Neuro-Symbolic Artificial Intelligence
Chapter 6
ProbLog

Nils Holzenberger

April 2, 2024

Outline

1. Probabilistic programming
   - Atoms
   - Predicates
   - Learning probabilities
2. Probabilities
3. ProbLog
   - Mechanics
     - Computing success probabilities
   - Options

Outline

1. Probabilistic programming
   - Atoms
   - Predicates
   - Learning probabilities
2. Probabilities
3. ProbLog
   - Mechanics
     - Computing success probabilities
   - Options

Problem

- Sometimes it's straightforward to determine the truth value of a predicate
  - `member(Element,List)`
  - `win(GameState)`, `loss(GameState)`
- Sometimes not
  - `is_cat(Image)`
  - Sentiment analysis

```
?- Sentence="This is a great vacuum cleaner if
             you're trying to ruin your carpet.",
   sentiment(Sentence,Polarity).
```

Goal

- Incorporate uncertainty into Prolog
- Incorporate learnable parameters into Prolog
    - Statistical machine learning
    - Neural networks — see next lecture
- Combine symbols (Prolog program) and neural networks

ProbLog

ProbLog = Prolog + probabilities

We introduce ProbLog which is — in a sense — the simplest probabilistic extension of Prolog one can design.

De Raedt *et al*, *ProbLog: A Probabilistic Prolog and Its Application in Link Discovery*, IJCAI 2007

ProbLog

- ProbLog is one of many probabilistic programming packages
- As far as I know it is very principled, and enjoys many extensions
    - Approximate and exact inference
    - Plugins for Pytorch

Weather

Example `weather.pl`

- Run queries
- Assert evidence

Poker dice

- Fair dice
- Biased dice

Monty Hall paradox

- The Monty Hall game
  - There are 3 doors. Behind one of them is a reward.
  - The player picks a door.
  - The game moderator opens a different door, revealing that there is no reward behind it.
  - The player can choose to keep the door picked at the beginning, or to pick the other closed door.
  - What is the best decision?
- Example `monty-hall.pl`
  - First, code a door-picking game (1 turn)
  - Second, code the Monty Hall game

Poker dice

Learning with ProbLog: `problog lfi myprogram.pl myexamples.pl`

- Learning the probability of an opponent cheating
- Learning the bias of the dice

Outline

1. Probabilistic programming
   - Atoms
   - Predicates
   - Learning probabilities
2. Probabilities
3. ProbLog
   - Mechanics
     - Computing success probabilities
   - Options

Probabilities

- What is a *probability*?
  - I toss a coin. What is the *probability* it lands on tails?
  - I throw two dice. What is the *probability* of getting a double six?

- A belief
  Measurement of my belief that the coin will land on tails
- The frequency of an outcome
  Frequency of outcome if I toss the same coin 10,000 times
- The ratio of monetary amounts people are willing to bet
  Predictive markets — possibly the most practical definition

Random variables

A *random variable* is a function that maps the outcome of an experiment to a value

Coin-flipping experiment:

$$X = \{ \quad \text{"the coin lands on heads"} \rightarrow X = 1, \\ \text{"the coin lands on tails"} \rightarrow X = 0 \quad \}$$

Poker game:

$$Y = \{ \quad \text{"my opponent cheated"} \rightarrow Y = 1, \\ \text{"my opponent did not cheat"} \rightarrow X = 0 \quad \}$$

$$Z = \{\text{"my opponent is dealt a royal flush"} \rightarrow Z = 1, ...\}$$

We can reason about the probability of $X = 1$, noted $p(X = 1)$

Random variables

- Random variables are not random
- Random variables are not variables
- Random variables are functions
- Random variables are deterministic
- The randomness comes from the outcome
- A random variable deterministically maps an outcome to a value

Adapted from Ryan Cotterell's Introduction to NLP

Why use probabilities in AI?

- There is theory about how to estimate probabilities from data samples
- They can efficiently model noisy processes
  - The process = the part of the mechanics we understand
  - The noise = the part we don't understand
- Probabilities can model deterministic processes

Useful properties

- Non-negativity $\forall x \in D, p(X = x) \geq 0 \ / \ \forall x \in D, f(x) \geq 0$
- Sums to 1 $\sum_{x \in D} p(X = x) = 1$
- Additivity If $A \subset B$ then $p(A) \leq p(B)$
- Joint probabilities $p(X = x, Y = y) \stackrel{\text{def}}{=} p(\{X = x\} \cap \{Y = y\})$
- Marginalization $p(X = x) = \sum_{y \in D_y} p(X = x, Y = y)$
- Conditional probabilities $p(X = x | Y = y) \stackrel{\text{def}}{=} \frac{p(X=x, Y=y)}{p(Y=y)}$

Example: probabilities in Natural Language Processing

Step 1. Express the quantities of interest as random variables.

eg spam classification:

Experiment = I receive an email

X = the email I receive (it's a string)

Y = 1 if the email is spam, 0 otherwise

Example: probabilities in Natural Language Processing

X = the email I receive (it's a string)

$Y = 1$ if the email is spam, 0 otherwise

$p(y\|x)$	Given that I received email x, is it spam?
$p(y)$	How probable is it that an email I receive should be spam?
$p(x)$	How probable is it that I should receive email x?
$p(x\|y)$	How probable is it that I should receive email x, assuming that it's spam/not spam?

Step 2. How to compute p(y|x)? → next lecture

Outline

1 Probabilistic programming
- Atoms
- Predicates
- Learning probabilities

2 Probabilities

3 ProbLog
- Mechanics
  - Computing success probabilities
- Options

Probability distributions over Prolog programs

- Experiment:
  - A ProbLog program is a set of Prolog clauses, each with a probability (weight in $[0,1]$)
  - We draw clauses from a ProbLog program, according to the probabilities
- Outcome: a set of clauses S
- Random variable X: $X = 1$ if $S \vdash G$ where G is a pre-defined query
- In Prolog we wanted to know whether or not G succeeds. In ProbLog, we get the probability that G succeeds — $p(X = 1)$
- How do we compute $p(X = 1)$? We enumerate all programs and their weights

De Raedt *et al*, *ProbLog: A Probabilistic Prolog (...)*, IJCAI 2007

Probability distributions over Prolog programs

A Prolog program L is a set $\{f_1, ..., f_m\}$ where f_i is a Prolog clause

A ProbLog program T is a set of Prolog clauses $C = \{c_1, ..., c_n\}$ and a function w that specifies each clause's probability $w(c_i)$

G is a clause whose probability we want to compute

- $p(L|T) = \prod_{c \in L} w(c) \prod_{c \in C \setminus L} 1 - w(c)$
  Probability of program L drawn from T
- $p(G|L) = 1 \text{ if } L \vdash G \text{ else } 0$
  Success probability of clause G given program L
- $p(G, L|T) = p(G|L)p(L|T)$
  Probability of clause G and program L under T
- $p(G|T) = \sum_{L \subset C} p(G, L|T)$
  Probability of clause G under T

⚠ We are abusing notation here

De Raedt *et al*, *ProbLog: A Probabilistic Prolog (...)*, IJCAI 2007

Weather example

cloudy	sunshine	raining	nice	funny	$p(L)$
T	T	T	F	T	0
T	T	F	T	F	0
T	F	T	F	F	$.3 * .8 = .24$
T	F	F	F	F	$.3 * .2 = .06$
F	T	T	F	T	0
F	T	F	T	F	0.7
F	F	T	F	F	0
F	F	F	F	F	0

The sum is 1

Weather example

Probability of `cloudy`: .3

`cloudy`	`sunshine`	`raining`	`nice`	`funny`	$p(L)$
T	T	T	F	T	0
T	T	F	T	F	0
T	F	T	F	F	$.3 * .8 = .24$
T	F	F	F	F	$.3 * .2 = .06$
F	T	T	F	T	0
F	T	F	T	F	0.7
F	F	T	F	F	0
F	F	F	F	F	0

Weather example

Probability of `nice`: .7

`cloudy`	`sunshine`	`raining`	`nice`	`funny`	$p(L)$
T	T	T	F	T	0
T	T	F	T	F	0
T	F	T	F	F	$.3 * .8 = .24$
T	F	F	F	F	$.3 * .2 = .06$
F	T	T	F	T	0
F	T	F	T	F	0.7
F	F	T	F	F	0
F	F	F	F	F	0

Weather example

Probability of `funny`: 0

`cloudy`	`sunshine`	`raining`	`nice`	`funny`	$p(L)$
T	T	T	F	T	0
T	T	F	T	F	0
T	F	T	F	F	$.3 * .8 = .24$
T	F	F	F	F	$.3 * .2 = .06$
F	T	T	F	T	0
F	T	F	T	F	0.7
F	F	T	F	F	0
F	F	F	F	F	0

- This is referred to as *model counting*
- This has the same issues as using truth tables to determine tautologies

SLD tree

```
% If X is a friend of Y, then X likes Y:
l(X,Y):- f(X,Y).
% If there is Z such that X is friends with Z and Z likes Y,
% then there is a 80% chance that X likes Y
0.8::l(X,Y):- f(X,Z), l(Z,Y).
% john is friends with mary with probability .5
0.5::f(john,mary).
0.5::f(mary,pedro).
0.5::f(mary,tom).
0.5::f(pedro,tom).
```

SLD tree

R1 `l(X,Y):- f(X,Y).`

R2 `0.8::l(X,Y):- f(X,Z), l(Z,Y).`

R3 `0.5::f(john,mary).`

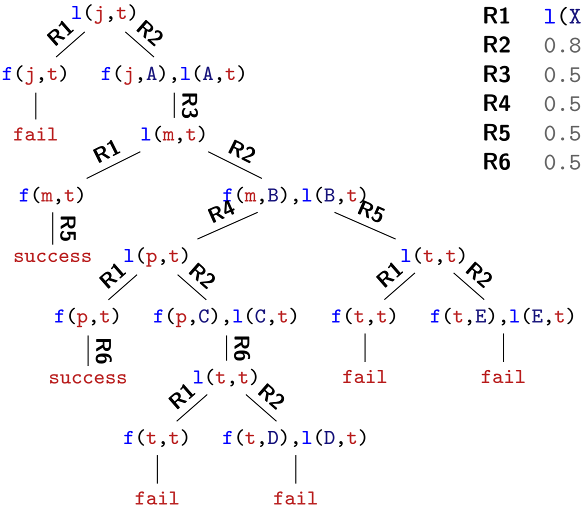
R4 `0.5::f(mary,pedro).`

R5 `0.5::f(mary,tom).`

R6 `0.5::f(pedro,tom).`

SLD tree

Query: `l(john,tom)`



R1	`l(X,Y):- f(X,Y).`
R2	`0.8::l(X,Y):- f(X,Z), l(Z,Y).`
R3	`0.5::f(john,mary).`
R4	`0.5::f(mary,pedro).`
R5	`0.5::f(mary,tom).`
R6	`0.5::f(pedro,tom).`

$Q = (R2 \wedge R3 \wedge R1 \wedge R5) \vee$
$(R2 \wedge R3 \wedge R2 \wedge R4 \wedge R1 \wedge R6)$

$Q = (R1 \wedge R2 \wedge R3 \wedge R5) \vee$
$(R1 \wedge R2 \wedge R3 \wedge R4 \wedge R6)$

De Raedt *et al*, *ProbLog: A Probabilistic Prolog (...)*, IJCAI 2007

SLD tree

In summary:

- Find all the ways of proving goal G
- Do this efficiently by using the trace of the proof by resolution
- $p(Q|T) = p(\bigvee_{b \in \text{proofs}(Q)} \bigwedge_{c \in \text{clauses}(b)} c)$

  $\text{proofs}(Q)$: the set of proofs for Q

  $\text{clauses}(b)$: the set of clauses that appear in proof b

→ but the paths are not disjoint, so in general $p(q|T) \neq \sum_{b \in pr(q)} \prod_{c \in cl(b)} p(c)$

De Raedt *et al*, *ProbLog: A Probabilistic Prolog (...)*, IJCAI 2007

Grounding

`l(t,j)` is *grounded*; `l(t,X)` is *not grounded*

- In some neuro-symbolic programming paradigms, the engine
  - grounds all formulas, then
  - computes the truth values of grounded atoms.
- The SLD tree only computes those groundings necessary for the proof
- In the previous example, $2 \times 4 \times 4 = 32$ groundings:

```
l(j,j)  l(m,j)  l(p,j)  l(t,j)  f(j,j)  f(m,j)  f(p,j)  f(t,j)
l(j,m)  l(m,m)  l(p,m)  l(t,m)  f(j,m)  f(m,m)  f(p,m)  f(t,m)
l(j,p)  l(m,p)  l(p,p)  l(t,p)  f(j,p)  f(m,p)  f(p,p)  f(t,p)
l(j,t)  l(m,t)  l(p,t)  l(t,t)  f(j,t)  f(m,t)  f(p,t)  f(t,t)
```

Binary Decision Diagrams

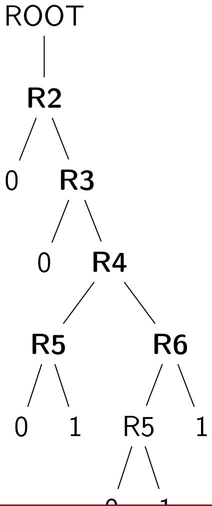
$Q = (R1 \wedge R2 \wedge R3 \wedge R5) \vee (R1 \wedge R2 \wedge R3 \wedge R4 \wedge R6)$

Computing the probability of DNF formulae is an NP-hard problem even if all variables are independent

- Binary decision diagrams represent the formula as a disjunction of disjoint conjunctions
- There are algorithms for efficient conversion

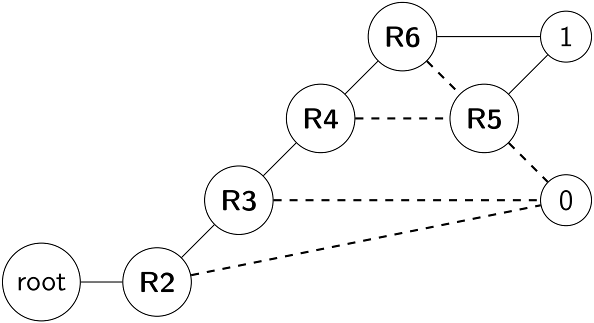
Binary Decision Diagrams

$$Q = (R1 \wedge R2 \wedge R3 \wedge R5) \vee (R1 \wedge R2 \wedge R3 \wedge R4 \wedge R6)$$



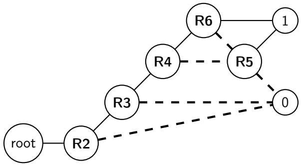
Binary Decision Diagrams

$$Q = (R1 \land R2 \land R3 \land R5) \lor (R1 \land R2 \land R3 \land R4 \land R6)$$



Binary Decision Diagrams

$$Q = (R1 \land R2 \land R3 \land R5) \lor (R1 \land R2 \land R3 \land R4 \land R6)$$



- Read off the 3 paths that end in 1:
  - **R2**, **R3**, ¬ **R4**, **R5**
  - **R2**, **R3**, **R4**, ¬ **R6**, **R5**
  - **R2**, **R3**, **R4**, **R6**

$$Q = (R2 \land R3 \land \neg R4 \land R5) \lor (R2 \land R3 \land R4 \land \neg R6 \land R5) \lor (R2 \land R3 \land R4 \land R6)$$

$$p(Q) = p_2 p_3 (1 - p_4) p_5 + p_2 p_3 p_4 (1 - p_6) p_5 + p_2 p_3 p_4 p_6$$

- Computing the BDD diagram:
  - Turn each successful proof in the SLD tree into a clause
  - Turn each clause into a BDD diagram
  - Merge diagrams (P-time)
  - Put diagram into canonical form (P-time)

De Raedt *et al*, *ProbLog: A Probabilistic Prolog (...)*, IJCAI 2007

Computing probabilities

- Use the Prolog engine to get all possible proofs
- Turn the SLD tree into a BDD diagram
- Read the probabilities off the BDD diagram

ProbLog options

- (default, no keyword): standard ProbLog inference
- sample: generate samples from a ProbLog program
- mpe: most probable explanation
- lfi: learning from interpretations
- dt: decision-theoretic problog
- map: MAP inference
- explain: evaluate using mutually exclusive proofs
- ground: generate a ground program
- bn: export a Bayesian network
- shell: interactive shell

https://problog.readthedocs.io/en/latest/cli.html

shell: interactive shell

```
problog shell
```

```
consult('file.pl')
```

shell: generate samples from a ProbLog program

```
problog sample likes.pl -N 10 --with-facts
```

mpe: most probable explanation

computing the possible world with the highest probability in which all queries and evidence are true

```
problog mpe likes.pl --full
```

lfi: learning from interpretations

next lecture

dt: decision-theoretic problog

File dt_model.pl:

```
0.3::rain.
0.5::wind.
?::umbrella.
?::raincoat.
broken_umbrella :- umbrella, rain, wind.
dry :- rain, raincoat.
dry :- rain, umbrella, not broken_umbrella.
dry :- not(rain).
utility(broken_umbrella, -40).
utility(raincoat, -20).
utility(umbrella, -2).
utility(dry, 60).
```

```
$ problog dt dt_model.pl
raincoat:       0
umbrella:       1
SCORE: 43.00000000000001
```

explain: evaluate using mutually exclusive proofs

```
problog explain likes.pl
```

ground: generate a ground program

```
problog ground likes.pl
```