CITS3005 Knowledge Representation

Lecture 06: Probabilistic Logic Programming

Tim French

The University of Western Australia

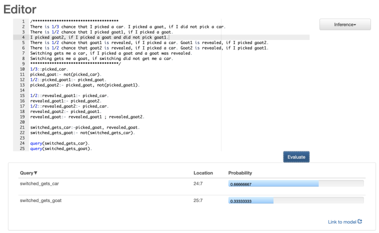
2023

Overview

In this lecture we will establish some basic properties of *probability* and *probabilistic reasoning*, and the look at these concepts in the context of *Problog*, a probabilistic logic programming language.

ProbLog2

ProbLog2 is a python3 library for probabilistic logic programming, produced at KU Leuven It and can be installed via pip:
`pip install problog`
and run from the command line (`>problog shell`), or via a web interface.



```
/***********************************
There is 1/3 chance that I picked a car. I picked a goat, if I did not pick a car.
There is 1/2 chance that I picked goat1, if I picked a goat.
I picked goat2, if I picked a goat and did not pick goat1.
There is 1/2 chance that goat1 is revealed, if I picked a car. Goat1 is revealed, if I picked goat2.
There is 1/2 chance that goat2 is revealed, if I picked a car. Goat2 is revealed, if I picked goat1.
Switching gets me a car, if I picked a goat and a goat was revealed.
Switching gets me a goat, if switching did not get me a car.
***********************************/
1/3::picked_car.
picked_goat:- not(picked_car).
1/2::picked_goat1:- picked_goat.
picked_goat2:- picked_goat, not(picked_goat1).

1/2::revealed_goat1:- picked_car.
revealed_goat1:- picked_goat2.
1/2::revealed_goat2:- picked_car.
revealed_goat2:- picked_goat1.
revealed_goat:- revealed_goat1 ; revealed_goat2.

switched_gets_car:-picked_goat, revealed_goat.
switched_gets_goat:- not(switched_gets_car).

query(switched_gets_car).
query(switched_gets_goat).
```

Query▼	Location	Probability
switched_gets_car	24:7	0.66666667
switched_gets_goat	25:7	0.33333333

Probability Theory

We will first look at the fundamental properties of probability, including:

- Events - things that happen.
- Dependence/Independence - Whether events are related.
- Distributions - The possible outcomes and likelihoods for events.
- Sampling - Selecting an event from a distribution
- Probability Spaces - the semantics of probabilities.
- Probability axioms - the logic of probability.

The concept of probability

- To describe variability of outcomes of repeatable experiments, e.g. chances of getting "Heads" in a flip of a coin, chances of failure of a component (mass production).
- To quantify the uncertainty of an outcome of a non-repeatable event. Here the probability will depend on the available information.
- To measure the present state of knowledge, e.g. the probability that the detected tumour is malignant.

Probabilities

Term *experiment* is used to refer to any process whose outcome is not known in advance. Consider an experiment.

- Sample space $\mathcal{S}$: A collection of all possible outcomes.
- Sample point $s \in \mathcal{S}$: An element in $\mathcal{S}$.
- Event A: A subset of sample points, $A \subset \mathcal{S}$ for which a statement about an outcome is true.

Rules for probabilities:

$$\Pr(A \cup B) = \Pr(A) + \Pr(B), \quad \text{if} \quad A \cap B = \emptyset.$$

For any event A,

$$0 \leq \Pr(A) \leq 1.$$

Statements which are always false have **probability zero**, similarly, always-true statements have **probability one**.

Kolmogorov Axioms

In probability theory, a probability space or a probability triple $(\Omega, \mathcal{F}, P)$ is a mathematical construct that provides a formal model of a random process or "experiment".

The elements of a probability space are

1. A sample space, Ω, which is the set of all possible outcomes.
2. An event space, which is a set of events $\mathcal{F}$, an event being a σ-algebra over Ω.
3. A probability function, P which assigns each event in the event space a probability, which is a number between 0 and 1.

A σ-algebra over Ω is a set of subsets of Ω, closed under intersection, complement and countable unions, and containing Ω.

The probability function P is countably additive, so if $\{A_i\}_{i=1}^{\infty} \subseteq \mathcal{F}$ is a countable collection of pairwise disjoint sets, then $P(\bigcup_{i=1}^{\infty} A_i) = \sum_{i=1}^{\infty} P(A_i)$, and $P(\Omega) = 1$.

How to find "useful" probabilities:

- Classically for finite sample spaces $\mathcal{S}$, if all outcomes are equally probable then

$$\Pr(A) = \text{number of outcomes for which } A \text{ is true/number of outcomes}$$

- Employ a concept of independence.
- Employ a concept of conditional probabilities.

If everybody agrees with the choice of Pr, it is called an **objective probability**. (If a coin is "fair" the probability of getting tails is 0.5.)
For many problems the probability will depend on the information a person has when estimating the chances that a statement A is true. One then speaks of **subjective probability**.

Probability concepts

- **Independence** For a sample space $\mathcal{S}$ and a probability measure Pr, the events $A, B \subset \mathcal{S}$ are called **independent** if $\Pr(A \cap B) = \Pr(A) \cdot \Pr(B)$ and otherwise they are **dependent**.
- **Conditional probability**

$$\Pr(B \mid A) = \frac{\Pr(A \cap B)}{\Pr(A)}$$

  The chances that some statement B is true when we *know* that some statement A is true.

- **Law of total probability** Let $A_1, \ldots, A_n$ be a partition of the sample space. Then for any event B

$$\Pr(B) = \Pr(B|A_1)\Pr(A_1) + \Pr(B|A_2)\Pr(A_2) + \cdots + \Pr(B|A_n)\Pr(A_n)$$

Bayes' formula

Again let $A_1, \dots, A_n$ be a partition of the sample space. The evidence is that B is true. Which of alternatives is most likely to be true?

Bayes' formula:

$$\Pr(A_i|B) = \frac{\Pr(A_i \cap B)}{\Pr(B)} = \frac{\Pr(B|A_i)\Pr(A_i)}{\Pr(B)} = \frac{\Pr(A_i)}{\Pr(B)}\Pr(B|A_i)$$

Name due to *Thomas Bayes* (1702-1761)

- **Likelihood:** $L(A_i) = \Pr(B|A_i)$ (How likely is the observed event B under alternative A_i?)
- Events B_1, B_2 are **conditionally independent** if for all i,

$$\Pr(B_1 \cap B_2 | A_i) = \Pr(B_1 \,|\, A_i)\Pr(B_2 \,|\, A_i).$$

and Events $B_1, \dots, B_n$ are conditionally independent if all pairs B_i, B_j are conditionally independent.

Bayesian Inference

Bayesian inference computes the posterior probability according to Bayes' theorem:

$$P(H \mid E) = \frac{P(E \mid H) \cdot P(H)}{P(E)}$$

- H stands for any hypothesis whose probability may be affected by data (called evidence below).
- $P(H)$, the prior probability, is the estimate of the probability of the hypothesis H before the data E, the current evidence, is observed.
- E, the evidence, corresponds to new data that were not used in computing the prior probability.
- $P(H \mid E)$, the posterior probability, is the probability of H given E, i.e., after $E$is observed. (What we want to know).
- $P(E \mid H)$ is the probability of observing E given H, and is called the likelihood. As a function of E with H, it indicates the compatibility of the evidence with the given hypothesis.
- $P(E)$ is the marginal likelihood or model evidence, and is the same for all possible hypotheses being considered.

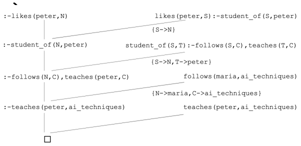
Therefore, given a discrete finite set of events $E_1, \ldots E_n \in \mathcal{F}$, the *full joint probability distribution* is the set of all conditional probabilities of sets of events. This can be thought of as a state of belief, or an approximate state of

Logic Programming (recap)

Recall the elements of logic programming from the first part of the course.

- A logic program consists a set of clauses.
- The clauses contain predicates, operating on terms, which can be variables, constants, or functions of terms.
- SLD resolution works by taking a query term and systematically, performing substitution of variables and resolutions to deduce a contradiction.

```
likes(peter,S):-student_of(S,pete
student_of(S,T):-follows(S,C),tea
teaches(peter,ai_techniques).
follows(maria,ai_techniques).
```



:-likes(peter,N)
likes(peter,S):-student_of(S,peter)
{S->N}
:-student_of(N,peter)
student_of(S,T):-follows(S,C),teaches(T,C)
{S->N,T->peter}
:-follows(N,C),teaches(peter,C)
follows(maria,ai_techniques)
{N->maria,C->ai_techniques}
:-teaches(peter,ai_techniques)
teaches(peter,ai_techniques)
□

Probabilistic Logic Programming

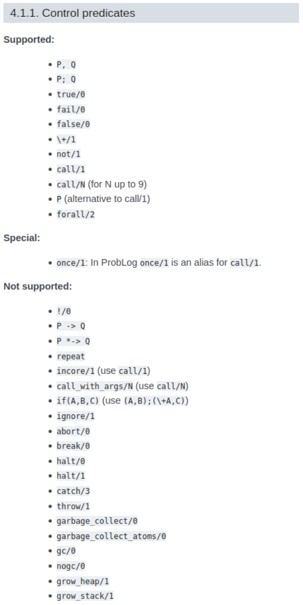
Probabilistic logic programming treats the ground literals of a logic program as probabilistic events.

- When a program is *evaluated* the interpreter considers the probability space of programs, and calculates the probability of a given query literal.
- When a set of literals is *sampled* the interpreter samples the probability space of programs and reports the value (true/false) of those literals.
- When a variable is *learnt* the interpreter takes a program and some evidence, and applies Bayesian inference to create a new model, based on the evidence.

A probabilistic logic program is essentially a *probability space of logic programs* which gives a much richer semantics for representing uncertain knowledge.

ProbLog vs Prolog

ProbLog supports a subset of the Prolog language for expressing models in probabilistic logic.
The main difference between ProbLog's language and Prolog is that Prolog is a complete logic programming language, whereas ProbLog is a logic representation language. This means that most of the functionality of Prolog that is related to the programming part (such as control constructs and input/output) are not supported in ProbLog.



4.1.1. Control predicates

Supported:

- `P, Q`
- `P; Q`
- `true/0`
- `fail/0`
- `false/0`
- `\+/1`
- `not/1`
- `call/1`
- `call/N` (for N up to 9)
- `P` (alternative to call/1)
- `forall/2`

Special:

- `once/1`: In ProbLog `once/1` is an alias for `call/1`.

Not supported:

- `!/0`
- `P -> Q`
- `P *-> Q`
- `repeat`
- `incore/1` (use `call/1`)
- `call_with_args/N` (use `call/N`)
- `if(A,B,C)` (use `(A,B);(\+A,C)`)
- `ignore/1`
- `abort/0`
- `break/0`
- `halt/0`
- `halt/1`
- `catch/3`
- `throw/1`
- `garbage_collect/0`
- `garbage_collect_atoms/0`
- `gc/0`
- `nogc/0`
- `grow_heap/1`
- `grow_stack/1`

Probabilistic facts

Likelihoods can be directly assigned to literals in the logic program.

```
% Probabilistic facts:
0.5::heads1.
0.6::heads2.

% Rules:
twoHeads :- heads1, heads2.

% Queries:
query(heads1).
query(heads2).
query(twoHeads).
```

Evaluating this program, `problog coin.pl` returns a probability, `twoheads : 0.3`, rather than `yes.` or `no.`

Noisy-OR

Consider a variant of the above example in which we are just interested in at least one coin landing heads. This example shows a useful aspect of the ProbLog language:
multiple rules with the same head lead to a *noisy-or* effect in ProbLog.

```
% Probabilistic facts:
0.5::heads1.
0.6::heads2.

% Rules:
someHeads :- heads1.
someHeads :- heads2.

% Queries:
query(someHeads).
```

We calculate the probability `someHeads` using the *noisy-or* formula:

$$\begin{aligned} P(someHeads) &= 1-(1-P(heads1))(1-P(heads2)) \\ &= 1-(1-0.5)(1-0.6) = 0.8. \end{aligned}$$

Probabilistic clauses

Rather than just applying probabilities to ground literals, we can apply probabilities to predicates and rules as well:

```
% Probabilistic facts:
0.6::heads(C) :- coin(C).

% Background information:
coin(c1).
coin(c2).
coin(c3).
coin(c4).

% Rules:
someHeads :- heads(_).

% Queries:
query(someHeads).
```

This gives a result for `someHeads` of 0.9744. The probabilistic rule is just "syntactic sugar" for guarding a rule with a probabilistic atom and each grounding of a probabilistic predicate is treated as an independent variable.

Annotated Disjunctions

We don't always want our variables to be independent. An *annotated disjunction* can be used to choose exactly one of a number of alternatives (if their probabilities sum to 1.0).

```
  % annotated disjunctions
1/6::one1; ... ; 1/6::six1.
3/20::one2; ... ;1/4 ::six2.

% Rules:
twoSix :- six1, six2.
someSix :- six1.
someSix :- six2.

% Queries:
query(six1).
query(six2).
query(twoSix).
query(someSix).
```

Again, this could be achieved using probilistic literals and the negation operator \+.

Recursion and Lists

The following example is a dice game where every roll determines the next die to roll, but we stop if we have used that die before.
It uses three sided dice, and calulates the probability of each possible outcome.

```
1/3::dice(1,D); 1/3::dice(2,D);
1/3::dice(3,D) :- die(D).

die(X) :- between(1,3,X).

roll(L) :- next(1,[1],L).

next(N,Seen,Rev) :- dice(Now,N),
   member(Now,Seen),
   reverse(Seen,[],Rev).
next(N,Seen,List) :-  dice(Now,N),
   \+ member(Now,Seen),
   next(Now,[Now|Seen],List).

member(X,[X|_]).
member(X,[_|Z]) :- member(X,Z).

reverse([],L,L).
reverse([H|T],A,L) :- reverse(T,[H|A],L).

query(roll(_)).
```

Problog as a Probability Space

A ProbLog program consists of two parts:

- a set of ground probabilistic facts. A ground probabilistic fact, written `p::f`, is a ground fact `f` annotated with a probability `p`.
- and a logic program, i.e. a set of rules and ('non-probabilistic') facts.

The set of ground probabilostoc facts give a probability space of interpretations for the program.

For example the following programs are equivalent:

```
0.7::red(X);
  0.3::green(X)
  :- ball(X).

ball(a).
ball(b).
ball(c).

query(red(_)).
query(green(_)).
```

```
0.7::choose_red_cprob(X).
1.0::choose_green_cprob(X).
choose_red(X)   :- choose_red_cprob(X).
choose_green(X) :- \+choose_red(X),
                   choose_green_cprob(X).
red(X)   :- ball(X), choose_red(X).
green(X) :- ball(X), choose_green(X).

ball(a). ball(b). ball(c).

query(red(_)).
query(green(_)).
```

Flexible Probabilities

In problog, probabilities can be treated as variables that will be unified by SLD resolution. Below this is used in the intensional probabilistic fact `P::pack(Item) :-`, the probability of packing an item is inversely proportional to its weight.

```
weight(skis,6).
weight(boots,4).
weight(helmet,3).
weight(gloves,2).

% intensional probabilistic fact with flexible probability:
P::pack(Item) :- weight(Item,Weight),  P is 1.0/Weight.

xs(Lim) :- xs([skis,boots,helmet,gloves],Lim).
xs([],Lim) :- Lim<0.
xs([I|R],Lim) :- pack(I), weight(I,W), L is Lim-W, xs(R,L).
xs([I|R],Lim) :- \+pack(I), xs(R,Lim).
query(xs(8)).
```

Bayesian Networks

Suppose there is a burglary in our house with probability 0.7 and an earthquake with probability 0.2. Whether our alarm will ring depends on both burglary and earthquake:

- if there is a burglary and an earthquake, the alarm rings with probability 0.9;
- if there is only a burglary, it rings with probability 0.8;
- if there is only an earthquake, it rings with probability 0.1;
- if there is neither a burglary nor an earthquake, the alarm doesn't ring.

This may be represented by a simple Bayesian Network with 3 nodes, or the following logic program:

```
0.7::burglary. 0.2::earthquake.
0.9::p_alarm1. 0.8::p_alarm2. 0.1::p_alarm3.

alarm :- burglary, earthquake, p_alarm1.
alarm :- burglary, \+earthquake, p_alarm2.
alarm :- \+burglary, earthquake, p_alarm3.

evidence(alarm,true).
```

Learning from Evidence

Rather than specifying probabilities , we can set probabilities to be learnt from evidence inone of three possible forms.

- Of the form `t(_)::p_alarm1`. This indicates that the probability of this fact is to be learned from data, and initialises its value randomly.
- Of the form `t(0.5)::burglary`. This indicates that the probability of this fact is to be learned from data, but will initialise the probability as 0.5
- Of the form `0.2::earthquake`. This indicates that the probability of this fact is fixed (not learned).

Learning a model is different to evaluating a program. It requires:

1. An initial model, with learnable parameters, and
2. a set of evidence. These are separated by dash lines `----`, indicating independent samples.

Executing the learning function outputs the learning model.

Learning from Evidence

```
 %%% The program:
t(0.5)::burglary.
0.2::earthquake.
t(_)::p_alarm1.
t(_)::p_alarm2.
t(_)::p_alarm3.

alarm :- burglary,
  earthquake, p_alarm1.
alarm :- burglary,
  \+earthquake, p_alarm2.
alarm :- \+burglary,
  earthquake, p_alarm3.
```

```
%%% The data:
evidence(burglary,false).
evidence(alarm,false).
-----
evidence(earthquake,false).
evidence(alarm,true).
evidence(burglary,true).
-----
evidence(burglary,false).
```

The command `$ problog lfi prog.pl example.pl -O learned.pl` outputs a learned program substituing teh trained variables.

Sampling

Problog can also be used to sample program executions:

```
0.7::leaf(T).
0.5::operator('+',T) ; 0.5::operator('-',T).

Px::l(x,T); P::l(0,T); P::l(1,T) :- Px = 0.5, P is (1-Px)/2.

expr(A) :- expr(A,1,R).

expr(L,T1,T2) :- leaf(T1), T2 is T1+1, l(L,T1).
expr(S,T1,T2) :- \+ leaf(T1), Ta is T1+1,
   expr(E1,Ta,Tb), expr(E2,Tb,T2),
   operator(Operator,Ta),
   S =.. [Operator,E1,E2].

query(expr(A)).
```

The command `$ problog sample prog.pl` will produce a random expression like `expr('+'('+'('-'(1,0),x),1))`.

Decision Theory

This program states there is a chance that it will rain and will be windy. Utility is given and the decisions are whether to bring an umbrella or a raincoat.

```
% probabilistic facts
0.3::rain. 0.5::wind.
% decision facts
?::umbrella. ?::raincoat.

broken_umbrella :- umbrella, rain, wind.
dry :- rain, raincoat.
dry :- rain, umbrella, not broken_umbrella.
dry :- not(rain).
% utilities
utility(broken_umbrella, -40).
utility(raincoat, -20).
utility(umbrella, -2).
utility(dry, 60).
```

The command `$ problog dt prog.pl` will produce an optimal decision, in this case, take an umbrella.