabet of the LTS, δ is the transition relation consisting of a source state, an action, and a destination state, and q_0 is the initial state. The *parallel composition* of two labelled transition systems $M_1 \parallel M_2$ is also an LTS. A *safety* LTS is a LTS which doesn't contain an error state. A *safety property* is "a safety LTS P whose language $\mathcal{L}(P)$ defines the set of acceptable behaviors over αP". Based on a property, we define an *error LTS*, P_{err}, which extends P so that all the transitions which are not in the LTS lead to an error state. The purpose of this is to detect violations of the safety property. Finally, an assume guarantee formula is a triple $\langle A \rangle M \langle P \rangle$ in which M is a component (in this case an LTS), P is a property (as defined above), and A is an assumption about the environment of M (typically a conjecture learned by L^*).

An important theoretical result is Thoerem 1 of [Păsăreanu et al., 2008], which states that: $\langle A \rangle M \langle P \rangle$ is true if and only if the error state is unreachable from $A \parallel M \parallel P_{err}$. The assume guarantee formulas are used to make rules which guarantee the satisfaction of a property by the system. In Figures 4 and 5 (reproduced from [Păsăreanu et al., 2008]) we have two such rules. The first rule, ASYM, states that if the property A should hold for a component, and if the other component satisfies property P under the assumption of A, then the system satisfies P. This rule can be generalized such that it incorporates circularity between n components (it is described as rule CIRC-N in [Păsăreanu et al., 2008]). The second rule states that if each component satisfies

P under its own assumption, and if the languages of the complements (coA_i) of the assumptions are a subset of the language of P, then the system satisfies P. The complement of an LTS M is an LTS which accepts the complement of M's language.

$$\begin{array}{l} 1 : \langle A \rangle M_1 \langle P \rangle \\ 2 : \langle true \rangle M_2 \langle A \rangle \\ \hline \quad \langle true \rangle M_1 \parallel M_2 \langle P \rangle \end{array}$$

Figure 6.6: Rule ASYM.

$$\begin{array}{ll} 1 : & \langle A_1 \rangle M_1 \langle P \rangle \\ 2 : & \langle A_2 \rangle M_2 \langle P \rangle \\ \vdots & \\ n : & \langle A_n \rangle M_n \langle P \rangle \\ n+1 : & \mathcal{L}(coA_1 \parallel coA_2 \parallel \cdots \parallel coA_n) \subseteq \mathcal{L}(P) \\ \hline & \langle true \rangle M_1 \parallel M_2 \parallel \cdots \parallel M_n \langle P \rangle \end{array}$$

Figure 6.7: Rule SYM.

To complete the presentation, we show an algorithm for rule ASYM in Figure 6.8 (Figure 4 from [Păsăreanu et al., 2008]). The top box of the Teacher represents the membership query to L^*, and the three bottom boxes of the Teacher represent the conjecture query. In the latter case, the oracles illustrate methods to check the premises of ASYM. Notice that if the second premise is false, the counterexample is not immediately returned to L^*, instead further counterexample analysis is performed, and if the counterexample is valid, than a real error trace is returned to the user, otherwise the Teacher returns to L^* the correct conjecture.

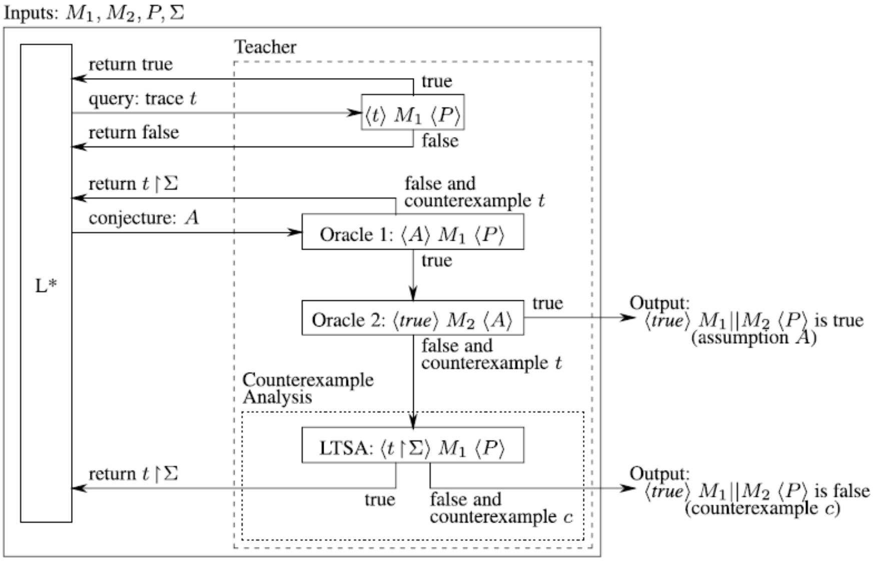

Inputs: M_1, M_2, P, Σ
Teacher
return true
true
query: trace t
$\langle t \rangle\ M_1\ \langle P \rangle$
return false
false
return $t \restriction \Sigma$
false and counterexample t
conjecture: A
Oracle 1: $\langle A \rangle\ M_1\ \langle P \rangle$
true
L*
Oracle 2: $\langle true \rangle\ M_2\ \langle A \rangle$
true
Output: $\langle true \rangle\ M_1 || M_2\ \langle P \rangle$ is true (assumption A)
false and counterexample t
Counterexample Analysis
LTSA: $\langle t \restriction \Sigma \rangle\ M_1\ \langle P \rangle$
return $t \restriction \Sigma$
true
false and counterexample c
Output: $\langle true \rangle\ M_1 || M_2\ \langle P \rangle$ is false (counterexample c)

Figure 6.8: Alogrithm for rule ASYM.

The reason we are interested in L^* and more importantly the application briefly described is that extracting requirements is similar to system verification in the sense that the property to be satisfied represents the fact that the learned requirements must not violate system constraints. Moreover, both techniques have made use of a model checker, with the same goal of generating counterexamples, however L^* has not been tried as a method to learn requirements, and this provides a novel direction of research.

6.3.3 Verification in a Probabilistic Context

The probabilistic extension of assume-guarantee reasoning is introduced in [Feng et al., 2011]. The same authors developed PRISM - a probabilistic model checker which is used to verify probabilistic properties. We will describe probabilistic model checking, its properties and models in the next chapter, however we exemplify a few properties, since they are used to parametrize assumptions. In a probabilistic context we can verify more expressive properties such as[5]:

- security properties like: "the probability of an airbag failing to deploy within 0.02 seconds is at most 0.0001",
- timeliness properties like: "expected time for a successful transmission of a data packet"

Properties of the type G: $\Box\neg fail$ become $\langle\Box\neg fail\rangle_{\geq 0.98}$. Adversaries (also known as strategies, schedulers, policies) are possible ways to incorporate non-determinism in probabilistic models, and are generally defined as functions which given a state, produce an action.

Let us assume a model like an LTS, with the difference that once an action is chosen, in a state, the next state is determined probabilistically. A model M satisfies property $\langle G\rangle_{\geq p_G}$ ($M \models \langle G\rangle_{\geq p_G}$) if $Pr^{\sigma}_{M}(G) \geq p_G$ for all adversaries σ. The assume guarantee triple becomes $\langle A\rangle_{\geq p_A} M_i \langle G\rangle_{\geq p_G}$, with M_i a model, and $\langle A\rangle_{\geq p_A}$, $\langle G\rangle_{\geq p_G}$ are safety properties whose extension would lead to bad traces.

The key idea of the learning process is to reduce the probabilistic checking of properties to the problem of learning a non-probabilistic assumption, by exploiting properties of the adapted ASYM rule and using a variant of the L* algorithm.

We leave out the technical details because we will define probabilistic models in the next chapter, and, for our purposes, the important aspect of this system is the use of a probabilistic model checker in conjunction with a learner to generate properties of probabilistic models.

6.4 The KAOS framework and related methods

6.4.1 Introduction and Inference of Requirements Specifications from Scenarios

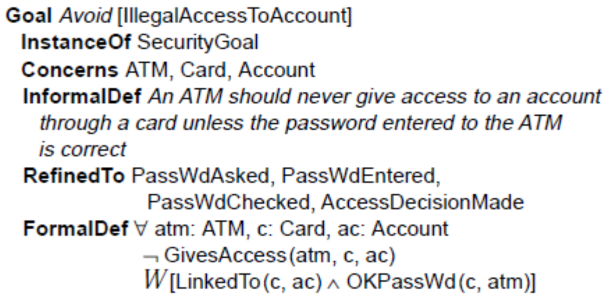
KAOS is a methodology which enables the modelling of requirements and automatic generation of requirements based on goals or scenarios[6]. This brief introduction to KAOS and the rest of the subsection is based on [van Lamsweerde and Willemet, 1998], which shortly describes the KAOS system, starting from its ontology:

- *Object* - can be an entity, relationship or event, characterized by attributes and invariant assertions.
- *Operation* - input/output relation between objects, characterized by pre-, post- and trigger-conditions. The pre- and post-conditions are classified into domain and required condition, the former being general constraints, while the latter capture additional conditions.
- *Agent* - a type of object which is allowed to perform certain operations on other objects.
- *Goal* - an objective the system should meet. Goals are usually refined in AND/OR graphs (a goal is satisfied if all the sub-goals are satisfied in an AND-refinement, or it is satisfied if at least one sub-goal is satisfied in an OR-refinement). Goals involve certain Objects, and may conflict with each other.
- *Requisite, requirement, assumption.* A requisite is a goal which can be controlled by an individual agent. Refinements must transform the initial goal into a set of requisites. Requirements are requisites assigned to software agents, while assumptions are requisites assigned to environmental agents.

[5] Examples are reproduced from [Feng et al., 2011].

[6] For an extensive list of publications see: http://www.info.ucl.ac.be/~avl/ReqEng.html (accessed 08.09.2011).

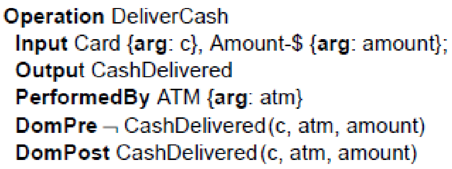
In what follows, we will give examples of some of these concepts. In Figure 6.9 (reproduced from section 2.1.2) we can see a goal specification. We notice object instantiation, the objects involved in the goal definition, the informal definition, which is implemented in the system in the context of a semantic net, the goals which refine the initial goal (it is the case of an AND-refinement), and the formal definition, which expresses the same idea as the informal one, but in the syntax of temporal logic.



Goal *Avoid* [IllegalAccessToAccount]
InstanceOf SecurityGoal
Concerns ATM, Card, Account
InformalDef *An ATM should never give access to an account through a card unless the password entered to the ATM is correct*
RefinedTo PassWdAsked, PassWdEntered, PassWdChecked, AccessDecisionMade
FormalDef ∀ atm: ATM, c: Card, ac: Account
¬ GivesAccess(atm, c, ac)
$\mathcal{W}$ [LinkedTo(c, ac) ∧ OKPassWd(c, atm)]

Figure 6.9: Goal specification.

For an example of an operation we refer to Figure 6.10[7]. Notice the input instances of objects ”Card” and ”Amount-$”, and the output object ”Cash Delivered” (of type event). It was established that operations are performed by agents, so the definition must specify it, in this case, an instance of the ”ATM” object. Finally, the domain pre- and post-conditions are shown. However, taking into account the goal specification, we also need to add required pre-post conditions. A possible pre-condition would be to check if the password is entered correctly.



Operation DeliverCash
Input Card {**arg**: c}, Amount-$ {**arg**: amount};
Output CashDelivered
PerformedBy ATM {**arg**: atm}
DomPre ¬ CashDelivered(c, atm, amount)
DomPost CashDelivered(c, atm, amount)

Figure 6.10: Definition of an Operation.

In what follows, we will briefly present the KAOS specification elaboration method starting from high-level goals, and some aspects of the goal-inference procedure from scenarios. The method of specification elaboration consists of several steps:

1. *Goal elaboration* - this steps involves goal refinement until requisites, as previously defined, are reached.

2. *Object and operation capture* - identifies relevant objects and operations to the goal formulations.

3. *Operationalization* - generates additional conditions for operations in order to satisfy the requisites obtained after goal elaboration.

4. *Responsibility assignment* - this final step is concerning the management of requisites, their distribution among agents, solving conflicts caused by the different goals of the stakeholders etc.

This framework has been enriched to allow scenarios as input, besides or instead of goals, and such a change gave rise to the problem of inferring goals from scenarios. Scenarios are given in the form of event trace diagrams, and can be positive or negative. Initially, scenarios are pre-processed in order to aggregate events, eliminate irrelevant events to the software agents, or to some scope of interest. The goals inferred may be of two types:

[7] Reproduced from section 2.1.2 of [van Lamsweerde and Willemet, 1998].

- Achieve/Cease - formally $P \Rightarrow \Diamond Q$ (where $\Diamond$ means at some point in the future, and $\Rightarrow$ expresses a usual implication that holds irrespective of the time, formally $P \Rightarrow Q$ iff $\Box(P \rightarrow Q)$, and $\Box$ means always in the future).
- Maintain/Avoid - formally $P \Rightarrow Q \mathsf{W} R$, where W means always in the future unless

The goals inferred should cover all positive scenarios and exclude all the negative ones, similarly to the ILP learning setting. The most important steps of the inference method (presented in section 4.1 and detailed in sections 4.2 - 4.7 of [van Lamsweerde and Willemet, 1998]) are the collection of progress facts (formally, $Pre-State \rightarrow \circ Post-state$, where $Pre-State$ and $Post-state$ are the states before and after the event), and invariance facts (formally, $R \wedge ST \rightarrow (R \mathsf{W} N)$, where R is a condition that remains true from state transition ST until another condition N becomes true). Subsequently, progress facts and invariance facts are generalized. We will focus on generalization over time and over instances. In generalization over time, implications ($\rightarrow$) are generalized over all states into $\Rightarrow$, and for progress facts, the next state operator ($\circ$) is generalized into some time in the future ($\Diamond$). In generalization over instances, the ground assertions are generalized in a *weak* manner by introducing existentially quantified variables or in a *strong* manner by introducing universally quantified variables.

To conclude, although there are more steps in the scenario elaboration process, we believe that the inference process is a crucial one. The final output of the system depends entirely on the goals inferred, and consists of specifications formally represented by temporal logic formulas involving agents, objects and operations.

6.4.2 Conflicts and Obstacles in Goal-Driven Requirements Engineering

This subsection briefly highlights the key ideas regarding how the KAOS system handles goal conflicts and obstacles, based on the work published in [van Lamsweerde et al., 1998] and [van Lamsweerde and Letier, 2000]. The detection and resolution of conflicts is integrated in the scenario elaboration framework in relation with goal elaboration. After conflicts are detected based on current goals, several solutions generate, modify or delete goals. There are also solutions in which conflicts are resolved by assigning the responsibilities to different agents, so there is also a connection between responsibility assignment and conflict resolution.

Inconsistencies can be of different types and relate to different aspects of requirements specifications. Inspired by [van Lamsweerde et al., 1998], we will mention here only the notions of conflict, divergence and obstruction:

- *Conflict* - occurs between assertions $A_1, ..., A_n$ in a domain Dom if:

  1) $(Dom, \wedge_{1 \leq i \leq n} A_i) \vdash false$ and

  2) $\forall i$, $1 \leq i \leq n$, $(Dom, \wedge_{j \neq i} A_j) \nvdash false.$

  The first condition states that there must be an inconsistency between the assertions in the domain, and the second condition enforces minimality, that is if we remove any of the assertions, there remaining assertions in the domain do not cause a conflict.

- *Divergence* - occurs between assertions $A_1, ..., A_n$ in a domain Dom if there exists a *boundary condition* B, such that:

  1) $(Dom, B, \wedge_{1 \leq i \leq n} A_i) \vdash false$,

  2) $\forall i$, $1 \leq i \leq n$, $(Dom, B, \wedge_{j \neq i} A_j) \nvdash false$ and

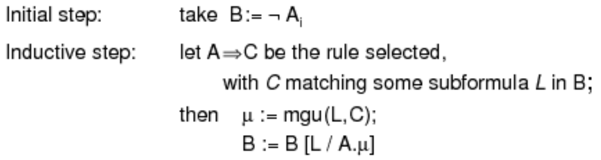
  3) $\exists$ scenario S and a time position i such that $(S, i) \vDash B$.

  The boundary condition B is a particular circumstance generated by scenario which gives rise to a conflict when taken together with assertions $A_1, ..., A_n$.

- *Obstruction* - represents the limiting case of a divergence when $n = 1$, i.e. we have only one goal assertion. Then, the boundary condition B is called an *obstacle* to the goal assertion A.

In [van Lamsweerde et al., 1998], the authors focus on detecting and resolving divergences, while in [van Lamsweerde and Letier, 2000] the main point of interests represents the analysis of obstacle identification and resolution. Although there are significant differences between divergence and obstacle detection, the general methods are the same in both situations: regressing negated assertions, using patterns and identification heuristics.

Informally, regressing negated assertions means negating an assertion, and then unifying it with another rule. The resolvent is the boundary condition. The formal method is given in section 4.1 of [van Lamsweerde et al., 1998], and reproduced in Figure 6.11.



```
Initial step:      take  $B := \neg A_i$
Inductive step:    let $A \Rightarrow C$ be the rule selected,
                       with $C$ matching some subformula $L$ in B;
                   then   $\mu := mgu(L,C)$;
                          $B := B [L / A.\mu]$
```

Figure 6.11: Method for regressing negated assertions.

Patterns provide an effective shortcut to the regression method, by using known and commonly encountered derivations of boundary conditions. An important aspect when dealing with obstacles is the fact that, unlike divergences, obstacles can be AND/OR-refined, such that obstacle refinement identifies both obstacles and domain properties. Finally, heuristics avoid any formal approach, and represent general rules of thumb helping to identify divergences or obstacles.

The problem of solving divergences and obstacles, although handled in a different way for divergences and obstacles, has common solving ideas. These relate to creating, deleting or modifying goals, or objects. We mention strategies such as:

- avoid boundary condition - can be realized by introducing a new goal of the form $P \Rightarrow \bigcirc\neg B$, where B is the boundary condition. The equivalent idea for obstacles is obstacle prevention.
- anticipate conflicts / obstacle anticipation - detect a circumstance which will lead at some point into the future to a conflict.
- goal weakening / goal deidealization - refine goal in order to cover the boundary condition
- alternative refinements to goals - which might avoid the conflicts/obstacles
- object/agent refinement - specializes the object/agent type, or reassign different responsibilities to agents in order to avoid conflicts

The ideas presented are only a subset of the broad analysis covered by the cited work, and are meant to provide a short description of the problem of managing conflicts in the context of requirements specifications elaboration process in the KAOS framework.

6.4.3 LTS synthesis based on End-User Scenarios

In [Damas et al., 2005], the authors present a system that accepts end-user scenarios as input, and provides a *labeled transitions system* (LTS) as output. There some restrictions for the input, the most significant being that end-user scenarios are communicated in the form of *message sequence charts* (MSC), no other information can be submitted to the system, and the scenarios supplied must contain at least one positive and one negative scenario. Positive scenarios describe desired behaviours of the system, while negative scenarios are expressed by a pair $\langle p, e \rangle$, where p is a positive MSC, called precondition, and e is a prohibited subsequent event. How can we relate the MSC input with an LTS representation? The answer is not a complex one: "An MSC timeline defines a total order on its input and output events. Therefore, it defines a unique finite LTS execution that captures a corresponding agent behavior." (quote from section 2.2, [Damas et al., 2005]).

The general outline of the method can be defined by three crucial steps:

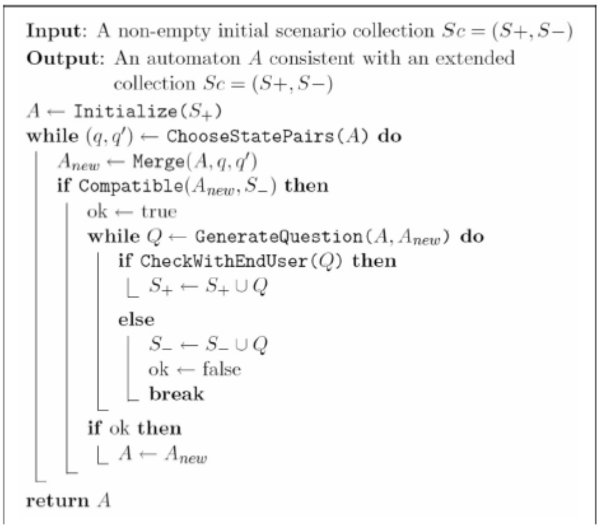
1. Submitting an initial set of positive and negative scenarios.

2. Generating scenario questions and synthesizing agent LTS.

3. Generating state invariants to document the generated state machines.

The first two steps are detailed in the algorithm in Figure 6.12 (reproduced from figure 11 from [Damas et al., 2005]). We will refer to the automaton A in the algorithm as an LTS, keeping in mind the difference that in an LTS all states are accepting states, while in an automaton only a subset of the possible states contains accepting states. The LTS is initialized such that it covers all positive scenarios, by constructing a *prefix tree acceptor* $PTA(S_+)$, that is the largest DFA (deterministic finite-state acceptor) that accepts the strings from S_+. This DFA will be generalized by merging a pair of states chosen according to a lexicographical order, and a potential update (A_{new}) of the LTS is created by merging [8] the two states. First of all, the potential update LTS has to be compatible with the negative scenarios (S_-), i.e. every string in S_- has to be rejected by A_{new}.

If A_{new} is compatible with S_-, the system will follow step 2, i.e. it will interact with the user through a series of questions, which essentially try to determine whether extensions of sequences accepted by A which are accepted by A_{new} are correct positive scenarios. As we can see in the figure below, in the *if CheckWithEndUser* branches, if the answer is affirmative, then the extended sequence is added as a new positive scenario, otherwise as a new negative scenario, and in this latter situation, the potential update DFA A_{new} is no longer an acceptable update.



```
Input: A non-empty initial scenario collection $Sc = (S+, S-)$
Output: An automaton $A$ consistent with an extended
        collection $Sc = (S+, S-)$
$A \leftarrow$ Initialize($S_+$)
while $(q, q') \leftarrow$ ChooseStatePairs($A$) do
    $A_{new} \leftarrow$ Merge($A, q, q'$)
    if Compatible($A_{new}, S_-$) then
        ok $\leftarrow$ true
        while $Q \leftarrow$ GenerateQuestion($A, A_{new}$) do
            if CheckWithEndUser($Q$) then
                $S_+ \leftarrow S_+ \cup Q$
            else
                $S_- \leftarrow S_- \cup Q$
                ok $\leftarrow$ false
                break
        if ok then
            $A \leftarrow A_{new}$
return $A$
```

Figure 6.12: Algorithm for LTS generation based on end-user scenarios.

Step three involves generating state invariants from fluent definitions. We will not describe the process here, but we mention it is an important step to help visualize the final LTS. To conclude, two potential directions of investigation shall be hinted. Is it possible to learn without initial negative scenarios from the end-user? Even in the current framework, negative scenarios can be generated by a negative answer to an end-user question, but perhaps a possibility would be to use the model-checking to generate negative scenarios. Another improvement might concern the choice of merging states, depending on the answer to the question: is there an ”intelligent” way to choose the states to be merged?

[8] Merging is not a trivial process: the resulting DFA, called a *quotient automaton*, might not be deterministic, and further merging of states would be necessary.

6.5 The I* Approach

The i* methodology proposes an alternative perspective on modelling requirements. Instead of focusing on scenarios or goals, the fundamental unit of its philosophy is the *intentional agent*, equipped with *individual* goals, beliefs, abilities and other properties, which are not always known or controllable. Agents depend on each other to reach their goals, share or trade resources and so on. Modelling a system as a complex interaction between agents is one of the reason i* is also referred to as *social modelling.*

Intentionality is either *explicit*, via *goals* which are decomposed into subgoals, or into means-ends alternatives, an idea which is similar to the AND/OR graph decomposition in KAOS. Intentionality expressed by *agents* is called *implicit*, because we might not have access to their internal goals, and is manifested thorough links between agents, which are called dependencies.

Dependencies are classified into:

- goal dependencies - are typically assertions, with no preference for the way of ensuring them;
- task dependencies - indicate precise activities which are expected from an agent
- resource dependencies - model the dynamics of entities which can be shared or transmitted between agents
- soft-goal dependencies - are similar to goal dependencies, with the mention that they claim a quality of the interaction, rather than a precise, binary assertion. This type of dependencies is used to model non-functional requirements.

Unfortunately, i* affords only a qualitative analysis of requirements, so the task of integrating quantitative measures such as probabilities remains a challenge which we hope to address in future work.

Chapter 7

Modelling and Verification. Probabilistic Model Checking.

In the previous part we have presented out contribution to PILP using SLPs, the ProbPoly framework. In this chapter we aim to show how we might adapt a methodology for learning requirements using model checking and ILP to the probabilistic context, using probabilistic model checking and PILP, and more specifically, ProbPoly. To our knowledge, this is an unexplored area of research, and the most relevant system which learns probabilistic assumptions is the one reviewed in Section 6.3, but it uses probabilistic model checking in conjunction with an modified version of L*. We demonstrate our novel approach of combining probabilistic model checking and logic learning as a teacher-learner method on a small example.

After the previous reviews of a range of relevant systems which focus on modelling and some also on learning requirements in Chapter 6, we introduce the main challenge of learning requirements, and begin to reason on how we may overcome it by using ProbPoly. Section 7.1 establishes the premises of our approach, so that we are able to offer a simple, yet representative example in Section 7.2. Finally, in Section 7.3 we discuss the main questions which motivate directions of future research.

7.1 Probabilistic Model Checking. PRISM.

Probabilistic model checking is a framework for the verification and analysis of probabilistic finite-state models, which are checked against probabilistic properties, e.g. formulas in probabilistic extensions of temporal logic. Unlike traditional model checking, the probabilistic extension can analyse properties of quantitative nature, e.g. "the system will reach a fail state in at most k time units with a probability less than 0.02". In this example, the probability can be computed based on the probabilities of the model, and the k time units is expressed in terms of costs (or equivalently rewards).

The result of checking a property is a truth value, expressing whether the property is satisfied or not. In the case that the property holds, additional output may consist of a lower and/or an upper bound on the probability of the respective property. In the case that the property isn't satisfied, a counterexample should be given, that is a set of traces through the model which violate the property. The definition and computation of counterexamples in probabilistic model checking is an active research topic among software engineers. We will give an intuitive definition of what we believe is the best counterexample in Section 7.2.

Probabilistic model checking must have efficient data structures (e.g. BDDs) and numerical algorithms for computation of the different quantities, in order to allow large-scale models. To gain a better understanding of probabilistic model checking, let us look at what type of models are commonly used in the field.

The fundamental problem of designing a system that learns requirements is the representation of the domain and of the software system. We believe that the LTS, as presented in Section 6.3, is a state-of-the-art representation for systems which don't incorporate probability, and in the probabilistic context perhaps the most popular models are:

- Discrete Time Markov Chains (DTMCs),
- Continuous Time Markov Chains (CTMCs), also known as Continuous Time Markov Processes,
- Markov Decision Processes (MDPs), which are almost equivalent to Probabilistic Automata (PA). We will use the latter in our terminology.

The simplest model is a DTMC:

Definition 7.1 ((Discrete Time) Markov Chain). *A Discrete Time Markov Chain, as defined in [Manning and Schütze, 1999], is a sequence of random variables $X = (X_1, \ldots, X_T)$ taking values in some finite set $S = \{s_1, \ldots, s_N\}$, the state space and satisfies the* Markov Properties*:*

- *Limited Horizon:*
  $P(X_{t+1} = s_k|X_1, \ldots, X_t) = P(X_{t+1} = s_k|X_t)$
- *Time invariant (stationary):*
  $P(X_{t+1} = s_k|X_t) = P(X_2 = s_k|X_1)$*, where* $\forall t, t+1 \in 1, \ldots, T$ *and* $\forall k \in 1, \ldots, N$

The time invariance condition is sometimes omitted, or mentioned as a property of a special case of Markov Chains (time-homogeneous or stationary), and is usually defined in a more intuitive way:

$P(X_{t+1} = s_k|X_t = s_l) = P(X_t = s_k|X_{t-1} = s_l)$

A DTMC is described by a stochastic transition matrix A with entries:

$a_{ij} = P(X_{t+1} = s_j|X_t = s_i)$, with

$a_{ij} \geq 0$, $\forall i, j \in 1, \ldots, N$ and $\displaystyle\sum_{j=1}^{N} a_{ij} = 1$, $\forall i \in 1, \ldots, N$.

Usually, Markov chains are also characterized by the probabilities on the initial states:

$\pi_i = P(X_1 = s_i)$, with $\displaystyle\sum_{i=1}^{N} \pi_i = 1$.

However, we can always add an extra "start" state, such that $P(X_1 = start) = 1$, and therefore the probabilities π_i are pushed into the transition matrix A. We will indeed choose to characterize a DTMC only by matrix A.

Let us exemplify a DTMC which illustrates a fair six-sided die. This example is considered as a case study in the probabilistic model checker PRISM (which we will discuss a little later) and was originally presented in [Knuth and Yao, 1976].

Example 7.1 (Fair Six-Sided Die DTMC). *The transition matrix for this DTMC is sparse, so we will represent it as such*[1]*:*

[1] A list of $\langle i, j, p\rangle$ tuples such that $i, j \in 1, \ldots, N$, $a_{ij} = p$, and for all omitted pairs k, l, with $k, l \in 1, \ldots, N$, $a_{kl} = 0$.

From	*To*	*Transition Probability*
0	*1*	*0.5*
0	*2*	*0.5*
1	*3*	*0.5*
1	*4*	*0.5*
2	*5*	*0.5*
2	*6*	*0.5*
3	*1*	*0.5*
3	*7*	*0.5*
4	*8*	*0.5*
4	*9*	*0.5*
5	*10*	*0.5*
5	*11*	*0.5*
6	*2*	*0.5*
6	*12*	*0.5*
7	*7*	*1*
8	*8*	*1*
9	*9*	*1*
10	*10*	*1*
11	*11*	*1*
12	*12*	*1*

Although this is an acceptable mathematical and computational representation, we understand DTMCs much easier if they are expressed graphically in Figure 7.1:

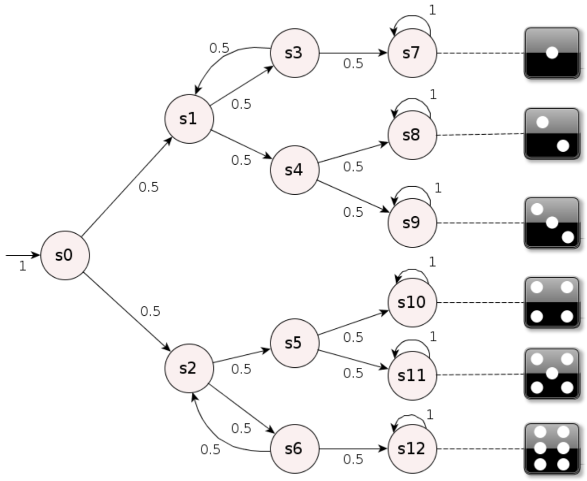


Figure 7.1: DTMC for a Fair Six-Sided Die. The dashed lines and die images are included to relate the model to its semantics; they are not part of the DTMC.

CTMCs are essentially DTMCs with continuous indexes instead of discrete ones. We don't treat them formally since we haven't done any experiments involving CTMCs yet.

Probabilistic Automata (PA) combine DTMCs and LTSs in an expressive way: from every state we must choose an action, in a non-deterministic fashion, similar to the LTS case, however, we aren't guaranteed to reach a single destination state. Instead, we have a function $trans : N \times M \times N \to [0,1]$, where N is the number of states, and M is the set of actions, such that:

$$\sum_{j=1}^{N} trans(i,a,j) = 1, \forall i \in 1,\dots,N, a \in M.$$

In Figure 7.2 we show two PA from [Kwiatkowska et al., 2010]. Similar to LTSs, a *parallel composition* operator can be defined on PA, but we leave out the formal details because we haven't tested our methods on compositions of PA.

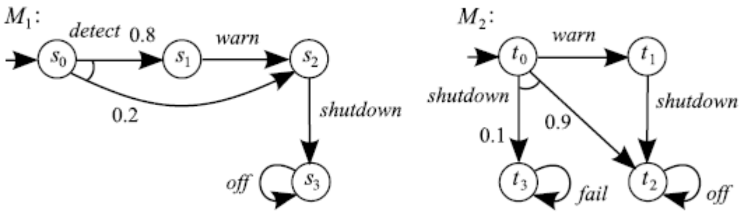
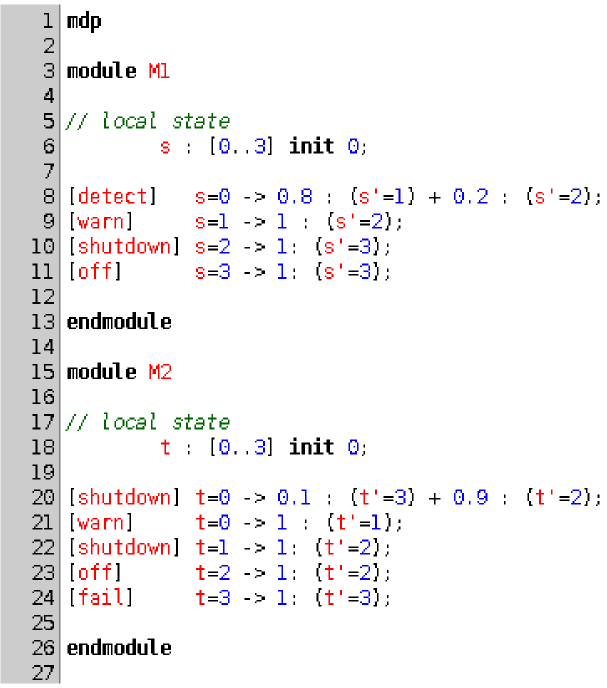


Figure 7.2: Two Probabilistic Automata from [Kwiatkowska et al., 2010].

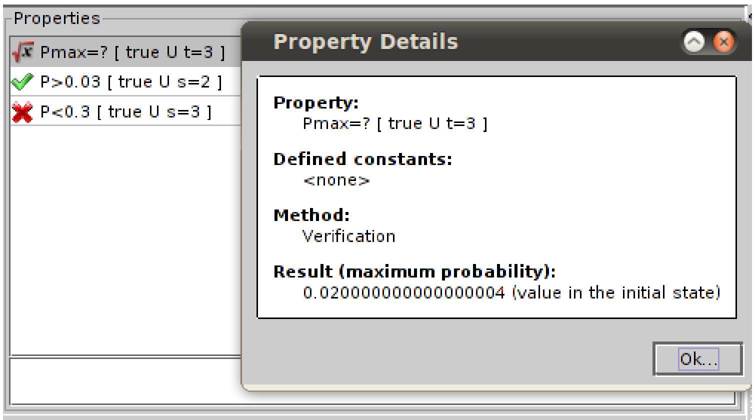
Finally, we would like to mention that we have chosen to perform our experiments in the PRISM probabilistic model checker[2] introduced in [Hinton et al., 2006]. PRISM has a very intuitive language for describing models, for example let us write the two models from Figure 7.2. [3] The result is shown in Figure 7.3.



```
mdp

module M1

// local state
        s : [0..3] init 0;

[detect]   s=0 -> 0.8 : (s'=1) + 0.2 : (s'=2);
[warn]     s=1 -> 1 : (s'=2);
[shutdown] s=2 -> 1: (s'=3);
[off]      s=3 -> 1: (s'=3);

endmodule

module M2

// local state
        t : [0..3] init 0;

[shutdown] t=0 -> 0.1 : (t'=3) + 0.9 : (t'=2);
[warn]     t=0 -> 1 : (t'=1);
[shutdown] t=1 -> 1: (t'=2);
[off]      t=2 -> 1: (t'=2);
[fail]     t=3 -> 1: (t'=3);

endmodule

```

Figure 7.3: Prism Model of the PA in Figure 7.2.

Properties are defined in a separate file. Figure 7.4 is a snapshot of a property file loaded into PRISM and checked against the model from Figure 7.3.



√x Pmax=? [true U t=3]
✓ P>0.03 [true U s=2]
✗ P<0.3 [true U s=3]

Property:
Pmax=? [true U t=3]

Defined constants:
<none>

Method:
Verification

Result (maximum probability):
0.020000000000000004 (value in the initial state)

Figure 7.4: Prism Properties checked against the model from Figure 7.3.

[2] http://www.prismmodelchecker.org (accessed 07.09.2011).

[3] When multiple models are defined in a single file, PRISM always considers their composition.

The first property checks the maximum probability that can be reached in state t_3. The formula *true* U $t = 3$ means that the probability must be checked at each time step *until* (U) we are in state t_3. The second and third properties are similar to the first one: they simply check a given probability as a lower/upper bound for a certain state.

7.2 Learning A Simple Discrete Time Markov Chain

In this section we present the learning process of the probabilities for a six-sided die, so that two basic properties hold in the new model. Let us formalize our task:

Given: an initial model as a DTMC, and a list of properties which need to be satisfied,

Learn: new parameters for the DTMC such that all the properties are satisfied.

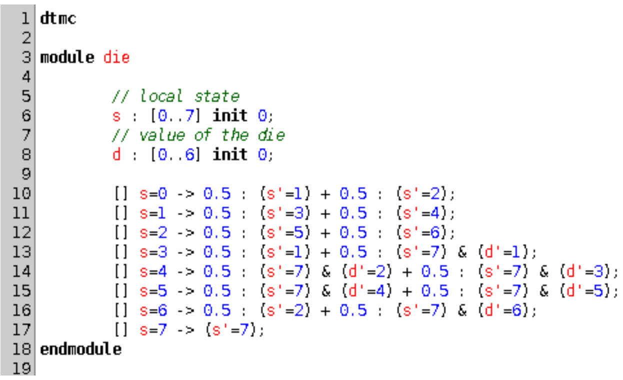
First of all notice that we only learn the parameters, and not the structure of the DTMC. The method of learning the parameters is ProbPoly, and as detailed in the first part of the thesis, ProbPoly learns a new predicate based on counterexamples, which should be generated by the model checker[4]. We mentioned at the beginning of the chapter the issue of counterexamples in probabilistic model checking. What should be a counterexample in the case of a DTMC?

Fortunately this problem has been addressed, so we adopt the definition of a counterexample from [Han and Katoen, 2007]. The cited publication defines the *strongest evidence* as a path in the model which has a maximum probability. The evidence may or may not be a counterexample based on the value of the evidence and the upper bound of a probabilistic property (of the form $P_{<p}(\dots)$) that the model is checked against.

We will use the term counterexample when referring to the strongest evidence in the case that is a counterexample, unless there is ambiguity. The key idea of computing the strongest evidence on a DTMC is by transforming it into a weighted digraph and solving the problem of k^{th}*-shortest path*. The authors also define the *smallest counterexample* as the counterexample with the shortest path, and maximum probability.

Consider the following setting:

Example 7.2 (Six-sided die)**.** *The* initial model *is the fair six-sided die from Example 7.1. Its PRISM model is:*



```
dtmc

module die

        // local state
        s : [0..7] init 0;
        // value of the die
        d : [0..6] init 0;

        [] s=0 -> 0.5 : (s'=1) + 0.5 : (s'=2);
        [] s=1 -> 0.5 : (s'=3) + 0.5 : (s'=4);
        [] s=2 -> 0.5 : (s'=5) + 0.5 : (s'=6);
        [] s=3 -> 0.5 : (s'=1) + 0.5 : (s'=7) & (d'=1);
        [] s=4 -> 0.5 : (s'=7) & (d'=2) + 0.5 : (s'=7) & (d'=3);
        [] s=5 -> 0.5 : (s'=7) & (d'=4) + 0.5 : (s'=7) & (d'=5);
        [] s=6 -> 0.5 : (s'=2) + 0.5 : (s'=7) & (d'=6);
        [] s=7 -> (s'=7);
endmodule

```

The desired properties *are:*

1. $P < 0.15$ [*true* U $s = 7$ & $d = 1$], *i.e. the probability of obtaining a die value of 1 is less than 0.15,*

2. $P < 0.02$ [*true* U $s = 7$ & $d = 5$], *i.e. the probability of obtaining a die value of 5 is less than 0.02.*

The model is not a straightforward translation of the DTMC from Example 7.1. States 7-12 of the DTMC from the previous section are absorbed into state 7, and we use a separate variable d

[4] The current version of PRISM (4.0.2) has no counterexample generation options, and it seems incompatible with the DiPro tool http://www.inf.uni-konstanz.de/soft/dipro/ (accessed 08.09.2011)

to encode the value of the die in s_7. This is possible because states 7-12 are all terminating states, i.e. extending a path from them is an loop of the same state with probability 1.

We may think of the die as a very simple system, and that obtaining die value 1 is an error situation, and obtaining a die value of 5 is a critical error state of the system. They are checked against the initial model, and both properties aren't satisfied, so we consider generating counterexamples for each one.

It is obvious that the counterexamples with the highest probability are:

- for the first property: $c_1 = [s_0,\ s_1,\ s_3,\ s_7,\ d_1]$. This path has a probability of $0.5 \cdot 0.5 \cdot 0.5 = 0.125$. This probability is smaller than 0.15, but take into account that the total probability of reaching d_1 is c_1 and the infinite number of paths obtained by looping n ($n \to \infty$) times through $[s_3, s_1, s_3]$. Its value for the initial model is, as expected, $1/6 = 0.1(6)$.

- for the second property: $c_2 = [s_0,\ s_2,\ s_5,\ s_7,\ d_5]$, with the same probability as c_1.

The next step in our approach is to remodel the DTMC in Prolog, because at the moment we are unfamiliar with the implementation details of PRISM. The initial model is represented as a background knowledge file in Listing A.6. Notice that each predicate has corresponds to a state, and we omit s_7 and simply specify d_i. The counterexamples for the properties are encoded as probabilistic examples for ProbPoly. We can't ensure strict inequality with the current implementation of ProbPoly, due to the fact that if we get a very small global minimum for the error function, the probabilities tend to be equal. An important observation is also that unlike the probabilistic model checker, we can't handle infinite recursion[5], so we compute an incomplete probability.

The current version of ProbPoly can't learn the probabilities of multiple predicates at the same time, since we enforce the constraint that the probabilities of the new clauses must sum up to one when there are no previously learned clauses, so our approach is the following:

Algorithm 1 Learn new DTMC parameters using ProbPoly

```
IN: initial model file $BK_0$, $N$ number of states
OUT: final model $BK_N$
for $i = 1 \to N$ do
    $BK_i \leftarrow BK_{i-1} \setminus \{P_{i-1} : C_{i-1} \mid C^{+}_{i-1} = s_{i-1}\}$
    $H \leftarrow C_{i-1}$
    $P'_{i-1} : C'_{i-1} \leftarrow ProbPoly(BK_i, H)$
    $BK_i \leftarrow BK_{i-1} \cup \{P'_{i-1} : C'_{i-1}\}$
end for
```

In Algorithm 1, we are given the initial model file, and we process it according to a certain order of the states: the probabilistic clauses for each state are removed from the background file and added without probabilities to the theory we want to learn. Then, the new probabilistic clauses are re-added to the background file and we continue with the next state.

The output of the learning process for Example 7.2 is illustrated in Figure 7.5.

Finally, we translate the new model into PRISM and the model checker successfully verifies the initial properties.

[5] The maximum recursion depth is controlled by the *max_proof_length*/1 predicate.

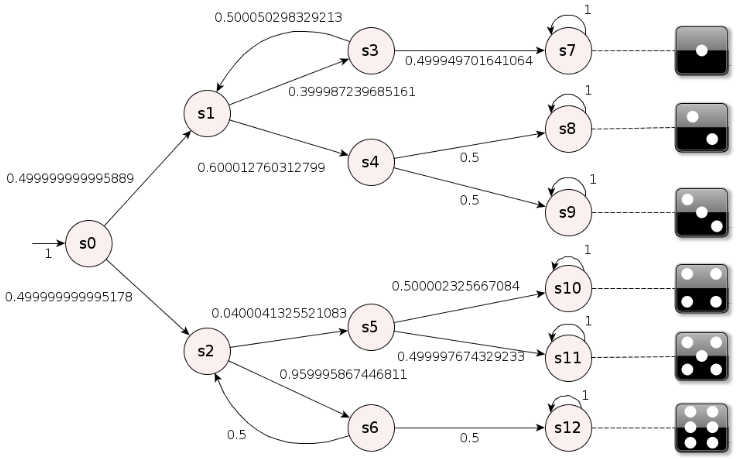


Figure 7.5: Graphical illustration of the learned model.

7.3 Conclusions

To conclude this chapter, we present some observations about our learning procedure. First of all, due to the fact that, at the moment, we learn the probabilities of each state, this forces us to use an initial model as input, and to perform learning incrementally. This is very similar to the Expectation Maximization (EM) paradigm. However, Expectation Maximization can only guarantee to reach a local maximum, determined by the initial parameters.

By minimizing the mean squared error function for the probabilities of each state, we get a greedy algorithm, e.g. consider the probabilities of transition from s_2 in the learned model. This strategy might output probabilities that simply can't be enforced in the system. Thus, one of our goals is to learn a model which is as close as possible to the initial one. This is somewhat similar to the regularization idea of parametric models, e.g. weight decay in artificial neural networks. A possible way to implement it is by modifying the error function such that each example is penalized by *adding* (since our task is minimization) a value proportional to the difference between the initial value of the probability and the learned value.

We believe that our current strategy could be redefined in a formal EM paradigm; more importantly, we are confident on the fact that minor implementation modifications will allow ProbPoly to learn multiple predicates, since it is only a matter of redefining constraints for sets of variables corresponding to the same clause head. In this context, we could learn the parameters based on a purely structural initial model. We also speculate that using ILP in conjunction with ProbPoly might allows to combine structure learning and parameter learning for probabilistic models.

Finally, an important theoretical and practical challenge is to adapt ProbPoly to infinite paths in the model. This is equivalent to the multivariate constrained minimization of the sum of a series of polynomials, an area which we haven't researched yet.

Chapter 8

Future Work

Using the conclusions we have drawn from the two parts of the thesis, we can formulate a list of interesting directions to pursue in future work.

Learn probabilities for refined clauses

It is possible to perform learning in the case that clause c exists in H – let us refer to it as $y_j : h_j$ – and it is refined into clause c'. This is the case in the cover loop algorithm (e.g. in FOIL [Quinlan and Cameron-Jones, 1993]) or in the ILP system HYPER, presented in [Bratko, 2000], where we consider refinements over a forest, each tree corresponding to the refinement of a clause. In this case, we could delete the clause to be refined $y_j : h_j$ from the SLP and normalize the other clauses by

$$y'_i = \frac{y_j}{\displaystyle\sum_{i \in \overline{1,n} \setminus \{j\}} y_i}$$

for $h_i, \forall i \in \overline{1,n} \setminus \{j\}$. Then, the problem is the same as adding clause $x' : c'$, i.e. the refined clause, to the normalized clauses $y'_i : h_i$.[1]

Use probabilistic quantities as scores that guide the ILP search

The fusion or at least interaction of structure learning and parameter learning is a promising idea which might allow to improve the quality of the learning process. We intend to use the value of the computed error function as a selection criterion for choosing a particular refinement, perhaps combined with traditional ILP scores.

We also plan to define measures such as precision, recall etc. for SLPs based on the quantities we introduced in the last subsection of Section 4.3. These measures could also be used to guide the search, as proven in [Raedt and Thon, 2010].

Extend ProbPoly to learning multiple predicates simultaneously

Learning multiple predicates simultaneously is the next generalization step in the development of ProbPoly. We already have a precise idea on what should be implemented.

Recall that the input of ProbPoly is the set of probabilistic clauses $OldH$ and the set of non-probabilistic clauses $NewC$. First we will add variables to $NewC$. Then we will partition the set $OldH \cup NewC$ into l partitions based on the head predicate of the clause, assuming we have l different head predicates in $OldH \cup NewC$. For each partition we will generate the appropriate constraints (the sum of all probabilities in that partition must be 1). After this step, the method will be the same as in the multi-clause case: we will build the error polynomial, generate the additional constraints for valid probabilities and negative examples, and run the optimization function.

[1] Note that after learning x' we need to normalize again clauses y_i by $(1 - x')$.

By making this modification to the implementation, we will be able to learn all the probabilities of a DTMC in one run of ProbPoly, and without relying on an initial model. However, in the context of larger models, this might be infeasible for GloptiPoly and we don't know how efficient Particle Swarm Optimization will be. This is because the number of states in the model is the number of predicates in the Prolog representation, and the number of transitions from one state is the number of clauses for the corresponding predicate. We might still want to rely on incremental learning.

Explore the advantages of learning using negative examples

Although we have formally defined negative examples as simply goals which have 0 probability, we haven't yet tested cases in which we learn from them. In a PCFG context, a negative example represents a string that can't be parsed by the grammar, so it is possible that the learning algorithm will essentially delete (from the point of view of the probabilities, and not the clause structure) transitions by assigning them a very low probability.

Develop techniques for learning from infinite proofs

Even in simple DTMC models infinite loops may occur. Our current framework can consider only a recursion depth of k on a certain state, where k is a parameter. However, this cannot be used for highly recursion models, since some states may accumulate significant probability for paths with more than k occurrences of unknown transitions.

We believe the methods used in probabilistic model checking might be valuable in allowing learning from infinite proofs. A better mathematical understanding on how to tackle the problem infinite polynomials is also required.

Validate ProbPoly by integrating it into ILP systems

We have made minor efforts so far in integrating ProbPoly in Hyper and TAL systems. We need to run additional experiments and allow ProbPoly to learn predicates such as equality ($X = Y$) or append ($append(L_1, L_2, L)$), which should always have a probability of 1, since they are purely logical.

Make the implementation scalable

Although we haven't yet implemented a method of ensuring that the products of probabilities doesn't produce underflows, we propose two ways of achieving this:

- transform the probabilities by using $log(p)$ or $log(p^{-1})$ instead of p.
- design a method based on *scaling factors*, inspired by the example of hidden Markov models (HMMs).

Address the problem of multiple solutions of the optimization

As shown in Example 4.5, it is sometimes the case that we might have more than one set of parameters which ensure the global minimum of the error function. In the current implementation, we assign the clauses the first learned solution. However, this is a completely random choice, without any formal reason to motivate it. In the case of incremental learning, discussed in Secion 7.2, it might prove useful to establish a backtracking structure based on the multiple possibilities we can assign probabilities to newly learned clauses, and explore it in order to find the most suitable solution, i.e. the choice of parameters for which the error function over the completely learned model is minimum.

Consider larger discrete time Markov chains (DTMCs)

Experiments on larger models, and on models other than DTMCs are essential if we wish to develop a robust learning framework using ProbPoly and PRISM. We also need to design improved learning mechanisms, based on updates of PorbPoly and the types of probabilistic models and properties we use.

Chapter 9

Conlcusions

In the first part of our thesis, we have formulated the learning task of computing probabilities for newly learned clauses in SLPs based on probabilistic examples. We managed to overcome the limitations of the initial learning paradigm, and we believe that through our examples we have illustrated the correctness of our approach. Although in the multi-clause case we cannot always guarantee an analytical solution to the optimization problem, we use particle swarm optimization to still be able to learn the desired probabilities, allowing non-optimality.

The design of a robust framework for learning requirements using probabilistic model checking and PILP is one of the major tasks that need to be achieved. However, we have taken some preliminary steps in this direction by designing a simple method for learning the parameters of a model in order to meet a list of properties, which proved successful on a simple example.

Overall, we are pleased with the results obtained, especially due to the fact that in almost all of the experiments the correct output was anticipated based on theoretical considerations. One of the surprising results was in the experiment for Example 4.5, in which learning based on only one example string (”aa”) proved to be optimal for two solutions. As we mention in our list of directions for future work, the problem of choosing between the two optimal solutions arises in the case of incremental learning, proposed in Secion 7.2, in which we learn based on a partially learned model, with the non-learned probabilities assigned as in the initial model.

To conclude, we believe that PILP and probabilistic model checking are relatively new areas of research in computer science, so we have to permanently keep up with the dynamics of both subjects, and at the same time to continue to extend and improve the ProbPoly framework, in a sustained effort to achieve results on real-world examples comparable to the state-of-the-art .

Appendix A

Code Listings

Listing A.1: ProbPoly main predicate

```prolog
% ==== probpoly_mc/4
% ==== probpoly_mc(+OldH, +NewClause, -NewH, -Err) -
%  OldH is a list of probabilistic clauses,
% NewClauses is a list of non-probabilistic clauses,
% NewH is the theory obtained by learning a probabilities for NewClauses
% and updating the others
% Err is the value of the error function for NewH on the examples
probpoly_mc(OldH, NewClauses, NewH, Err) :-
    retractall(mpoly_nvar(_)),
    length(NewClauses, LenNC),
    assertz(mpoly_nvar(LenNC)),
    add_probs(NewClauses, EvalNewClauses, 1),
    append(OldH, EvalNewClauses, EvalH),
    (OldH=[] ->
     Equal = true
    ;
     Equal = false
    ),
    rec_sol(EvalH, Equal, X, Err),
    (X = [] ->
    Val is 1/LenNC,
    list_gen(LenNC, Val, L),
    add_vals(NewClauses, UpNewClauses, L),
    sum_list(L, SumL),
    updateH(OldH, UpOldH, SumL),
    append(UpOldH, UpNewClauses, NewH)
    ;
    sum_list(X, SumX),
    updateH(OldH, UpOldH, SumX),
    add_vals(NewClauses, UpNewClauses, X),
    append(UpOldH, UpNewClauses, NewH)
    ),
    retractall(mpoly_nvar(_)).
```

Listing A.2: A Probabilistic Hypothesis Interpreter for Learning a Single Clause

```prolog
% === prob_prove/3
% === prob_prove(+Goal, +Hypo, -Poly) - for a given Goal and
% a probabilistic hypothesis Hypo, Poly is the polynomial
% corresponding to a proof of Goal using Hypo (and background knowledge)
prob_prove( Goal, Hypo, Poly) :-
  max_proof_length( D),
  prob_prove( Goal, Hypo, D, _RestD, Poly, [(1,0)]).

prob_prove( _Goal, _Hypo, []).        % Otherwise Goal definitely cannot be proved

prob_prove( _G, _H, D, D, [], _PolyAcc) :-
  D < 0, !.                % Proof length overstepped

prob_prove( [], _Hyp, D, D, Poly, Poly) :- !.

prob_prove( [G1 | Gs], Hypo, D0, D, Poly, PolyAcc) :- !,
  prob_prove( G1, Hypo, D0, D1, P1, PolyAcc),
  prob_prove( Gs, Hypo, D1, D, Poly, P1).

prob_prove( G, _Hyp, D, D, Poly, PolyAcc) :-
  prolog_predicate( G),              % Background predicate in Prolog?
  pc(Prob, G),
  mul_by_const(PolyAcc, Prob, Poly),
  call(pc(_Prob, G)).                          % Call of background predicate

prob_prove( G, Hyp, D0, D, Poly, PolyAcc) :-
  prolog_predicate( G),              % Background predicate in Prolog?
  pc(Clause/_Vars),
  copy_term( Clause, [Prob, Head | Body] ), % Rename variables in clause
  G = Head,
  mul_by_const(PolyAcc, Prob, PolyAcc1),
  prob_prove( Body, Hyp, D0, D, Poly, PolyAcc1).          % Call of background predicate

prob_prove( not(G), Hyp, D, D, Poly, PolyAcc) :-
   total_prob(G, Hyp, NPoly),
   inverse_poly(NPoly, NPoly1),
   add_poly(NPoly1, [(1, 0)], NPoly2),
   mul_by_poly(PolyAcc, NPoly2, Poly).

prob_prove( G, Hyp, D0, D, Poly, PolyAcc) :-
  D0 =< 0, !, D is D0-1,              % Proof too long
  Poly=[]
  ;
  D1 is D0-1,                 % Remaining proof length
  member( Clause/_Vars, Hyp),          % A clause in Hyp
  copy_term( Clause, [Prob, Head | Body] ), % Rename variables in clause
  G = Head,                          % Match clause's head with goal
  (ground(Prob) ->
    mul_by_const(PolyAcc, Prob, PolyAcc1), % mul by Prob
    mul_by_poly(PolyAcc1, [(-1,1), (1,0)], PolyAcc2) % mul by (1-X)
    ;
    mul_by_poly(PolyAcc, [(1, 1)], PolyAcc2) % mul by X
  ),
  prob_prove( Body, Hyp, D1, D, Poly, PolyAcc2).          % Prove G using Clause
```

Listing A.3: Predicate for MatLab Initialization

```
% === ml_init/0
% === ml_init - initialize matlab environment
ml_init :-
    matlab_on, !.
ml_init :-
    getcwd(D),
    cd('../MatLab'),
    start_matlab('matlab -nojvm -nosplash -nodisplay'),
    matlab_eval_string('path(path, genpath(pwd))'),
    cd(D).
```

Listing A.4: Predicate for Computation of Binomial Coefficients

```
% === binom_coeff/3
% === binom_coeff(+N, +K, -Val) computes the value Val
% of the combinations of N elements in groups of K
%      ( N )        n!
%   = (    ) =  --------
%      ( K )     k!(n-k!)

binom_coeff( N, K, Val) :-
    K>=0,
    N>=K,
    NK is N-K,
    (K>NK ->
        binom_coeff( N, K, Val, 1, 1);
        binom_coeff( N, NK, Val, 1, 1)
    ).

binom_coeff( _N, K, Val, Val, K1) :- K1 is K+1, !.

binom_coeff( N, K, Val, ValAcc, Counter) :-
    ValAcc1 is ValAcc * (N-Counter+1)/(Counter),
    Counter1 is Counter + 1,
    binom_coeff( N, K, Val, ValAcc1, Counter1).
```

Listing A.5: Predicate for Computation of Binomial Coefficients using MatLab Package Interface

```
% === ml_binom_coeff/3
% === ml_binom_coeff(+N, +K, -Val) computes the value Val
% of the combinations of N elements in groups of K
%      ( N )        n!
%   = (    ) =  --------
%      ( K )     k!(n-k!)
ml_binom_coeff(N, K, Val) :-
    ml_init,
    matlab_vector(1, [N], n),
    matlab_vector(1, [K], k),
    val <-- nchoosek(n,k),
    matlab_get_variable(val, [Val]).
```

Listing A.6: Prolog File Representing the Six-Sided Fair Die DTMC from Example 7.1

```

% PREDICATES TO BE EVALUATED BY PROLOG
prolog_predicate(s0(_)).
prolog_predicate(s1(_)).
prolog_predicate(s2(_)).
prolog_predicate(s3(_)).
prolog_predicate(s4(_)).
prolog_predicate(s5(_)).
prolog_predicate(s6(_)).

% PROBABILISTIC CLAUSES
pc([0.5,s0([s(0)|T]), s1(T)]/[]).
pc([0.5,s0([s(0)|T]), s2(T)]/[]).
pc([0.5,s1([s(1)|T]), s3(T)]/[]).
pc([0.5,s1([s(1)|T]), s4(T)]/[]).
pc([0.5,s2([s(2)|T]), s5(T)]/[]).
pc([0.5,s2([s(2)|T]), s6(T)]/[]).
pc([0.5,s3([s(3)|T]), s1(T)]/[]).
pc([0.5,s3([s(3), d(1)])]/[]).
pc([0.5,s4([s(4), d(2)])]/[]).
pc([0.5,s4([s(4), d(3)])]/[]).
pc([0.5,s5([s(5), d(4)])]/[]).
pc([0.5,s5([s(5), d(5)])]/[]).
pc([0.5,s6([s(6)|T]), s2(T)]/[]).
pc([0.5,s6([s(6), d(6)])]/[]).

%EXAMPLES
pex(0.1, s0([s(0), s(1), s(3), d(1) ]), 1).
pex(0.01, s0([s(0), s(2), s(5), d(5)]), 1).
```

Listing A.7: MatLab Script for GloptiPoly generated when learning the probabilities in Example 4.5 (with all three examples)

```
 clear
 mpol x 2;
 g0=( + 0.500000*x(1)^2*x(2)^8 ...
+ 0.500000*x(1)^2*x(2)^4 ...
+ -0.005700*x(1)^1*x(2)^4 ...
+ -0.063000*x(1)^1*x(2)^2 ...
+ 0.500000*x(1)^2 ...
+ -0.700000*x(1)^1 ...
+ 0.247001 ...
);
K = [ 0 <= x(1), x(1) <= 1, ...
0 <= x(2), x(2) <= 1, ...
x(1)+x(2) == 1];
```

Listing A.8: MatLab Objective Function for PSOpt generated when learning the probabilities in Example 4.5 (with all three examples)

```
function f = polyfcn(x)
if strcmp(x,'init')
     f.PopInitRange = [0; 1] ;
 else
 f = + 0.500000*x(1)^2*x(2)^8 ...
+ 0.500000*x(1)^2*x(2)^4 ...
+ -0.005700*x(1)^1*x(2)^4 ...
+ -0.063000*x(1)^1*x(2)^2 ...
+ 0.500000*x(1)^2 ...
+ -0.700000*x(1)^1 ...
+ 0.247001 ...
;
 end
```

Listing A.9: MatLab Constraint Function for PSOpt generated when learning the probabilities in Example 4.5 (with all three examples)

```
function [c, ceq] = confun(x)
 c = []; ceq = [x(1)+x(2)-1];
 end
```

Listing A.10: MatLab Function to Optimize a Polynomial using GloptiPoly or PSOpt

```
function [xmin, polymin, allsol] = mpoly_gloptipoly3(x, g0, K)
P = msdp(min(g0), K);
[status,obj] = msol(P);
if(status == 1)
    polymin = obj;
    allsol = double(x);
    xmin = allsol(:,:,1);
else
    nvar = length(x);
    [xmin, polymin] = pso(@polyfcn, nvar, [], [], [], [], ...
                          zeros(1,nvar), ones(1,nvar), @confun);
    allsol=xmin;
end
end
```

Bibliography

[Alrajeh et al., 2009a] Alrajeh, D., Kramer, J., Russo, A., and Uchitel, S. (2009a). Learning operational requirements from goal models. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 265–275, Washington, DC, USA. IEEE Computer Society.

[Alrajeh et al., 2009b] Alrajeh, D., Ray, O., Russo, A., and Uchitel, S. (2009b). Using abduction and induction for operational requirements elaboration. *Journal of Applied Logic*, 7(3):275 – 288. Special Issue: Abduction and Induction in Artificial Intelligence.

[Alrajeh et al., 2006] Alrajeh, D., Ray, O., Russo, R., and Uchitel, S. (2006). Extracting requirements from scenarios with ilp. In *Proceedings of the 16th International Conference on Inductive Logic Programming*, pages 63–77.

[Angluin, 1987] Angluin, D. (1987). Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75:87–106.

[Bishop, 2007] Bishop, C. M. (2007). *Pattern Recognition and Machine Learning (Information Science and Statistics).* Springer, 1st ed. 2006. corr. 2nd printing edition.

[Borges et al., 2010a] Borges, R. V., d'Avila Garcez, A., and Lamb, L. C. (2010a). Integrating model verification and self-adaptation. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pages 317–320, New York, NY, USA. ACM.

[Borges et al., 2010b] Borges, R. V., Garcez, A. D., and Lamb, L. C. (2010b). Representing, learning and extracting temporal knowledge from neural networks: a case study. In *Proceedings of the 20th international conference on Artificial neural networks: Part II*, ICANN'10, pages 104–113, Berlin, Heidelberg. Springer-Verlag.

[Bratko, 2000] Bratko, I. (2000). *Prolog Programming for Artificial Intelligence.* Addison Wesley, 3rd edition.

[Chen et al., 2008] Chen, J., Muggleton, S., and Santos, J. C. A. (2008). Learning probabilistic logic models from probabilistic examples. *Machine Learning*, 73(1):55–85.

[Corapi et al., 2010] Corapi, D., Russo, A., and Lupu, E. (2010). Inductive logic programming as abductive search. In *ICLP (Technical Communications)*, pages 54–63.

[Costa et al., 2011] Costa, V. S., Damas, L., and Rocha, R. (2011). The yap prolog system. *CoRR*, abs/1102.3896.

[Damas et al., 2005] Damas, C., Lambeau, B., Dupont, P., and van Lamsweerde, A. (2005). Generating annotated behavior models from end-user scenarios. *IEEE Transactions on Software Engineering*, 31:1056–1073.

[Ding et al., 2006] Ding, J., Gower, J. E., and Schmidt, D. S. (2006). Zhuang-zi: A new algorithm for solving multivariate polynomial equations over a finite field.

[Feng et al., 2011] Feng, L., Kwiatkowska, M., and Parker, D. (2011). Automated learning of probabilistic assumptions for compositional reasoning. In *Proceedings of the 14th international* *conference on Fundamental approaches to software engineering: part of the joint European conferences on theory and practice of software*, FASE'11/ETAPS'11, pages 2–17, Berlin, Heidelberg. Springer-Verlag.

[Han and Katoen, 2007] Han, T. and Katoen, J.-P. (2007). Counterexamples in probabilistic model checking. In *TACAS*, pages 72–86.

[Henrion et al., 2007] Henrion, D., bernard Lasserre, J., and Löfberg, J. (2007). Gloptipoly 3: moments, optimization and semidefinite programming.

[Henrion and Lasserre, 2002] Henrion, D. and Lasserre, J. (2002). Gloptipoly: Global optimization over polynomials with matlab and sedumi. *ACM Trans. Math. Soft*, 29:165–194.

[Hinton et al., 2006] Hinton, A., Kwiatkowska, M., Norman, G., and Parker, D. (2006). PRISM: A tool for automatic verification of probabilistic systems. In Hermanns, H. and Palsberg, J., editors, *Proc. 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06)*, volume 3920 of *LNCS*, pages 441–444. Springer.

[Hitzler et al., 2004] Hitzler, P., Hölldobler, S., and Seda, A. K. (2004). Logic programs and connectionist networks. *Journal of Applied Logic*, 2:2004.

[Kakas et al., 1993] Kakas, A. C., Kowalski, R. A., and Toni, F. (1993). Abductive logic programming.

[Kennedy and Eberhart, 2002] Kennedy, J. and Eberhart, R. (2002). Particle swarm optimization. In *Neural Networks, 1995. Proceedings., IEEE International Conference on*, volume 4, pages 1942–1948.

[Kimming, 2010] Kimming, A. (2010). *A Probabilistic Prolog and its Applications.* PhD thesis, Katholieke Universiteit Leuven.

[Knuth and Yao, 1976] Knuth, D. and Yao, A. (1976). *Algorithms and Complexity: New Directions and Recent Results*, chapter The complexity of nonuniform random number generation. Academic Press.

[Kwiatkowska et al., 2010] Kwiatkowska, M., Norman, G., Parker, D., and Qu, H. (2010). Assume-guarantee verification for probabilistic systems. In Esparza, J. and Majumdar, R., editors, *Proc. 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'10)*, volume 6105 of *LNCS*, pages 23–37. Springer.

[Manning and Schütze, 1999] Manning, C. D. and Schütze, H. (1999). *Foundations of statistical natural language processing.* MIT Press, Cambridge, MA, USA.

[Mitchell, 1997] Mitchell, T. M. (1997). *Machine learning.* McGraw Hill series in computer science. McGraw-Hill.

[Muggleton, 1996] Muggleton, S. (1996). Stochastic logic programs. In *New Generation Computing.* Academic Press.

[Muggleton, 2002] Muggleton, S. (2002). Learning structure and parameters of stochastic logic programs. In *ILP*, pages 198–206.

[Muggleton et al., 2008] Muggleton, S., Santos, J. C. A., and Tamaddoni-Nezhad, A. (2008). Toplog: Ilp using a logic program declarative bias. In *ICLP*, pages 687–692.

[Nienhuys-Cheng and Wolf, 1997] Nienhuys-Cheng, S.-H. and Wolf, R. d. (1997). *Foundations of Inductive Logic Programming.* Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[Păsăreanu et al., 2008] Păsăreanu, C. S., Giannakopoulou, D., Bobaru, M. G., Cobleigh, J. M., and Barringer, H. (2008). Learning to divide and conquer: applying the l* algorithm to automate assume-guarantee reasoning.

[Quinlan and Cameron-Jones, 1993] Quinlan, J. R. and Cameron-Jones, R. M. (1993). Foil: A midterm report. In *ECML*, pages 3–20.

[Raedt and Thon, 2010] Raedt, L. D. and Thon, I. (2010). Probabilistic rule learning. In *ILP*, pages 47–58.

[Rivest and Schapire, 1989] Rivest, R. L. and Schapire, R. E. (1989). Inference of finite automata using homing sequences. In *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, STOC '89, pages 411–420, New York, NY, USA. ACM.

[Rouillier and Zimmermann, 2004] Rouillier, F. and Zimmermann, P. (2004). Efficient isolation of polynomial's real roots. *J. Comput. Appl. Math.*, 162:33–50.

[Sato, 1995] Sato, T. (1995). A statistical learning method for logic programs with distribution semantics. In *IN PROCEEDINGS OF THE 12TH INTERNATIONAL CONFERENCE ON LOGIC PROGRAMMING*, pages 715–729. MIT Press.

[Sato and Kameya, 2001] Sato, T. and Kameya, Y. (2001). Parameter learning of logic programs for symbolic-statistical modeling. *Journal of Artificial Intelligence Research*, page 454.

[van Lamsweerde et al., 1998] van Lamsweerde, A., Darimont, R., and Letier, E. (1998). Managing conflicts in goal-driven requirements engineering. *IEEE Transactions on Software Engineering*, 24:908–926.

[van Lamsweerde and Letier, 2000] van Lamsweerde, A. and Letier, E. (2000). Handling obstacles in goal-oriented requirements engineering.

[van Lamsweerde and Willemet, 1998] van Lamsweerde, A. and Willemet, L. (1998). Inferring declarative requirements specifications from operational scenarios. *IEEE Transactions on Software Engineering*, 24:1089–1114.