Learning Logic Programs with Annotated Disjunctions

Hendrik Blockeel
K.U.Leuven

With thanks to Joost Vennekens, Daan Fierens, and many others from the Leuven ML gang
Overview

- Introduction: LPADs (a.k.a. CP-logic)
- Learning LPADs
  - Existing approaches and their limitations
  - Two new approaches
    - 1: mapping acyclic LPADs onto Bayesian networks with unobserved variables
    - 2: gradient descent approach for learning LPADs directly
  - Some experiments
LPADs: logic programs with annotated disjunctions

- Introduced in 2004 by Vennekens et al.
- Uses disjunctive clauses
  - 1 rule: $h_1 \lor h_2 \lor \ldots \lor h_m \leftarrow b_1 \land b_2 \land \ldots \land b_n$
- In an LPAD, each disjunct in the head is annotated with the probability that this disjunct is “selected” when the rule fires
  - $h_1 : \alpha_1 \lor h_2 : \alpha_2 \lor \ldots \lor h_m : \alpha_m \leftarrow b_1 \land b_2 \land \ldots \land b_n$
- Vennekens et al. (2004) formally define the semantics of LPADs
  - An LPAD defines a probability distribution over interpretations
Example LPADs

Heads: 0.5, tails: 0.5 <- toss.
Toss: 0.2.

\{toss, heads\} 0.1
\{toss, tails\} 0.1
\{\}\ 0.8

Heads: 0.5, tails: 0.5 <- toss.
Example LPADs

```
result(N,1) : 0.16, result(N,2) : 0.16, ..., result(N,6) : 0.16 <- die_roll(N).
die_roll(1).
die_roll(2) : 0.5.
```

```
result(1,1) : 0.16, ..., result(1,6) : 0.16 <- die_roll(1).
result(2,1) : 0.16, ..., result(2,6) : 0.16 <- die_roll(2).
die_roll(1).
die_roll(2) : 0.5.
```

```
result(1,5) <- die_roll(1).  % 1/6
result(2,3) <- die_roll(2).  % 1/6
die_roll(1) <- .               % 1
die_roll(2) <- .               % 1/2
```

```
\{die_roll(1), result(1,5)\} 1/12
\{die_roll(1), die_roll(2), result(1,5), result(2,3)\} 1/72
```
CP-logic: reformulation of LPAD framework in terms of causality (Vennekens, Denecker and Bruynooghe, submitted)

- Identification of **two basic principles** that make CP-logic a more natural formalism than, e.g., Bayesian nets:
  - Independent causation
  - No “deus ex machina”
- Causal structure can be described more accurately than with bayesian nets
  - E.g., CP-logic allows us to formulate cyclic causal structures

```
hiv(a) : 0.6 ← hiv(b).
hiv(b) : 0.6 ← hiv(a).
hiv(a) : 0.01.
hiv(b) : 0.01.
```
Interpretation of $\alpha_i$

- In "$h_1 : \alpha_1 \lor h_2 : \alpha_2 \lor \ldots \lor h_m : \alpha_m \leftarrow B$", it is in general **not correct** to interpret $\alpha_i$ as $P(h_i|B)$.

- All we can say is that
  - $P(h_i|B) \geq \alpha_i$
  - $P(h_i|B) = \alpha_i$ if $B$ is the only possible "cause" for $h_i$ being true

```plaintext
hair_wet: 0.7 <- raining.
hair_wet: 0.4 <- gone_swimming.
raining: 0.5.
gone_swimming: 0.5.
```

```plaintext
{}: 0.5 * 0.5 = 0.25
{raining}: 0.25 * 0.3
{raining, hair_wet}: 0.25 * 0.7
{gone_swimming}: 0.25 * 0.6
{gone_swimming, hair_wet}: 0.25 * 0.4
{raining, gone_swimming}: 0.25 * 0.3 * 0.6
{raining, gone_swimming, hair_wet}: 0.25 * 0.82
```

$P(hair_wet|raining) = P(hair_wet \land raining)/P(raining) = 0.25 \times (0.7 + 0.82) / 0.5 = 0.76$
Interpretation of $\alpha_i$

- If not $P(h_i|B)$, then what is $\alpha_i$?

- A simple and natural interpretation:
  - Conditional probability of $h_i$ given that $B$ is true, *in the absence of other causes of $h_i$*
  - This is in fact a very **natural** and **local** kind of knowledge
    - **natural**: corresponds to our notion of a conditional probability
    - **local**: can be defined for one rule without knowing the context of that rule (the other rules in the LPAD)
      - An essential property! $P(h_i|B)$ is not a property of the rule itself, but of the LPAD, whereas $\alpha_i$ is a property of the rule itself
  - From the point of view of knowledge representation, the $\alpha_i$ are a more basic concept than “conditional probabilities” in general
    - Easier to interpret
    - Easier to express
  - But... harder to learn! $P(h_i|B)$ can be estimated directly from data, $\alpha_i$ cannot
Learning LPADs

Riguzzi, ILP 2004:
- Learns structure and parameters of LPADs
- Limitation imposed on these LPADs
  - When $h_i$ occurs in the head of more than one clause, the bodies of these clauses must be mutually exclusive
  - Let’s call such LPADs ME-LPADs (ME for mutual exclusion)
  - $\alpha_i$ are then estimated as $P(h_i|B)$
    - Only correct if $h_i$ can never be caused by more than one rule
    - This is guaranteed by the mutual exclusion rule
- ME-LPADs are a strict subset of all LPADs
- Leaves open the problem of learning a more general class of LPADs
1-LPADs and CP-LPADs

We call an LPAD a **CP-LPAD** iff for each rule $h_1: \alpha_1 \lor h_2: \alpha_2 \lor \ldots \lor h_m: \alpha_m \leftarrow B$, it holds that $\alpha_i = P(h_i|B)$ for all $i$

- All the parameters are *conditional probabilities*
- This means they can easily be estimated from data

**Note:** this is a semantic, not syntactic, constraint

We call an LPAD a **1-LPAD** if for each $h_i: \alpha_i$ in the LPAD, one of the following conditions holds:

- $h_i$ occurs only **once** in the head of a rule
- OR $\alpha_i = 1$

**Note:** this is a syntactic constraint
Properties

- All ME-LPADs are CP-LPADs
  - Indirectly proven by Riguzzi (by proving the correctness of his algorithm)

- All 1-LPADs are CP-LPADs
  - If \( h_i \) occurs in head of only one rule, \( P(h_i|B) = \alpha_i \)
  - If \( h_i \) occurs in multiple rules, \( \alpha_i = 1 \), and since we know \( P(h_i|B) \geq \alpha_i \), we can conclude \( P(h_i|B) = 1 = \alpha_i \)

- Each LPAD can be transformed into a 1-LPAD
  - Hence, the class of 1-LPADs is equivalent to the class of LPADs
Transformation LPAD to 1-LPAD

► For each rule $R_j = h_1: \alpha_1 \lor h_2: \alpha_2 \lor \ldots \lor h_m: \alpha_m \leftarrow B$ with at least one $h_i: \alpha_i$ occurring also in another head and $\alpha_i < 1$

  ▪ (1) Give all symbols in the head an index $j$
    ▪ $R'_j = h_{1j}: \alpha_{1j} \lor h_{2j}: \alpha_{2j} \lor \ldots \lor h_{mj}: \alpha_{mj} \leftarrow B$

  ▪ (2) For each $h_i$ in $R_j$, add a rule $h_i:1 \leftarrow h_{ij}$

► Result is equivalent to original LPAD

► Result is a 1-LPAD (by construction)

Example:

```
hair_wet : 0.7 <- raining.
hair_wet : 0.4 <- gone_swimming.
raining : 0.5.
gone_swimming : 0.5.
```

```
hair_wet1 : 0.7 <- raining.
hair_wet <- hair_wet1.
hair_wet2 : 0.4 <- gone_swimming.
hair_wet <- hair_wet2.
raining : 0.5.
gone_swimming : 0.5.
```
Propositional acyclic 1-LPAD = bayesian network with hidden nodes

Parameters of such networks can be learned (EM, gradient descent, noisy-or learning...)

\[
\begin{align*}
hair\_wet : 0.7 & \leftarrow \text{raining.} \\
hair\_wet : 0.4 & \leftarrow \text{gone\_swimming.} \\
\text{raining} : 0.5. \\
\text{gone\_swimming} : 0.5.
\end{align*}
\]
Conclusion 1

- Propositional, acyclic LPADs can be turned into a Bayesian net with hidden variables
- Hence, these can be learned with standard BN learning techniques
  - EM, gradient descent, ...
  - Methods specifically for learning noisy-or (e.g., Vomlel, 2004)
- However:
  - EM, gradient descent, ... are general techniques, may get stuck in local optima, ...
  - If the transformation to BN gives us nothing more than the ability to use a general technique, why not use EM or gradient descent to learn LPADs directly?
    - possibly including cyclic LPADs
Learning LPADs directly

- A gradient descent approach to learning LPAD parameters directly from a distribution over interpretations $D$:
  - Randomly assign values to parameters of LPAD
  - Repeat:
    - Compute probabilities of interpretations $D'$ according to current parameters
    - Adapt parameters to reduce error $D'-D$ (gradient descent)
  - Until $D'-D$ is not reduced any further

- Works also with cyclic, recursive, first-order LPADs
  - Although not yet clear how to handle infinite models
Our implementation

► Does not learn from data, but from distribution
► Input = set of logic formulae $\phi_j$ with their probability $\Pr(\phi_j) = p_j$
  - Distribution can be fully specified (probabilities for all interpretations), or underspecified (e.g., only marginal probabilities of atoms)
► Symbolic computation of error and partial derivatives in terms of $\alpha_i$
  - Given LPAD with parameters $\alpha_i$, compute:
    - For each $\phi_i$, formula $f_j$ expressing $\Pr(\phi_i)$ in terms of $\alpha_i$
    - Formula expressing error $E$ in terms of $f_j$ ($E = \Sigma_i (f_j - p_j)^2$)
    - Formulas expressing all partial derivatives $\partial E/\partial \alpha_i$ in terms of $\alpha_i$
  - During gradient descent, just fill in the numbers in the formulae
    - Avoids having to recompute models and their probabilities all the time
► Implemented approach performs bottom-up computation of models
  - Can handle first order clauses, recursion, etc., but not infinite models
  - But still buggy (currently handles only propositional LPADs correctly)
LPAD structure

hair_wet : α <- raining.
hair_wet : β <- gone_swimming.
raining: γ.
gone_swimming: δ.

Constraints (the “data” to learn from)

\[
\begin{align*}
\{ & \Pr(\text{raining})=0.5, \\
& \Pr(\text{gone_swimming})=0.3, \\
& \Pr(\text{hair_wet})=0.2, \\
& \Pr(\text{hair_wet} \land \text{swimming}) = 0.1 \}
\end{align*}
\]

Minimize: \[(γ - 0.5)^2 + (δ - 0.3)^2 + (γ(1-δ)α+(1-γ)δβ+γδ(1-(1-α)(1-β)) - 0.2)^2 + ((1-γ)δβ + γδ(1-(1-α)(1-β)) - 0.1)^2\]
Example run

   Target = [c([a], 0.1), c([b], 0.05), c([a,b], 0.05)],
   learn_param(Prog, Target, Prog2).

LPAD structure:
[a:x1]:-[b]
[b:x2]:-[a]
[a:x3]:-[]
[b:x4]:-[]

Constraints:
p([a])=0.1  p([b])=0.05  p([a,b])=0.05

Target formula and derivatives:
f = (-1*(x1*(x3*x4))+(1*x3+(1*(x1*x4)+0))-0.1)**2+((-1*(x2*(x3*x4))+(1*x4+(1*(x2*x3)+0))-0.05)**2+0)
d(f)/d(x4) = -0.3*x1+(4*(x1*(x1*x4))+(2*(x1*(x2*x3))+(4*(x1*(x3*x4))+(-4*(x1*(x2*(x3*x4)))+(-0.1+(2*x4+(-4*(x2*(x3*x4))+(-2*(x1*(x3*x3))+0))))))))
d(f)/d(x3) = ...
d(f)/d(x2) = ...
d(f)/d(x1) = ...

Result: ok(9.77269318949174e-08,20)
[a:0.682734399120201]:-[b]
[b:0.492464216818767]:-[a]
[a:0.0990729787703779]:-[]
[b:0.00153210778243481]:-[]
Simple procedure to learn structure and parameters of LPADs:

- Define an LPAD refinement operator
- Starting with a simple LPAD $= \{\}$, repeat:
  - Refine LPAD
  - Compute optimal parameters of LPAD using gradient descent
- Until LPAD cannot be improved further
Structure learning: Example

s :-
    Constr = [c([a],0.1), c([b],0.5), c([a,b], 0.07)],
    search([], Constr, [a,b,c], LPAD).

?- s.
Best: 0.07017201
[b:0.7351]:-[]

Best: 0.00366808393641315
[b:0.55888600963091]:-[]
[a:0.0999124994023969]:-[]

Best: 0.000321586392016593
[a:0.0,b:0.502147294669357]:-[]
[a:0.109797916609453]:-[]

Best: 8.77082178059683e-08
[a:0.0303963930279203,b:0.500095528914561]:-[]
[a:0.139583058666119]:-[b]

?-

Pr(a)=0.1
Pr(b)=0.5
Pr(a \lor b)=0.07

a:0.03, b:0.5 <-
a:0.14 <- b
Concluding remarks

- LPADs are a powerful representation formalism for probabilistic logical learning
- We do not want to be limited to subclasses of LPADs

LPAD parameter learning:
- Transformation to BN is possible for acyclic LPADs
  - Straightforward way to learn them
- Gradient descent method for LPADs with finite models proposed

LPAD structure learning:
- A matter of defining a refinement operator for LPADs
Concluding remarks

► Open questions:
  ▪ LPADs with infinite models?
  ▪ Practical feasibility of learning complex LPADs?
  ▪ Time complexity of learning?
  ▪ “Sample” complexity? (or, how detailed should the input distribution be?)
  ▪ It seems problems tend to be underconstrained.