Probabilistic Logic Programming

Xin Zhang
Peking University

Recap of Last Lecture

- Learning in probabilistic programming
  - Parameter learning, structure learning
  - Still an active research area

This Lecture

- Probabilistic logic programming
    - Motivation
    - Syntax
    - Semantics
    - Inference

Classical AI: Logic

- Rich logic systems provide significant expressiveness power
  - Concise and learnable models

- Example: first-order logic. Rules of chess occupy
  - 10^0 pages of first-order logic
  - 10^5 pages in propositional logic
  - 10^{38} pages in finite automata

Quick Recap on First-Order Logic

- Compared to propositional logic, introduces predicates and quantifications for expressiveness

$$\forall\, h1, h2, h3.\, sibling(h1, h2) \wedge sibling(h2, h3) \rightarrow sibling(h1, h3)$$

- Undecidable

Modern AI: Probability Theory for Uncertainty

- Bayesian network

- Fixed variables in fixed ranges
  - Similar to propositional logic and Boolean logic

Probabilistic Logic Programming: Unifying Logic and Probability

- Logic: the ability to describe complex domains concisely in terms of objects and relations

- Probability: the ability to handle uncertainty

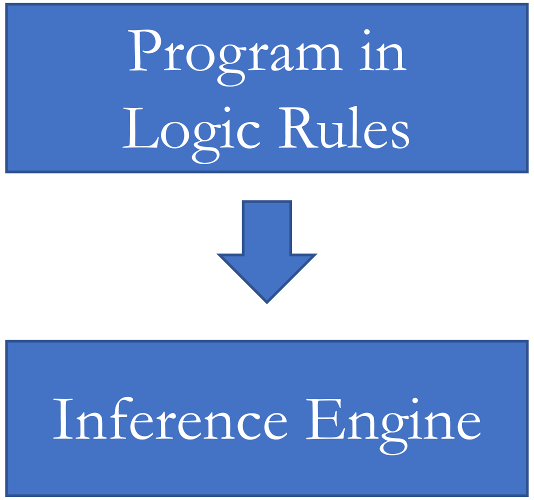
- Logic + probability = Probabilistic Logic Programming

Example Probabilistic Logic Languages

- Markov Logic Network. University of Washington
- Probabilistic Soft Logic. University of Maryland
- **Problog**. KU Leuven https://dtai.cs.kuleuven.be/problog/index.html
- BLOG. UC Berkeley
- ….

Background: Logic Programming

- Declarative: specifies what rather than how



- Leverages powerful inference engine

Background: Prolog and Datalog

- Prolog: once popular in AI, still being used in pattern matching (NLP)
    - Turning-complete

- Datalog: a subset of Prolog
    - Can only express polynomial algorithms
    - Originates from the Database community (SQL with recursions)
    - Logic part of Problog

Background: Datalog

Input Relation:

Edge(e1, e2)

Output Relation:

Path(e1, e2)

Rules:

Path(e1, e2) :- edge(e1, e2) $\forall\, edge(e_1, e_2) \Rightarrow path(e_1, e_2)$

Path(e1, e3) :- path(e1, e2), edge(e2, e3) $\forall\, path(e_1, e_2) \wedge edge(e_2, e_3) \Rightarrow path(e_1, e_2)$

Background: Datalog

Edge(1, 2)   Edge(2, 3)

Path(1, 2) :- Edge(1, 2)
Path(2, 3) :- Edge(2, 3)

Path(1, 3) :- Path(1, 2), Edge(2, 3)

Adding Probabilities to Datalog

- If A is a friend of B, and B is a friend of C, then A is likely a friend of C.

Can you write a program for the above sentence?

Adding Probabilities to Datalog

- Suppose edges exist with probabilities (by observation), compute path reachability.

Can you write a program for the above sentence?

Add probabilities to rules or facts?

What is the semantics?

Path(E1, E2) :- edge(E1, E2)
0.5: Path(E1, E3) :- path(E1, E2), edge(E2, E3)

Given a set of derived tuples/facts, assign a probability to them.

Problog: Introduction

- A language developed by the group led by Luc De Raedt at KU Leuven

- Extends Prolog with probabilities
  - Actually closer to Datalog

Problog: Syntax

- **Value**: numbers, mixed numbers and letters starting with a letter in lower cases

- **Variable**: starting with a capital letter

Problog: Syntax

Definition	Example
fact	`a.`
probabilistic fact	`0.5::a.`
clause	`a :- x.`
probabilistic clause	`0.5::a :- x.`
annotated disjunction	`0.5::a; 0.5::b.`
annotated disjunction	`0.5::a; 0.5::b :- x.`

From the documentation of Problog

Problog: Syntax

Queries:

```
0.5::heads(C).
two_heads :- heads(c1), heads(c2).
query(two_heads).
```

```
0.5::heads(C) :- between(1, 4, C).
query(heads(C)).
```

```
0.5::heads(C) :- between(1, 4, C).
query(heads(C)).
```

Evidence:

```
0.5::heads(C).
two_heads :- heads(c1), heads(c2).
evidence(\+ two_heads).
query(heads(c1)).
```

From the documentation of Problog

Example Program I

```
0.5 :: stayUp.
0.7 :: drinkCoffee :- stayUp.
0.5 :: drinkCoffee :- \+ stayUp.
0.9 :: fallSleep :- \+ drinkCoffee, stayUp.
0.3 :: fallSleep :- drinkCoffee, stayUp.
0.1 :: fallSleep :- \+stayUp.

evidence(fallSleep).

query(stayUp).
```

What does the following program compute?

```
0.5 :: stayUp.
0.7 :: drinkCoffee :- stayUp.
0.5 :: drinkCoffee :- \+ stayUp.
0.9 :: fallSleep :- \+ drinkCoffee, stayUp.
0.3 :: fallSleep :- drinkCoffee, stayUp.
0.1 :: fallSleep :- \+stayUp.

query(stayUp).

evidence(fallSleep).
```

What does the following program compute?

```
0.5::heads1.
0.5::heads2.

heads1 :- heads2.

query(heads1).
query(heads2).
```

What does the following program compute?

```
0.5::heads1.
0.5::heads2.

\+ heads1 :- heads2.

query(heads1).
query(heads2).
```

Example Program 2

```
0.9 :: edge(0,1).
0.8 :: edge(1,2).
0.7 :: edge(2,3).
0.8 :: edge(2,4).

1 :: path(A,B) :- edge(A,B).
0.8 :: path(A,C) :- path(A,B), edge(B,C).

evidence(\+ path(0,3)).

query(path(0,4)).
```

Semantics of Problog

- What is the semantics of the following program?

```
0.5 :: stayUp.
0.7 :: drinkCoffee :- stayUp.
0.3 :: fallSleep :- drinkCoffee, stayUp.

query(fallSleep).
```

Semantics of Problog

- For simplicity, we assume all probabilities are attached to facts

- First idea: we can convert the program into a Bayesian network, but how?

Semantics of Problog

- Converting into a Bayesian network is viable, but there are small catches

- We give another semantics that defines a distribution of Datalog programs

Semantics of Problog

- From a Problog program, we can sample a Datalog program by sampling the facts

```
0.5 :: stayUp.
0.7 :: drinkCoffee :- stayUp.
0.3 :: fallSleep :- drinkCoffee, stayUp.
```

=

```
0.5 :: stayUp.
0.7 :: r1.
0.3 :: r2.
drinkCoffee :- stayUp, r1.
fallSleep :- drinkCoffee, stayUp, r2.
```

sample →

```
stayUp.
r1.
r2.
drinkCoffee :- stayUp, r1.
fallSleep :- drinkCoffee, stayUp, r2.
```

Probability: 0.5*0.7*0.3

Semantics of Problog

- What about queries?

```
0.5 :: stayUp.
0.7 :: r1.
0.3 :: r2.
drinkCoffee :- stayUp, r1.
fallSleep :- drinkCoffee, stayUp, r2.

query(fallSleep)
```

A query calculates a marginal probability of a fact. Informally,

$$p(f) = \frac{\sum p(any\ program\ that\ derives\ f)}{\sum p(any\ program)}$$

Semantics of Problog

- What about evidence?

```
0.5 :: stayUp.
0.7 :: r1.
0.3 :: r2.
drinkCoffee :- stayUp, r1.
fallSleep :- drinkCoffee, stayUp, r2.

evidence(\+ fallSleep)
query(stayUp)
```

Evidence filters out certain programs. Informally,

$$p(f) = \frac{\sum p(any\ program\ that\ derives\ f|evidence)}{\sum p(any\ program|evidence)}$$

Semantics of Problog

- What about relations and quantified variables?

```
0.9 :: edge(0,1).
0.8 :: edge(1,2).
0.7 :: edge(2,3).
0.8 :: edge(2,4).

path(A,B) :- edge(A,B).
0.8 :: path(A,C) :- path(A,B), edge(B,C).

evidence(\+ path(0,3)).

query(path(0,4)).
```

Semantics of Problog

- Move probabilities to facts

```
0.9 :: edge(0,1).
0.8 :: edge(1,2).
0.7 :: edge(2,3).
0.8 :: edge(2,4).
0.8 :: r(A,B,C).

path(A,B) :- edge(A,B).
path(A,C) :- path(A,B), edge(B,C), r(A,B,C).

evidence(\+ path(0,3)).

query(path(0,4)).
```

Semantics of Problog

- Ground

  Constants: 0, 1, 2, 3 4

  path(A,C) :- path(A,B), edge(B,C), r(A,B,C).

  Generates

  path(0,0) :- path(0,0), edge(0,0), r(0,0,0). A=0, B=0, C=0

  path(0,1) :- path(0,0), edge(0,1), r(0,0,1). A=0, B=0, C=1

  path(0,1) :- path(0,0), edge(0,1), r(0,0,1). A=0, B=0, C=1

  …

Semantics of Problog

- After grounding, each ground term can be seen as a Boolean variable, then the whole program can be solved using the semantics of the Boolean case

path(0,0) -> t1, edge(0,0) -> t2, r(0,0,0) -> t3

path(0,0) :- path(0,0), edge(0,0), r(0,0,0).

t1 :- t1,t2,t3

Semantics of Problog

- First, ground the program into a Boolean program

- The Boolean program describes a distribution of Datalog program, which in turn defines a distribution of outputs

Questions

- Can you use Problog to express uniform distributions?

- What about loops?

Logic Part in Problog is more than Datalog

```
:- use_module(library(aggregate)).

pull(0).
count(1).

pull(N+1) :- pull(N), N < 10.
0.1 :: pull_SSR(N) :- pull(N).

num_SSRs(sum<X>) :- pull_SSR(N),count(X).

query(num_SSRs(X)).
```

But It is also Not Prolog

- The following program terminates in Problog but not in Prolog

```
child(anne,bridget).
child(bridget,caroline).
child(caroline,donna).
child(donna,emily).
descend(X,Y) :- descend(Z,Y), child(X,Z).
descend(X,Y) :- child(X,Y).

query(descend(anne,emily))
```

Inference

- As described before, inference can be done in two steps:
  - **Grounding**. Convert the program into a probabilistic program with only Boolean variables (no quantifiers)
  - **Solving**. Solve with the Boolean program produced above.

Optimization on Grounding

- Grounding replaces all variables with their values
  - Number of grounded rules is proportional to cartesian product of the domain sizes

- How to optimize?
  - A simple idea: only ground the part that is relevant to the queries and evidence.
  - Backtrack over the rules starting from the queries and evidence (SLD resolution).
  - A further optimization: stop tracking if a rule body doesn't hold according to the evidence

Optimization on Grounding

- If the logic part is Datalog without negation, we can use a Datalog solver to compute the grounding

- Datalog without negation is monotonic: the more rules or input facts, the more output facts

- If negation is on the input, it is still fine

Negation in Problog

- Unfortunately, Problog allows the following program:

```
one(1).
odd(X) :- one(X).
even(X) :- \+ odd(X).
```

And

```
0.5::a.        0.9 :: e:-a.
0.5::b.        0.9 :: e:-b.
               0.1 :: \+e:-a,b
```

If such negations are not present, we can use a Datalog solver to ground, which is highly efficient.

Solving

- Once we have a grounded program, we can leverage existing techniques

- Idea 1: convert the program into a Bayesian network

- Idea 2: convert the program into a Boolean formula with weights (MaxSAT)

Solving: Converting into a Bayesian Net

```
0.8 :: a.
0.7 :: b.

0.5 :: c:- a.
0.5 :: c:- b.

query(c).
```

Solving: Converting into a Bayesian Net

- We move all probabilities to input facts
- We add a root node whose prior distribution is P(r =1) = 1. Then we add a rule p :: f:-r for each input fact p::f
- For each fact f, suppose it is derived using r1, …, rn, we add arcs from all facts in the rule bodies to f.
- We set conditional probabilities:

$$p(f \mid \vee\, body(r_i) == True) = 1$$
$$p(f \mid \vee\, body(r_i) == False) = 0$$

Only works for program without cycles

Solving: Converting into a MaxSAT

- Finding the most likely solution becomes solving the MaxSAT

- Computing marginal probabilities becomes weighted model counting

Brief Introduction on MaxSAT

MaxSAT:

	$a \quad \wedge$	(C1)
	$\neg a \vee b \quad \wedge$	(C2)
4	$\neg b \vee c \quad \wedge$	(C3)
2	$\neg c \vee d \quad \wedge$	(C4)
7	$\neg d$	(C5)

=

Subject to	**C1** **C2**
Maximize	**4×C3+2×C4+7×C5**

Solution: a = true, b = true, c = true, d = false

(Objective = 11)

Brief Introduction on MaxSAT

- Popular MaxSAT solving techniques: converting the problem into a series of SAT problem

- Brief idea: can any solution satisfy k clauses?
  - Linear search
  - Binary search
  - (UNSAT) core guided

Core-Guided MaxSAT Solving

- UNSAT core: a set of clauses which are not unsatisfiable
  - Minimum UNSAT core: removing any clause will make it satisfiable
  - Modern SAT solvers come with the ability to return UNSAT cores

- [Fu & Malik]: Each time allow one and only one clause to be relaxed

Example using MaxSAT for Inference

0.6 :: rain.

0.5 :: sprinkle.

0.9 :: grass_wet :- rain, sprinkle.

0.6 rain

0.4 !rain

0.5 sprinkle

0.5 !sprinkle

0.9 r

0.1 !r

grass_wet or !rain or !sprinkle or !r

!grass_wet or rain

!grass_wet or sprinle

!grass_wet or r

grass_wet :- rain, sprinkle is translated into
$grass_{wet} \leftrightarrow rain \wedge sprinkle \wedge r$

Example using MaxSAT for Inference

- When translating rules, we have to consider the least fixed point semantics of Datalog

- Suppose the rules are acyclic, for a given fact f, we have to consider all grounded rules that derive f

$$f \leftrightarrow \vee\, body(r_i)$$

Example using MaxSAT for Inference

- When rules are cyclic, problems become complicated:

  0.5::a.   b:-a.   b:-c.   c:-b

- For reference:
  - Janhunen, T. 2004. Representing normal programs with clauses. In In Proc. of the 16th European Conference on Artificial Intelligence. IOS Press, 358–362.
  - Mantadelis, T. and Janssens, G. 2010. Dedicated tabling for a probabilistic setting. In Tech. Comm. of 26th International Conf. on Logic Programming. 124–133.

Brief Introduction on Weighted Model Counting

- Model counting: compute the number of assignments to a SAT expression

a or b   3 assignments

- Weighted model counting
  - Each variable has a weight for each assignment: w(v)
  - The model weight is the the product of variable weights
  - Now the count is a weighted sum

Example using WMC for Inference

0.6 rain

0.4 !rain

0.5 sprinkle

0.5 !sprinkle

0.9 r

0.1 !r

grass_wet or !rain or !sprinkle

!grass_wet or rain

!grass_wet or sprinle

w(rain =true) = 0.6
w(rain =false) = 0.4
w(sprinkle = true) = 0.5
w(sprinkle = false) = 0.5
w(r = true) = 0.9
w(r = false) = 0.1

P(grass_wet = true) = WMC(M∧grass_wet=true)

What if we want to evaluate
P(rain | grass_wet = true)?

Using WMC for Marginal Inference

- Let the constructed weighted formula be M, queries be Q, evidence be E, then

$$P(Q) = \frac{WMC(M \land Q \land E)}{WMC(M \land E)}$$

- For more, refer to

Sang, T., Beame, P. and Kautz, H., 2005. Solving Bayesian networks by weighted model counting. In Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-05) (Vol. 1, pp. 475-482). AAAI Press.

Further Reading on Problog

- https://dtai.cs.kuleuven.be/problog/index.html

Next Lecture

- Causality