Probabilistic Logic Programming Learning and Applications

Elena Bellodi

Department of Engineering
University of Ferrara, Italy

University of Udine, PhD Course, December 4th-5th 2018

Outline

1. Learning Problems in Probabilistic Logic Programming
2. Parameter Learning
   - EMBLEM
   - LFI-ProbLog
   - LeProbLog
3. Structure Learning
4. Applications
5. References

Reasoning Tasks

- **Inference**: we want to compute the probability of a query given the model and, possibly, some evidence
- Weight learning: we know the structural part of the model (the logic formulas) but not the numeric part (the weights) and we want to infer the weights from data
- Structure learning: we want to infer both the structure and the weights of the model from data

Weight Learning

- Given
  - model: a probabilistic logic model with unknown parameters
  - data: a set of interpretations
- Find the values of the parameters that maximize the probability of the data given the model
- **Discriminative learning**: maximize the conditional probability of a set of outputs (e.g. ground instances for a predicate) given a set of inputs
- **Generative learning**: tries to be equally capable of predicting the truth value of all predicates

Structure Learning

- Given
  - language bias: a specification of the search space
  - data: a set of interpretations
- Find the formulas and the parameters that maximize the likelihood of the data given the model
- Discriminative or generative learning

Parameter Learning for PLP

- Based on *Expectation - Maximization (EM)* (Dempster et al. (1977))
  - Thon et al. (2008) proposed an adaptation of EM for CPT-L, a simplified version of LPADs. The algorithm computes the counts efficiently by repeatedly traversing the BDDs representing the explanations
  - Ishihata et al. (2008) independently proposed a similar algorithm
  - **EMBLEM** (Bellodi and Riguzzi (2013)) adapts Ishihata et al. (2008) to LPADs
  - **LFI-ProbLog** (Fierens et al. (2015)) for the ProbLog language
- Based on *gradient descent*: **LeProbLog** (Gutmann et al. (2008)) for the ProbLog language

EM-based Learning

EMBLEM: *EM over Bdds for probabilistic Logic programs Efficient Mining*

Definition (EMBLEM Learning Problem)

Given an LPAD $\mathcal{P}$ with unknown parameters and two sets $E^+ = \{e_1, \ldots, e_T\}$ and $E^- = \{e_{T+1}, \ldots, e_Q\}$ of ground atoms (positive and negative examples), find the value of the parameters Π of $\mathcal{P}$ that maximize the likelihood of the examples, i.e., solve

$$argmax_\Pi P(E^+, \sim E^-) = argmax_\Pi \prod_{t=1}^{T} P(e_t) \prod_{t=T+1}^{Q} P(\sim e_t).$$

Predicates for the atoms in E^+ and E^-: are called target because the objective is to be able to better predict the truth value of atoms for them.

EM-based Learning

EMBLEM: *EM over Bdds for probabilistic Logic programs Efficient Mining*

- Typically, the LPAD has two components:
  - a set of rules, annotated with parameters
  - a set of certain ground facts, representing background knowledge on individual cases of a specific world
- Useful to provide information on more than one world: a background knowledge and sets of positive and negative examples for each world
- Description of one world: *mega-interpretation* or *mega-example*
- **Positive examples**: ground facts that are *true* in the mega-interpretation
- **Negative examples** : ground facts that are declared *false* ($neg(a)$ for negative example a)

EM-based Learning

EMBLEM: *EM over Bdds for probabilistic Logic programs Efficient Mining*

- Predicates can be treated as closed-world or open-world
- In case of CWA: the body of clauses with that predicate in the head is resolved only with facts in the interpretations
- In case of OWA: the body of clauses with that predicate in the head is resolved both with facts in the interpretations and with clauses in the theory. If this option is set and the theory is cyclic, EMBLEM may use a *depth bound* on SLD derivations to avoid going into infinite loops

EM-based Learning

EMBLEM: *EM over Bdds for probabilistic Logic programs Efficient Mining*

- Problem: given a set of interpretations and a program, **find the parameters maximizing the likelihood** of the interpretations

Why do we need EM?

- The interpretations record the truth value of ground atoms, not of the (latent) random variables
- Unseen (or "missing") data: the number of times that i-th head atom has been selected from groundings of the clauses used in the proof of the queries
- An iterative method for problems with incomplete data is needed

EM-based Learning

EMBLEM: *EM over Bdds for probabilistic Logic programs Efficient Mining*

Expectation-Maximization (EM) algorithm:

- **Expectation step**: the distribution of the unseen variables is computed given the observed data and the current estimate of parameters
- **Maximization step**: new parameters are computed from the estimates of the E step using relative frequency
- Ends when likelihood does not improve anymore

EMBLEM

EM over Bdds for probabilistic Logic programs Efficient Mining

- EMBLEM is based on **Knowledge Compilation to BDDs**
- EMBLEM generates a BDD for each example in $E = \{e_1, \ldots, e_T, \sim e_{T+1}, \ldots, \sim e_Q\}$.
- Each example is a query, and the BDD encodes its explanations
- Then it enters the EM cycle, in which the steps of Expectation and Maximization are repeated until the log-likelihood of the examples reaches a local maximum
- Expectations computed with two passes directly over the BDDs $\rightarrow$ efficient
- BDDs *have to be compiled only once*, before the EM cycle, so can be reused in all further iterations

EMBLEM

EM over Bdds for probabilistic Logic programs Efficient Mining

- To use binary random variables: CUDD library `https://www.cs.rice.edu/~lm30/RSynth/CUDD/cudd/doc/`
- Encoding of multi-valued random variable with Boolean random variables

$$\pi_{i1} = P(X_{ij1}) = P(X_{ij} = 1)$$

$$\ldots$$

$$\pi_{ik} = P(X_{ijk}) = \frac{P(X_{ij} = k)}{\prod_{l=1}^{k-1}(1 - P(X_{ijl}))}$$

EMBLEM

EM over Bdds for probabilistic Logic programs Efficient Mining

- **Expectation**: compute $\mathbf{E}[c_{ik0}|e]$ and $\mathbf{E}[c_{ik1}|e]$ for all examples e, rules C_i and $k = 1, \ldots, n_i - 1$, where c_{ikx} is the number of times a Boolean variable X_{ijk} takes value x for $x \in \{0, 1\}$, with $j \in g(i) = \{j|\theta_j$ is a substitution grounding $C_i\}$.

$$\mathbf{E}[c_{ikx}|e] = \sum_{j \in g(i)} P(X_{ijk} = x|e)$$

- **Maximization**: update parameters π_{ik} representing $P(X_{ijk} = 1)$ for all rules C_i and $k = 1, \ldots, n_i - 1$

$$\pi_{ik} = \frac{\sum_{e \in E} \mathbf{E}[c_{ik1}|e]}{\sum_{e \in E} \mathbf{E}[c_{ik1}|e] + \mathbf{E}[c_{ik0}|e]}$$

EMBLEM

EM over Bdds for probabilistic Logic programs Efficient Mining

- $P(X_{ijk} = x|e)$ is given by $\frac{P(X_{ijk}=x,e)}{P(e)}$
- $P(X_{ijk} = x, e)$: probability of the paths passing through the x-child of a node n associated with variable X_{ijk} so

$$P(X_{ijk} = x, e) = \sum_{n \in N(X_{ijk})} \pi_{ikx} F(n) B(child_x(n)) = \sum_{n \in N(X_{ijk})} e^x(n)$$

- $F(n)$ is the *forward probability*, the probability mass of the paths from the root to n (computed with one recursive pass on the BDD)
- $B(n)$ is the *backward probability*, the probability mass of paths from n to the 1-leaf (computed with a second recursive pass on the BDD)
- $P(e) = B(root)$: the sum of the probabilities of all the paths in the BDD from the root to a 1-leaf, where the probability of a path is defined as the product of the probabilities of the individual choices along the path

EMBLEM

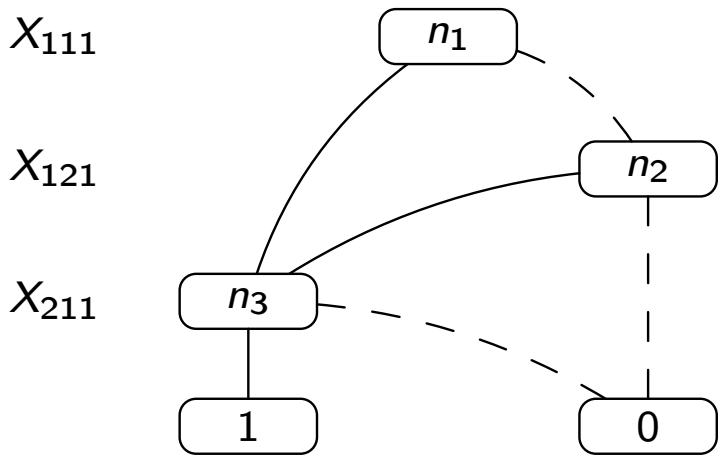
EM over Bdds for probabilistic Logic programs Efficient Mining

1: **function** EMBLEM($E, \mathcal{P}, \epsilon, \delta$)
2: build *BDDs* for examples in E
3: $LL = -inf$
4: **repeat**
5: $LL_0 = LL$
6: $LL =$ Expectation($BDDs$)
7: Maximization
8: **until** $LL - LL_0 < \epsilon \vee LL - LL_0 < -LL \cdot \delta$
9: return LL, π_{ik} for all i, k
10: **end function**

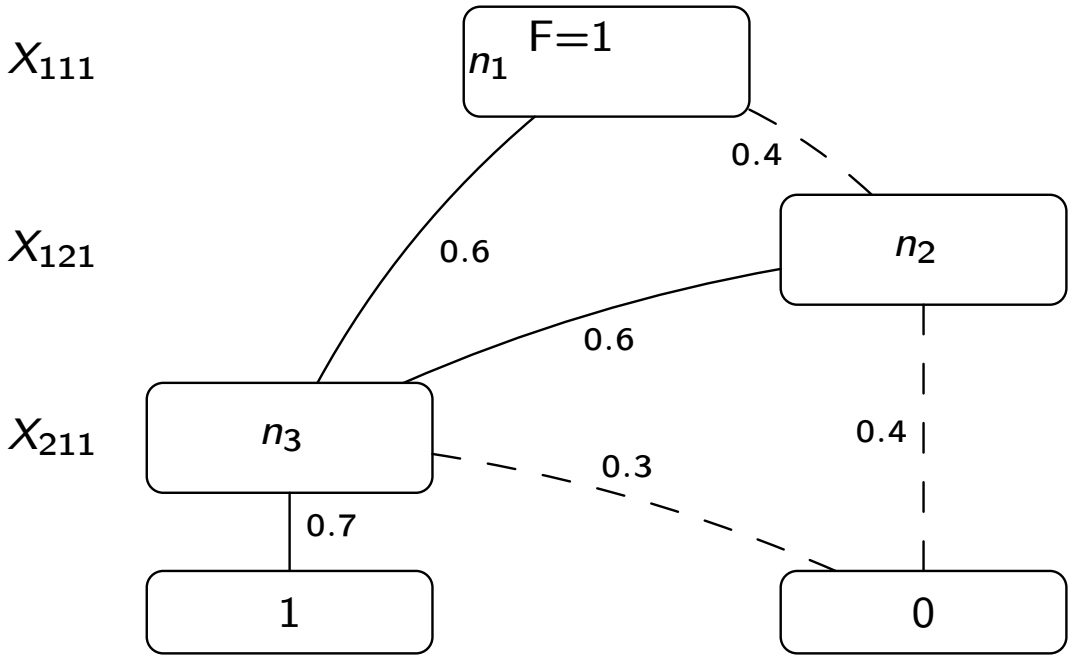
EMBLEM Example

$C_1 = epidemic : 0.6; pandemic : 0.3 : -flu(X), cold.$
$C_2 = cold : 0.7.$
$C_3 = flu(david).$
$C_4 = flu(robert).$

- Clause C_1: 2 groundings $(X/david, X/robert)$. R.v.: X_{111} and X_{112} for the 1st grounding, X_{121} and X_{122} for the 2nd grounding
- C_2: single grounding, random variable X_{211}
- BDD for $e = epidemic$:

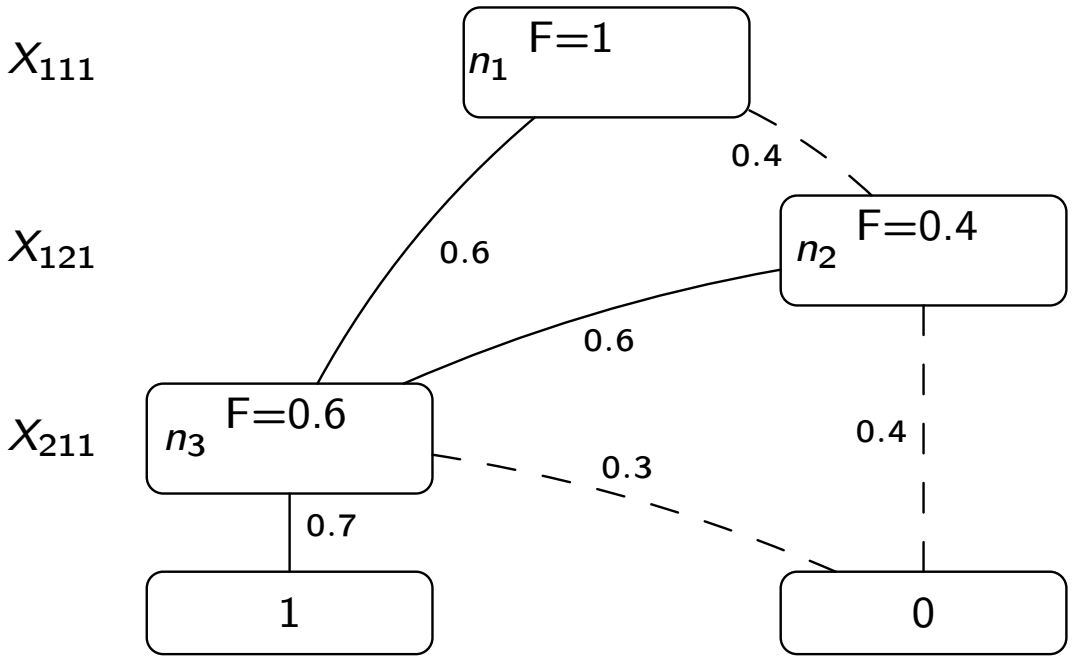


EMBLEM Example



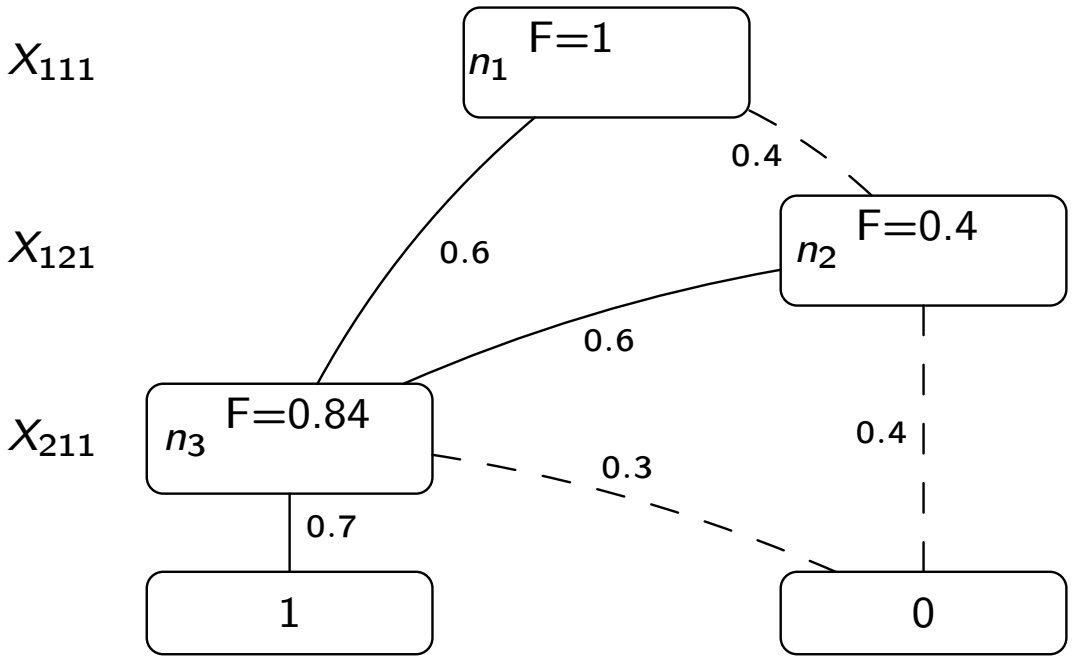
Start from the first level (root): $F(n_1) = 1$

EMBLEM Example



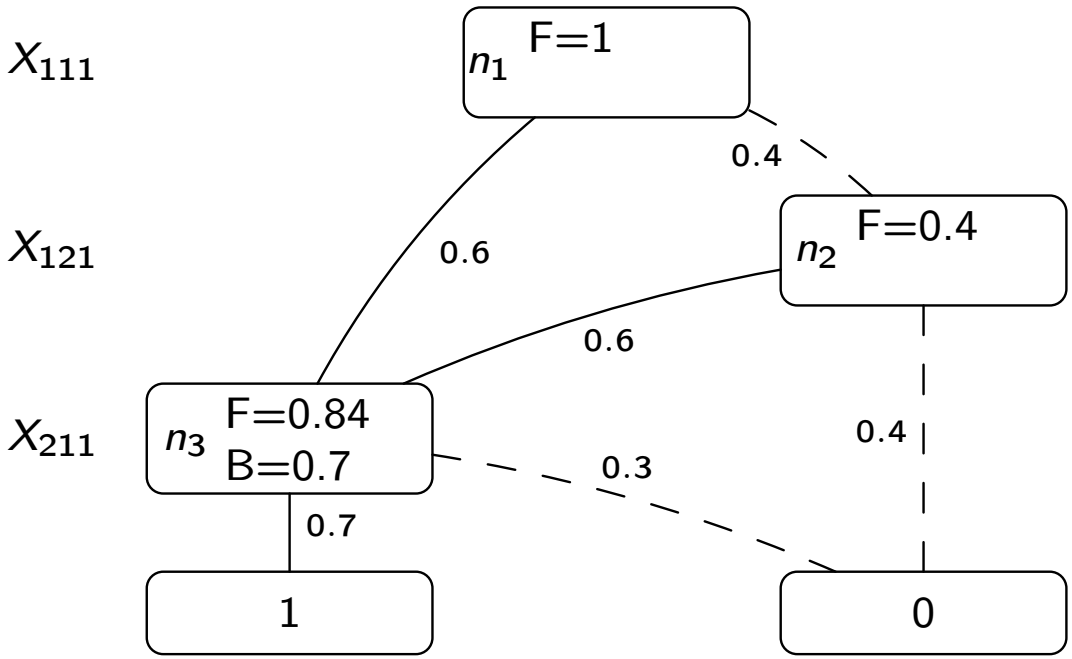
Compute F for the 0- and 1-child:
$F(n_2) = 1 * 0.4 = 0.4$; $F(n_3) = 1 * 0.6 = 0.6$

EMBLEM Example



Move to the second level (n_2) and compute F for the 0- and 1-child:
0-leaf is skipped; $F(n_3) = 0.6 + 0.4 * 0.6 = 0.84$
Move to the third level (n_3): children of n_3 are leaves so nothing is done

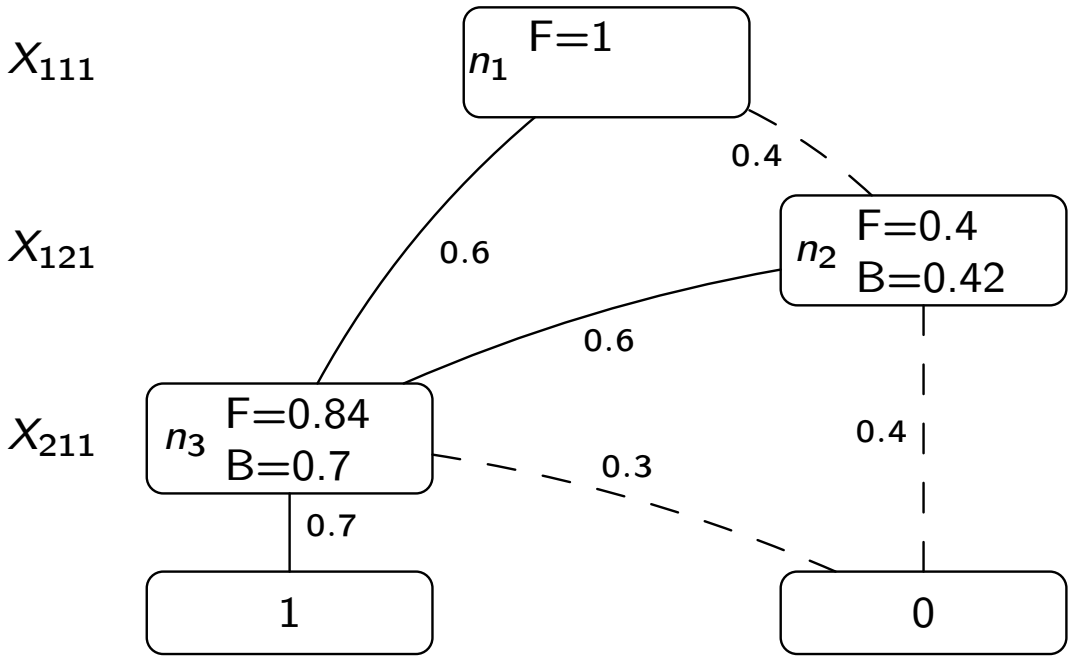
EMBLEM Example



Start from the leaves and go up one level at a time recursively

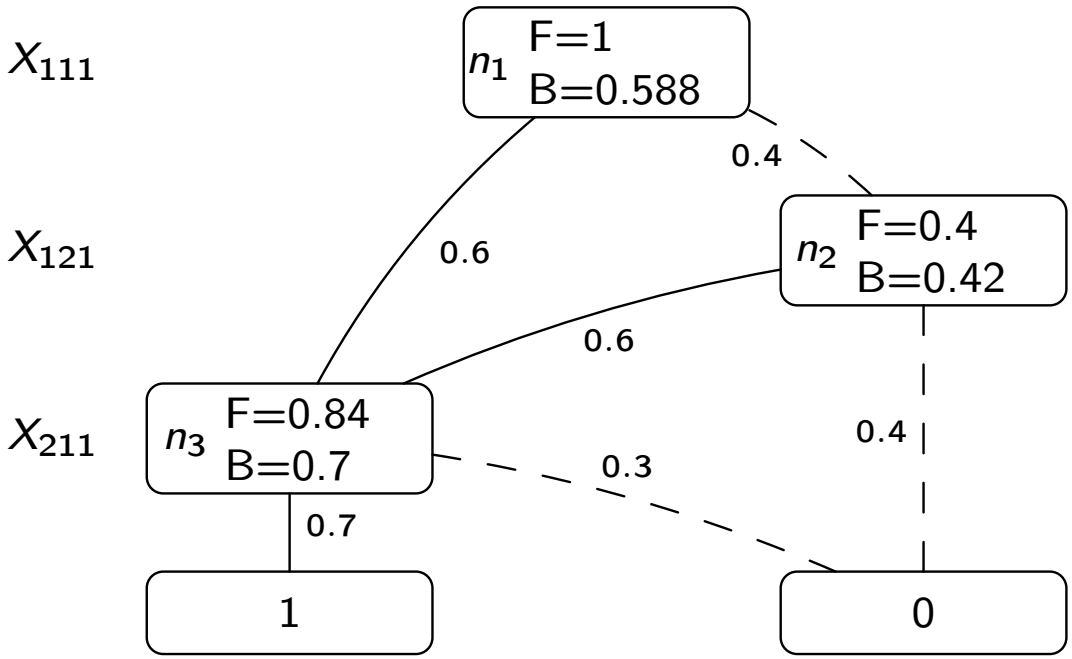
$B(n_3) = 1 * 0.7 * 0 * 0.3 = 0.7$

EMBLEM Example



$$B(n_2) = 0.7 * 0.6 + 0 * 0.4 = 0.42$$

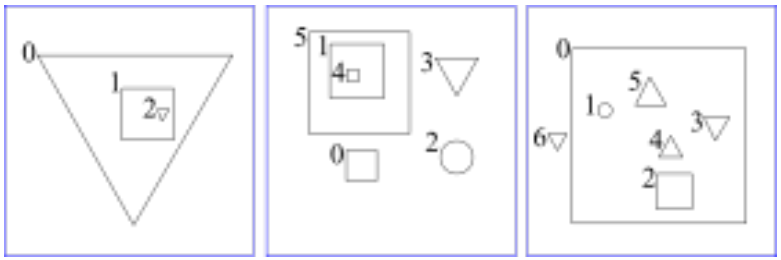
EMBLEM Example



$$B(n_1) = 0.7 * 0.6 + 0.4 * 0.42 = 0.588 = P(e)$$

Example: Bongard Problems

- Introduced by the Russian scientist M. M. Bongard
- Two sets of relatively simple diagrams, say A and B, are given
- All the diagrams from set A (positive ex) have *a common factor or relationship or attribute*, which is lacking in all the diagrams of set B (negative ex)
- Problem: discriminate the two



Example: Bongard Problems

Data

Each mega-example encodes a single picture.

begin(*model*(2)).	*begin*(*model*(3)).
pos.	**neg(pos).**
triangle(*o*5).	*circle*(*o*4).
config(*o*5, *up*).	*circle*(*o*3).
square(*o*4).	*in*(*o*3, *o*4).
in(*o*4, *o*5).	*square*(*o*2).
circle(*o*3).	*circle*(*o*1).
triangle(*o*2).	*in*(*o*1, *o*2).
config(*o*2, *up*).	*end*(*model*(3)).
in(*o*2, *o*3).	
triangle(*o*1).	
config(*o*1, *up*).	
end(*model*(2)).	

Example: Bongard Problems

LPAD

$pos : 0.5 : - \quad circle(A), in(B, A).$
$pos : 0.5 : - \quad circle(A), triangle(B).$

The task is to tune the two parameters.

Hands-on

- http://cplint.eu/e/bongard.pl
- Learn the parameters and test the result with:

```
induce_par([train],P),test(P,[test],LL,AUCROC,ROC,AUCPR,PR).
```

- Expected result

```
AUCPR = 0.674,
AUCROC = 0.800,
LL = -219.099,
P =
pos:0.0841358 :-  circle(A),in(B,A).
pos:0.412669  :-  circle(A),triangle(B).
```

EM-based Learning

LFI-ProbLog: *Learning From Interpretations-ProbLog*

- Full or partial observability of the domain
- LFI-ProbLog learns the parameters of ProbLog programs from both full and partial interpretations (Fierens et al. (2015))
- Partial interpretations specify the truth value of *some but not necessarily all* ground atoms
- Partial interpretation $I = (I_T, I_F)$: the atoms in I_T are true and those in I_F are false

EM-based Learning

LFI-ProbLog: *Learning From Interpretations-ProbLog*

- The probability of a partial interpretation is the sum of the probabilities of all possible worlds consistent with the known atoms
- The known atoms can be represented by the **conjunction**: $q(I) = \bigwedge_{a \in I_T} \wedge \bigwedge_{a \in I_F} \sim a$

Definition (LFI-ProbLog learning problem)

Given a ProbLog program $\mathcal{P}$ with unknown parameters and a set $E = \{I_1, \ldots, I_T\}$ of partial interpretations (the examples), find the value of the parameters Π of $\mathcal{P}$ that maximize the likelihood of the examples, i.e., solve

$$argmax_{\Pi} P(E) = argmax_{\Pi} \prod_{t=1}^{T} P(q(I_t))$$

EM-based Learning

LFI-ProbLog: *Learning From Interpretations-ProbLog*

$p1 :: burglary.$
$p2 :: earthquake.$
$p3 :: hears\_alarm(X) \leftarrow person(X).$
$alarm \leftarrow burglary.$
$alarm \leftarrow earthquake.$
$calls(X) \leftarrow alarm, hears\_alarm(X).$
$person(mary).$
$person(john).$

$I_{T1} = \{alarm\}, I_{T2} = \{earthquake, calls(mary)\}, I_{T3} = \{calls(john)\}$
We may not know whether a burglary occurred.

EM-based Learning

LFI-ProbLog: *Learning From Interpretations-ProbLog*

- Partially observed data $\rightarrow$ EM algorithm
- A d-DNNF circuit is built for each partial interpretation I_t with the evidence $q(I)$
- A Boolean random variable X_{ij} for each ground probabilistic fact $f_i\theta_j$
- E-step:

$$\mathbf{E}[c_{ix}] = \sum_{t=1}^{T} \mathbf{E}[c_{ix}|I_t]$$

where

$$\mathbf{E}[c_{ix}|I_t] = \sum_{j \in g(i)} P(X_{ij} = x|I_t)$$

EM-based Learning

LFI-ProbLog: *Learning From Interpretations-ProbLog*

- $\mathbf{E}[c_{ix}|I]$ is the expected value given example I of the number of times variable X_{ij} takes value x for any j in $g(i)$, the set of grounding substitutions of f_i
- $\mathbf{E}[c_{ix}]$ is the expected value given all the examples
- M-step: parameters π_i of probabilistic facts f_i can be computed as

$$\pi_i = \frac{\mathbf{E}[c_{i1}]}{\mathbf{E}[c_{i0}] + \mathbf{E}[c_{i1}]}$$

- The algorithm keeps on updating the parameters until the log likelihood of the interpretations is maximal

EM-based Learning

LFI-ProbLog: *Learning From Interpretations-ProbLog*

- $P(X_{ij} = x|I) = \frac{P(X_{ij}=x, I_t)}{P(q(I))}$
- $P(q(I)) = P(root)$: computed bottom-up on the d-DNNF
- $P(X_{ij} = x, I)$: computed top-down for all variables X_{ij} and values x on the d-DNNF
- As, BDDs, d-DNNFs *have to be compiled only once*, before the EM cycle, so can be reused in all further iterations
- A previous version (Gutmann et al. (2011)) used BDDs instead of d-DNNFs, but this algorithm scales better

Gradient Descent-based Learning

LeProbLog

- Parameter learning system that *starts from a set of examples annotated with a probability*
- It derives from **probabilistic databases**, i.e., generalizations of traditional relational databases that can deal with uncertainty
- "Dirty" databases arise when integrating data from various sources, when analyzing social, biological, and chemical networks
- Probabilistic databases associate probabilities to facts, indicating the probabilities with which these facts hold
- This information is then used to compute the success probability of queries

Gradient Descent-based Learning

LeProbLog: Example

- A graph with uncertain edges can be expressed in ProbLog as:

$$0.8 : edge(a, c). \quad 0.7 : edge(a, b). \quad 0.8 : edge(c, e).$$
$$0.6 : edge(b, c). \quad 0.9 : edge(c, d). \quad 0.5 : edge(e, d).$$

- Aim: finding the value of the parameters Π of a ProbLog program so that the probability assigned by the program to the examples is as close as possible to the one given

Gradient Descent-based Learning

LeProbLog

Definition (LeProbLog learning problem)

Given a ProbLog program $\mathcal{P}$ and a set of training examples $E = \{(e_1, p_1), \ldots, (e_T, p_T)\}$ where e_t is a ground atom and $p_t \in [0, 1]$ for $t = 1, \ldots, T$, find the parameters of the program so that the mean squared error

$$MSE = \frac{1}{T} \sum_{t=1}^{T} (P(e_t) - p_t)^2$$

is minimized.

Gradient Descent-based Learning

LeProbLog

- Gradient descent is a standard way of minimizing a given error function
- LeProbLog iteratively updates the parameters in the opposite direction of the gradient of the error

$$\frac{\partial MSE}{\partial \Pi_j} = \frac{2}{T} \sum_{t=1}^{T} (P(e_t) - p_t) \cdot \frac{\partial P(e_t)}{\partial \Pi_j}$$

- Π_j are the parameters of the ProbLog program
- LeProbLog compiles queries to BDDs, therefore $P(e_t)$ can be computed with Function PROB
- It uses the k-best explanations for each example e_t

Gradient Descent-based Learning

LeProbLog

- $\frac{\partial P(e_t)}{\partial \Pi_j}$ is computed by a dynamic programming algorithm that traverses the BDD bottom up, as

$$\frac{\partial P(e_t)}{\partial \Pi_j} = \frac{\partial P(f(\mathbf{X}))}{\partial \Pi_j}$$

  where $f(\mathbf{X})$ is the Boolean function represented by the BDD

- When performing gradient descent, the parameters must remain in the [0,1] interval
- **Reparameterization** with the sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$ that takes a real value $x \in (-\infty, +\infty)$ and returns a real value in $(0, 1)$
- When using the k-best explanations, the set of k best proofs may change due to parameter updates $\rightarrow$ recompute the set of proofs and the corresponding BDD

Structure Learning for PLP

- For LPADs: SLIPCOVER (*Structure LearnIng of Probabilistic logic program by searching OVER the clause space* (Bellodi and Riguzzi (2015)))
- For ProbLog: ProbFOIL and ProbFOIL+ (De Raedt and Thon (2011),De Raedt et al. (2015)): learn rules from probabilistic examples

SLIPCOVER

- Two-phase algorithm
  1. Beam search in the space of clauses to find the promising ones
  2. Greedy search in the space of probabilistic programs guided by the LL of the data
- *Discriminative learner*: the search for clauses is directly guided by the goal of maximizing the predictive accuracy of the resulting theory on the target predicates
- **Input**: a set of mega-examples containing positive and negative examples for all predicates that may appear in the head of clauses, either target or non-target (*background* predicates); an indication of which predicates are target, i.e., those for which we want to optimize the predictions of the final theory

SLIPCOVER

- Beam search starts with a set of beams
- The initial set of beams is generated by building a set of bottom clauses as in Progol (Muggleton (1995))
- This requires a user-defined language bias, based on *mode* declarations

Language Bias

- Syntax

```
modeh(RecallNumber,PredicateMode).    [for head]
modeb(RecallNumber,PredicateMode).    [for body]
```

- `RecallNumber` can be a number or *. Usually *. Maximum number of answers to queries to include in the bottom clause
- `PredicateMode`: template

```
p(ModeType1,ModeType2,...)
```

Language Bias

- `p` is a predicate of the domain
- `ModeType` can be:
  - Simple:
    - `+T` input variables of type `T`;
    - `-T` output variables of type `T`; or
    - `#T`, `-#T` constants of type `T`. (They behave differently during the generation of bottom clauses)
  - Structured: of the form `f(..)` where `f` is a function symbol and every argument can be either simple or structured.

Language Bias

- Examples:

```
modeb(1,mem(+number,+list)).
modeb(2,mem(+number,[+number|+list])).
modeb(5,(+integer)=(#integer)).
modeb(*,advisedby(-stud,+prof)).
```

Bottom Clause $\bot$

- Most specific clause covering an example e: $e \leftarrow B$
- B: set of ground literals that are true for example e
- B obtained by considering the **constants in e and querying the predicates of the background for true atoms** regarding these constants
- A map from types to lists of constants is kept is enlarged with constants in the answers to the queries
- Values for output arguments (`-T`) are used as input arguments (`+T`) for other predicates
- `-#T` constants can be used as input constant arguments for other predicates, `#T` can't

Bottom Clause $\perp$

Procedure

1. Initialize a map m from types to lists of values to $\emptyset$
2. Pick a $modeh(r, s)$ and an example e matching s
3. Add to m(T) the values of +T arguments in e
4. For $i = 1$ to d
   - For each $modeb(r, s)$ select ground facts matching s and the values of +T arguments found in $m(T)$

You'll get a clause $e : -b_1, ..., b_m$, typically with m large
SLIPCOVER allows to use more than one example e for each $modeh(r, s)$

Bottom Clause $\bot$: Example

```
modeh(*,father(+person,+person)).
modeb(*,parent(+person,-person)).
modeb(*,parent(-#person,+person)).
modeb(*,male(+person)).
modeb(*,female(#person)).

e = father(john,mary)
KB={parent(john,mary),parent(david,steve),parent(kathy,mary),
    female(kathy),male(john),male(david)}
```

$\bot = e \leftarrow B$:

```
father(john,mary) ← parent(john,mary),
      parent(kathy,mary),male(john),female(kathy).
```

Bottom Clause $\bot$

- The resulting ground clause $\bot$ is then processed by **replacing each term in a + or - placemarker with a variable**, using the same variable for identical terms
- A constant (#T or -#T) is not replaced by a variable

$e \leftarrow B$:

```
father(X,Y) ← parent(X,Y),male(X),
              parent(kathy,Y),female(kathy).
```

Language Bias

- To generate clauses with more than two head atoms:

$$modeh(r, [s_1, \ldots, s_n], [a_1, \ldots, a_n], [P_1/Ar_1, \ldots, P_k/Ar_k])$$

- $s_1, \ldots, s_n$ are schemas
- $a_1, \ldots, a_n$ are atoms such that a_i is obtained from s_i by replacing placemarkers with variables
- $a_1, \ldots, a_n$ are used to indicate which variables should be shared by the atoms in the head
- P_i/Ar_i are the predicates admitted in the body

Language Bias

- The generation of a bottom clause is the same except for the fact that the goal to call is composed of more than one atom
- Goal $a_1, \ldots, a_n$ is called and r answers that ground all a_is are kept
- Resulting bottom clause: $a_1 \ ; \ \ldots \ ; \ a_n :- b_1, \ldots, b_m$
- Ex.:

```
modeh(*,[advisedby(+person,+person),tempadvisedby(+person,
+person)], [advisedby(A,B),tempadvisedby(A,B)],
[professor/1,student/1,hasposition/2,inphase/2,
publication/2,taughtby/3,ta/3,courselevel/2,yearsinprogram/2]).
```

SLIPCOVER

Beam search in the space of clauses

- Consider the bottom clauses built as
  $e :- b_1, ..., b_m$ or $a_1 \ ; \ \ldots \ ; \ a_n :- \ b_1, \ldots, b_m$
- For each predicate that appears in a modeh declaration (either target or background) a beam is created as a set $\{(Cl, Literals)\}$ where:
- $Cl = e : 0.5 :- true.$ (disj. clause with 2 heads), or
- $Cl = a_1 : \frac{1}{n+1} \ ; \ \ldots \ ; \ a_n : \frac{1}{n+1} :- true.$ (disj. clause with >2 heads)
- $Literals = \{b_1, \ldots, b_m\}$

SLIPCOVER

Beam search in the space of clauses

- Cycle over each beam to refine the bottom clauses, for a predefined number of iterations or until the beams are not empty
- For each clause Cl, **refinements** are computed by **adding** a literal from $b_1, \ldots, b_m$ to its body or **deleting** an atom from the head in the case of a multiple-head clause
- Refinements must respect the input-output modes of the bias declarations
- Refinements must be connected (i.e., each body literal must share a variable with the head or a previous body literal)
- $(Cl', Literals')$: Cl' is Cl with a new body or head, $Literals'$ is $Literals$ with a body literal removed

SLIPCOVER

Beam search in the space of clauses

- EMBLEM is executed on the LPAD containing Cl' to optimize its parameters $\rightarrow Cl''$
- Log-likelihood (LL) is used as the score of Cl''
- **Two lists for the promising clauses**: TC for target predicates and BC for background predicates, with a maximum size
- Cl'' is inserted in TC if a target predicate appears in its head, otherwise in BC, in order of LL

SLIPCOVER

Greedy search in the space of theories

- **Starts with an empty theory and adds a target clause at a time from the list** *TC*
- After each addition, runs EMBLEM and computes the LL of the data as the score of the resulting theory
- If the score is better than the current best, the clause is kept in the theory, otherwise it is discarded

- After the cycle, SLIPCOVER adds all the clauses in *BC* to the theory and performs a final parameter learning on the resulting theory

SLIPCOVER

Execution Example

- UW-CSE dataset: University of Washington Department of Computer Science and Engineering described through 22 different predicates
- **Target predicate:** `advisedby/2`
- Examples of modeh declarations for 2-head clauses

```
modeh(*,advisedby(+person,+person)).
modeh(*,courselevel(+course,#level)).
```

SLIPCOVER

Execution Example

- Examples of modeh declarations for multiple-head clauses (>2)

```
modeh(*,[advisedby(+person,+person),tempadvisedby(+person,+person)],
  [advisedby(A,B),tempadvisedby(A,B)],
  [professor/1,student/1,hasposition/2,inphase/2,publication/2,
  taughtby/3,ta/3,courselevel/2,yearsinprogram/2]).

modeh(*,[student(+person),professor(+person)],
  [student(P),professor(P)],
  [hasposition/2,inphase/2,taughtby/3,ta/3,courselevel/2,
  yearsinprogram/2,advisedby/2,tempadvisedby/2]).

modeh(*,[inphase(+person,pre_quals),inphase(+person,post_quals),
 inphase(+person,post_generals)],
[inphase(P,pre_quals),inphase(P,post_quals),inphase(P,post_generals)],
 [professor/1,student/1,taughtby/3,ta/3,courselevel/2,
 yearsinprogram/2,advisedby/2,tempadvisedby/2,hasposition/2]).
```

SLIPCOVER

Execution Example

- Examples of modeb declarations

```
modeb(*,courselevel(+course, -level)).
modeb(*,courselevel(-course, +level)).
modeb(*,courselevel(+course, #level)).
modeb(*,hasposition(+person, -position)).
modeb(*,hasposition(+person, #position)).
modeb(*,taughtby(+course, -person, -quarter)).
modeb(*,taughtby(-course, +person, -quarter)).
modeb(*,taughtby(+course, +person, -quarter)).
```

SLIPCOVER

Execution Example

1. Example of a two-head bottom clause generated from the *modeh* declaration `modeh(*,advisedby(+person,+person))` and the example $e = advisedby(person155, person101)$

```
advisedby(A,B):0.5 :- professor(B),student(A),hasposition(B,C),
        hasposition(B,faculty),inphase(A,D),inphase(A,pre_quals),
        yearsinprogram(A,E),taughtby(F,B,G),taughtby(F,B,H),
        taughtby(I,B,J),taughtby(I,B,J),taughtby(F,B,G),taughtby(F,B,H),
        ta(I,K,L),ta(F,M,H),ta(F,M,H),ta(I,K,L),ta(N,K,O),ta(N,A,P),
        ta(Q,A,P),ta(R,A,L),ta(S,A,T),ta(U,A,O),ta(U,A,O),ta(S,A,T),
        ta(R,A,L),ta(Q,A,P),ta(N,K,O),ta(N,A,P),ta(I,K,L),ta(F,M,H).
```

SLIPCOVER

Execution Example

2. Example of a multi-head bottom clause generated from the *modeh* declaration:

```
modeh(*,[student(+person),professor(+person)],
[student(P),professor(P)],
[hasposition/2,inphase/2,taughtby/3,ta/3,courselevel/2,
yearsinprogram/2,advisedby/2,tempadvisedby/2]).
```

and the examples $e = student(person218), professor(person218)$

```
student(A):0.33; professor(A):0.33 :- inphase(A,B),
                                     inphase(A,post_generals),
                                     yearsinprogram(A,C).
```

SLIPCOVER

Execution Example

- Example of a refinement from the first bottom clause is

```
advisedby(A,B):0.5 :- professor(B).
```

- EMBLEM is applied to the theory, the parameter is updated obtaining:

```
advisedby(A,B):0.108939 :- professor(B).
```

- The clause is further refined to

```
advisedby(A,B):0.108939 :- professor(B),hasposition(B,C).
```

SLIPCOVER

Execution Example

- Example of a refinement that is generated from the second bottom clause is

  ```
  student(A):0.33; professor(A):0.33 :- inphase(A,B).
  ```

- Updated refinement after EMBLEM

  ```
  student(A):0.5869;professor(A):0.09832 :- inphase(A,B).
  ```

SLIPCOVER

Execution Example

- Suppose that, during the search in the *space of theories* for the target predicate `advisedby`, SLIPCOVER has generated the program:

```
advisedby(A,B):0.1198 :- professor(B),inphase(A,C).
advisedby(A,B):0.1198 :- professor(B),student(A).
```

  with a LL of -350.01.

- After EMBLEM we get:

```
advisedby(A,B):0.05465 :- professor(B),inphase(A,C).
advisedby(A,B):0.06893 :- professor(B),student(A).
```

  with a LL of -318.17.

- Since the LL increased, the last clause is retained and at the next iteration a new clause is added:

```
advisedby(A,B):0.12032 :- hasposition(B,C),inphase(A,D).
advisedby(A,B):0.05465 :- professor(B),inphase(A,C).
advisedby(A,B):0.06893 :- professor(B),student(A).
```

Experiments - Area Under the PR Curve

System	HIV	UW-CSE	Mondial
SLIPCOVER	0.82 ± 0.05	0.11 ± 0.08	0.86 ± 0.07
SLIPCASE	0.78 ± 0.05	0.03 ± 0.01	0.65 ± 0.06
LSM	0.37 ± 0.03	0.07 ± 0.02	-
ALEPH++	-	0.05 ± 0.01	0.87 ± 0.07
RDN-B	0.28 ± 0.06	0.28 ± 0.06	0.77 ± 0.07
MLN-BT	0.29 ± 0.04	0.18 ± 0.07	0.74 ± 0.10
MLN-BC	0.51 ± 0.04	0.06 ± 0.01	0.59 ± 0.09
BUSL	0.38 ± 0.03	0.01 ± 0.01	-

Experiments - Area Under the PR Curve

System	Carcinogenesis	Mutagenesis	Hepatitis
SLIPCOVER	0.60	0.95 ± 0.01	0.80 ± 0.01
SLIPCASE	0.63	0.92 ± 0.08	0.71 ± 0.05
LSM	-	-	0.53 ± 0.04
ALEPH++	0.74	0.95 ± 0.01	-
RDN-B	0.55	0.97 ± 0.03	0.88 ± 0.01
MLN-BT	0.50	0.92 ± 0.09	0.78 ± 0.02
MLN-BC	0.62	0.69 ± 0.20	0.79 ± 0.02
BUSL	-	-	0.51 ± 0.03

Example: Bongard Problems

SLIPCOVER Settings

```
:- use_module(library(slipcover)).
:- if(current_predicate(use_rendering/1)).
:- use_rendering(c3).
:- use_rendering(lpad).
:- endif.
:- sc.                                  //initializes SLIPCOVER
:- set_sc(megaex_bottom,20).
:- set_sc(max_iter,3).                  //no of it. of beam search
:- set_sc(max_iter_structure,10). //no of it. of theory search
:- set_sc(maxdepth_var,4).
:- set_sc(verbosity,1).
```

See `http://cplint.eu/help/help-cplint.html#parameters-1` for the complete setting list.

Example: Bongard Problems

Predicate settings

- Folds
- Target predicates: `output(<predicate>).`
- Input predicates are those whose atoms you are not interested in predicting

  `input_cw(<predicate>/<arity>).`

  True atoms are those in the interpretations and those derivable from them using the background knowledge
- Open world input predicates are declared with

  `input(<predicate>/<arity>).`

Example: Bongard Problems

Predicate settings

```
fold(train,[2,3,5,...]).
fold(test,[490,491,494,...]).
output(pos/0).
input_cw(triangle/1).
input_cw(square/1).
input_cw(circle/1).
input_cw(in/2).
input_cw(config/2).
```

Example: Bongard Problems

Language Bias

- *determination*$(p/n, q/m)$: atoms for q/m can appear in the body of rules for p/n

```
determination(pos/0,triangle/1).
determination(pos/0,square/1).
determination(pos/0,circle/1).
determination(pos/0,in/2).
determination(pos/0,config/2).
modeh(*,pos).
modeb(*,triangle(-obj)).
modeb(*,square(-obj)).
modeb(*,circle(-obj)).
modeb(*,in(+obj,-obj)).
modeb(*,in(-obj,+obj)).
modeb(*,config(+obj,-#dir)).
```

Hands-on

- http://cplint.eu/e/bongard.pl
- Learn both the structure and the parameters and test the result with:

```
induce([train],P),test(P,[test],LL,AUCROC,ROC,AUCPR,PR).
```

- Expected result

```
AUCPR = 0.416,
AUCROC = 0.673,
LL = -104.91,
P =
pos:0.222249 :- triangle(A),config(A,down).
pos:0.124824 :- triangle(A),in(B,A).
pos:0.314871 :- triangle(A).
```

ProbFOIL and ProbFOIL+

- Combine the relational rule learner **FOIL** (first-order inductive learner, Quinlan (1990)) with **ProbLog**
- FOIL learns function-free Horn clauses $C \leftarrow L_1, ..., L_n$, where L_i can be negated
- Given positive and negative examples of some *target* relation, and a set of background-knowledge predicates, FOIL inductively generates rules for the target relation

ProbFOIL and ProbFOIL+

FOIL

Example

- Target relation = *can_reach*
- Examples for the target relation:
  $E+ = \{(0,1),(0,2),(0,3),(1,2),..\}, E- = \{(1,0),(0,7),...\}$
- KB = {link(0,1), link(0,3), ...}
- FOIL learns
  $can\_reach(X,Y) \leftarrow link(X,Y).$
  $can\_reach(X,Y) \leftarrow link(X,Z), can\_reach(Z,Y).$

ProbFOIL and ProbFOIL+

Definition (ProbFOIL/ProbFOIL+ learning problem)

Given

1. a set of training examples $E = \{(e_1, p_1), \ldots, (e_T, p_T)\}$ where each e_i is a ground fact for a target predicate, and $p_i \in [0, 1]$ is the probability of e_i
2. a background theory B containing information about the examples in the form of a ProbLog program
3. a space of possible clauses $\mathcal{L}$

find a hypothesis $H \subseteq \mathcal{L}$ so that the absolute error $AE = \sum_{i=1}^{T} |P(e_i) - p_i|$ is minimized, i.e.,

$$argmin_{H \in \mathcal{L}} \sum_{i=1}^{T} |P(e_i) - p_i|$$

ProbFOIL

- *Probabilistic rule learning problem*: inducing a set of rules that allows one to predict the probability of a target example from its description
- Here the probabilities of the queries are fixed and the structure, that is the rules, are to be learned

ProbFOIL

Example

- The forecast might state that tomorrow the probability of precipitation (pop) is 20%, the wind will be strong enough with probability 70%, and the sun is expected to shine 60% of the time

  $0.2 :: pop(t).$ $0.7 :: windok(t).$
  $0.6 :: sunshine(t).$ $0.7 :: surfing(t).$

- t: identifier of the examples
- Target predicate: $surfing(t)$
- The following rules could be induced

  $surfing(X) : -not\ pop(X), windok(X).$
  $surfing(X) : -not\ pop(X), sunshine(X).$

  and $P(surfing(t))$ can be computed as
  $0.8 * 0.7 + 0.8 * 0.6 * (1 - 0.7) = 0.704$

ProbFOIL

Algorithm 1 The ProbFOIL algorithm

1: $h := t(X_1, \ldots, X_n)$ where t is the target predicate and the X_i distinct variables;
2: $H := \emptyset$; $c := (h \leftarrow b)$; $b := []$;
3: **repeat**
4: $b := []$; initially the body of the rule is empty;
5: **while** $\neg$localstop(H,$h \leftarrow b$) **do** ▷ Grow rule
6: $l := \arg\max_{l \in \rho(h \leftarrow b)}$ localscore$(h \leftarrow [b, l])$
7: $b := [b, l]$
8: let $b = [l_1, \ldots, l_n]$
9: $i := \arg\max_i$ localscore$(H, h \leftarrow l_1, ..., l_i)$;
10: $c := p(X_1, \ldots, X_n) \leftarrow l_1, \ldots, l_i$;
11: **if** globalscore(H) < globalscore$(H \cup \{c\})$ **then**
12: $H := H \cup \{c\}$
13: **until** globalscore(H) > globalscore$(H \cup \{c\})$
14: **return** H

Two nested loops

ProbFOIL

Outer or "Covering" Loop

- Adds clauses (one at each iteration) to the hypothesis H until adding further clauses decreases the quality of the hypothesis
- *Why?* Because adding clauses to the hypothesis for the target predicate is a monotonic operation, that is, it can only increase the probability of an individual example (adding a clause results in extra possibilities for proving that the example is true)
- **Global score(H)**: accuracy of H over the dataset

$$accuracy_H = \frac{TP_H + TN_H}{T}$$

where T is number of examples and TP_H and TN_H are, respectively, the number of *true positives* and of *true negatives*

ProbFOIL

Outer or "Covering" Loop

- *How to compute TP_H and TN_H?*
- Each example e_i is associated with a probability p_i
- It contributes a part p_i to the positive part of the training set and $n_i = 1 - p_i$ to the negative part:
  - $P = \sum_{i=1}^{T} p_i$ and $N = \sum_{i=1}^{T} n_i$
- **Hypothesis H assigns a probability $p_{H,i}$ to each example** e_i (and $n_{H,i} = 1 - p_{H,i}$)
- Perfect prediction: $p_{H,i} = p_i$, $n_{H,i} = n_i$
- If not...

ProbFOIL

Outer or "Covering" Loop

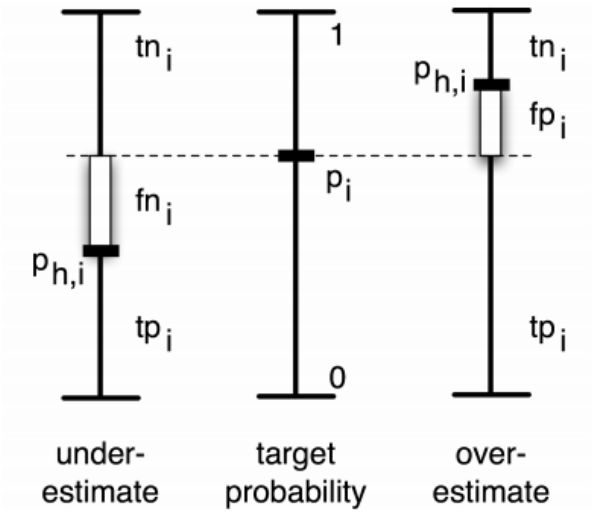
- ...we can build a **contingency table for probabilistic examples**:
  - true positive part $tp_i = min(p_i, p_{H,i})$
  - true negative part $tn_i = min(n_i, n_{H,i})$
  - false positive part $fp_i = max(0, n_i - tn_i)$
  - the false negative part $fn_i = max(0, p_i - tp_i)$
- and then compute

$$TP_H = \sum_{i=1}^{T} tp_i, FP_H = \sum_{i=1}^{T} fp_i, TN_H = \sum_{i=1}^{T} tn_i, FN_H = \sum_{i=1}^{T} fn_i$$

ProbFOIL

Outer or "Covering" Loop

- if $p_{h,i} > p_i$, the hypothesis overestimates the positive part of the example, and hence, the true positive part is still p_i but the false positive part will be non-zero;
- if $p_{h,i} < p_i$, the true negative part is still n_i but the false negative part will be nonzero



ProbFOIL

Inner greedy Loop

- Searches greedily for the a clause at a time (lines 5-8) according to a local scoring function, until some local stopping criterion is satisfied
- Starts from a clause c with empty body, and repeatedly adds literals l to it
- To determine the possible literal, a refinement operator ρ is applied to the current clause and the clause with body $[b, l]$ is evaluated through a **local score**
- $localstop(H, c) = (TP(H \cup c) - TP(H) = 0) \vee (FP(c) = 0)$ (c does not cover any extra negative part or any extra positive part any more)

ProbFOIL

Inner greedy Loop

- *Local score*(c) = *m-estimate of the precision of* $c = \frac{TP_c + m\frac{P}{P+N}}{TP_c + FP_c + m}$
- *Local score*(H) = *m-estimate of the precision of* $H = \frac{TP_H + m\frac{P}{P+N}}{TP_H + FP_H + m}$
- *Local score*(H, c) = m-estimate$(H \cup c)$ - m-estimate(H) (each rule may only cover fractions of the examples)
- m-estimate is more robust against noise in the training data than plain precision

ProbFOIL

Inner greedy Loop

- *Specializing a clause* (adding literals to it) can only decrease the probability of examples
- Traditional deterministic rule learners typically *manipulate the set of examples, e.g. deleting the already covered examples*: this operation is warranted because if one rule in the hypothesis covers the example, the overall hypothesis will cover the example
- ProbFOIL *takes into account all examples all the time*, as a given rule may only cover part of the example, and therefore a multi-rule hypothesis may be needed to cover the full positive part of an example

ProbFOIL+

- ProbFOIL learns definite clauses, i.e., of the form $h \leftarrow B$
- **ProbFOIL+** learns probabilistic clauses, i.e., of the form $x :: h \leftarrow B$, with $x \in [0, 1]$
- x: the probability that the body of the clause entails its head
- It is based, as ProbFOIL, on two nested loops and on the same scoring functions

ProbFOIL+

Algorithm 1 The ProbFOIL$^+$ learning algorithm.

```
1:  function ProbFOIL+(target)
2:    H := ∅
3:    while true do
4:      clause := LearnRule(H, target)
5:      if GScore(H) < GScore(H ∪ {clause}) then
6:        H := H ∪ {clause}
7:      else return H
8:  function LearnRule(H, target)
9:    candidates := {x :: target ← true}
10:   best := (x :: target ← true)
11:   while candidates ≠ ∅ do
12:     next_cand := ∅
13:     for all x :: target ← body ∈ candidates do
14:       for all refinement ∈ ρ(target ← body) do
15:         if not Reject(H, best, x :: target ← body) then
16:           next_cand := next_cand ∪ {x :: target ← body ∧
17:                                     refinement}
18:           if LScore (H, x :: target ← body ∧ refinement) >
19:                                     LScore(H, best) then
20:             best := (x :: target ← body ∧ refinement)
21:     candidates := next_cand
22:   return best
```

ProbFOIL+

Computing the probability x

- Done in the inner loop: it searches for the clause $c(x) = (x :: c)$ that maximizes the local scoring function m-estimate(x)=$M(x)$
- $x = argmax_x\ M(x)$
- m-estimate(x) $= \frac{TP_{H\cup c(x)} + mP/(N+P)}{TP_{H\cup c(x)} + FP_{H\cup c(x)} + m}$
- Need to express the contingency table of $H \cup c(x)$ in function of x
- For each example e_i, we can decompose $TP_{H\cup c(x),i}$ and $FP_{H\cup c(x),i}$ in

$$TP_{H\cup c(x),i} = TP_{H,i} + TP_{c(x),i} \quad FP_{H\cup c(x),i} = FP_{H,i} + FP_{c(x),i}$$

where $TP_{c(x)}$ and $FP_{H\cup c(x),i}$ indicate the additional contribution of clause $c(x)$ to the true and false positive rates

ProbFOIL+

Computing the probability x

- $\frac{dM(x)}{dx}$ is a piecewise function, which is either 0 or different from 0 everywhere in each interval so the maximum of $M(x)$ can only occur at the x_i values that are the endpoints of the intervals
- Compute the value of $M(x)$ for each x_i and pick the maximum

ProbFOIL+

Inner loop: refinements

In order for a refinement to be a viable candidate it has to have

1. a higher local score than the current best rule (line 18)
2. has a *significance* that is high enough (according to a preset threshold); it si computed as a function of TP/FP and precision
3. has a better global score than the current rule set without the additional clause

- ProbFOIL+ uses a declarative bias based on modes Muggleton (1995) to specify syntactic restrictions on the clauses of interest and are used by the refinement operator during the search process
- It uses relational path finding to generate clauses by considering the connections be- tween the variables in the example literals
- ProbFOIL+ computes the probabilities $p_{H,i}$ using the ProbLog2 system
- ProbFOIL+ outperforms ProbFOIL

PLP Online

- cplint on SWISH: for writing and run reasoning tasks on **LPADs** online
- http://cplint.eu/ based on the SWISH web front-end for SWI-Prolog
- **Inference**:
  - Exact inference with PITA, approximate inference with MCINTYRE
  - Inference on Hybrid Probabilistic Logic Programs, where some of the random variables are continuous
  - Allows to draw BDDs and SLDNF trees
  - Allows to draw probability density functions when the program has continuous random variables
- **Parameter learning**: EMBLEM
- **Structure learning**
  - Aleph (non probabilistic)
  - SLIPCOVER, LEMUR (Di Mauro et al. (2015))
- Allows to draw ROC and PR curves to evaluate performance

PLP Online

- **ProbLog1** (Kimmig et al. (2011)):
  `https://dtai.cs.kuleuven.be/problog/problog1/problog1.html`
  - integrated in YAP Prolog
  - employs BDDs
- **ProbLog2**: `https://dtai.cs.kuleuven.be/problog/`
  - Written in Python
  - Employs d-DNNFs
  - **Inference**: exact (unconditional, conditional, MAP and MPE) and approximate (Monte Carlo)
  - **Parameter learning**: from partial interpretations (LFI-ProbLog)
  - Structure Learning: ProbFOIL (can only be downloaded)

Applications

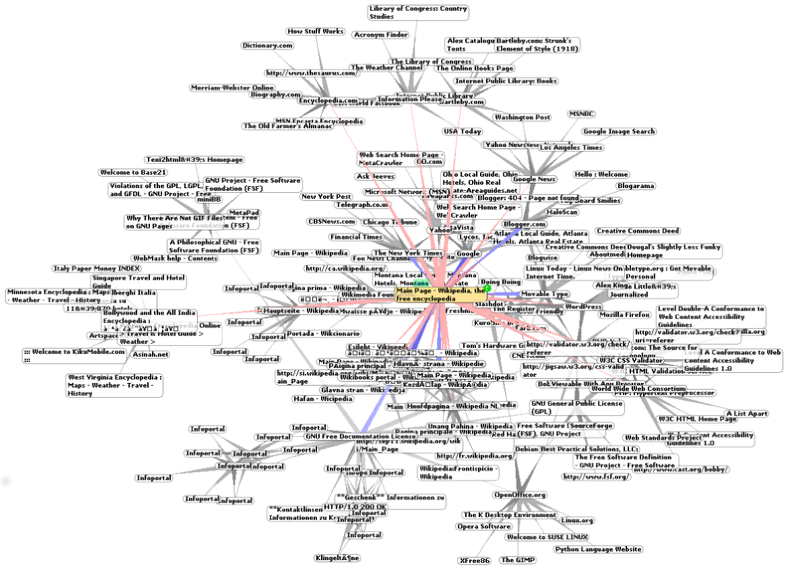
- **Link Prediction**: given a (social) network, compute the probability of the existence of a link between two entities

- *Example*. UWCSE: University of Washington Department of Computer Science and Engineering

```
advisedby(X, Y):0.7:- publication(P, X),
                      publication(P, Y),
                      student(X).
```

Applications

- **Classification**: classify web pages on the basis of the link structure

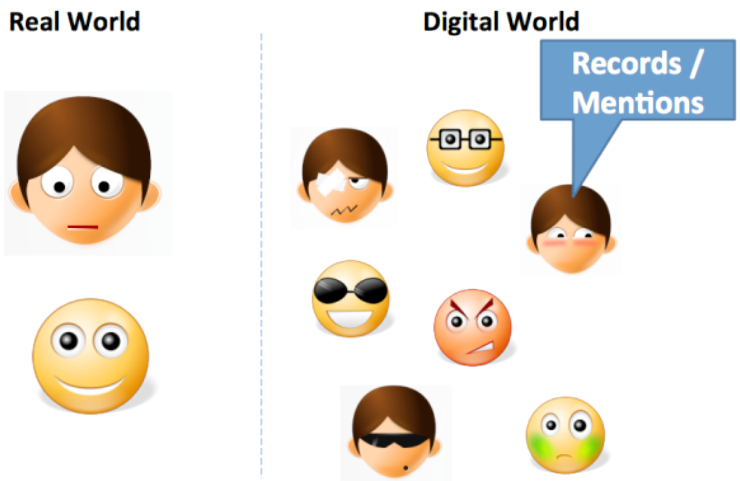


- *Example*. WebKB: WWW-pages from computer science departments of four American universities

```
coursePage(Page1):0.3:-linkTo(Page2,Page1),coursePage(Page2).
coursePage(Page1):0.6:-linkTo(Page2,Page1),facultyPage(Page2).
coursePage(Page): 0.9:-has('syllabus',Page).
...
```

Applications

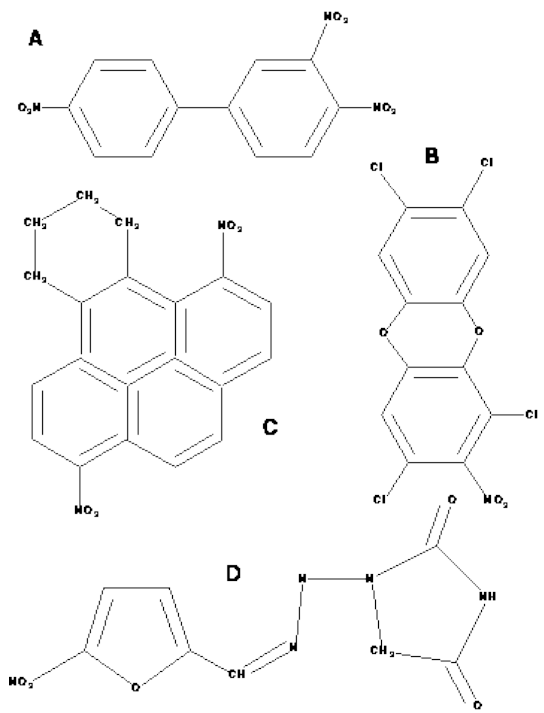
- **Entity resolution**: identify identical entities in texts or databases
- *Example*. Cora: scientific publications




```
samebib(A,B):0.9 :-
samebib(A,C), samebib(C,B).
sameauthor(A,B):0.6 :-
  sameauthor(A,C), sameauthor(C,B).
sametitle(A,B):0.7 :-
  sametitle(A,C), sametitle(C,B).
samevenue(A,B):0.65 :-
  samevenue(A,C), samevenue(C,B).
samebib(B,C):0.5 :-
  author(B,D),author(C,E),sameauthor(D,E).
samebib(B,C):0.7 :-
  title(B,D),title(C,E),sametitle(D,E).
samebib(B,C):0.6 :-
  venue(B,D),venue(C,E),samevenue(D,E).
samevenue(B,C):0.3 :-
  haswordvenue(B,logic),
  haswordvenue(C,logic).
...
```

Applications

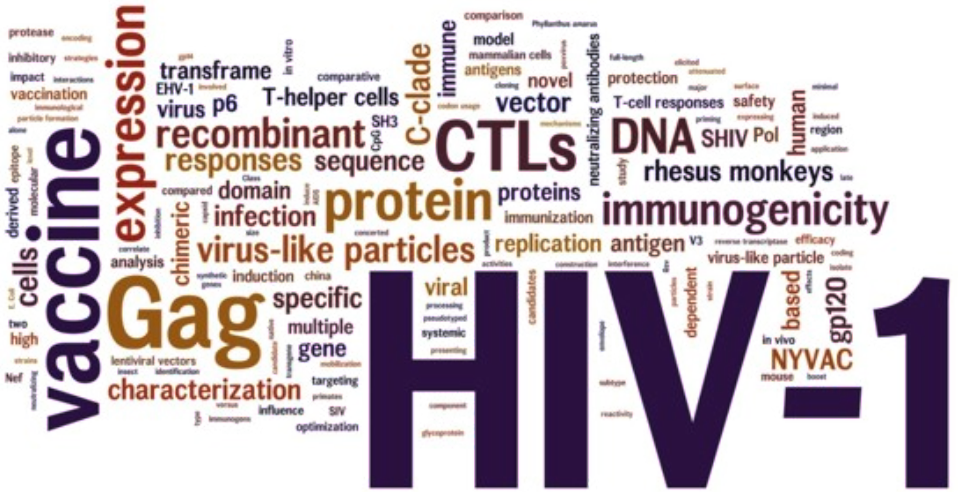
- **Chemistry**: given the chemical composition of a substance, predict its mutagenicity or its carcenogenicity
- *Example*. Mutagenesis




```
active(A):0.4 :-
   atm(A,B,c,29,C),
   gteq(C,-0.003),
   ring_size_5(A,D).
active(A):0.6:-
   lumo(A,B), lteq(B,-2.072).
active(A):0.3 :-
   bond(A,B,C,2),
   bond(A,C,D,1),
   ring_size_5(A,E).
active(A):0.7 :-
   carbon_6_ring(A,B).
...
```

Applications

- **Medicine**: diagnose diseases on the basis of patient information (Hepatitis), influence of genes on HIV, risk of falling of elderly people



Conclusions

- Exciting field!
- Much is left to do:
  - Lifted inference
  - Continuous variables
  - Structure learning search strategies

References I

Bellodi, E. and Riguzzi, F. (2013). Expectation Maximization over binary decision diagrams for probabilistic logic programs. *Intelligent Data Analysis*, 17(2).

Bellodi, E. and Riguzzi, F. (2015). Structure learning of probabilistic logic programs by searching the clause space. *TPLP*, 15(2):169–212.

De Raedt, L., Dries, A., Thon, I., Van Den Broeck, G., and Verbeke, M. (2015). Inducing probabilistic relational rules from probabilistic examples. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, pages 1835–1843. AAAI Press.

De Raedt, L. and Thon, I. (2011). Probabilistic rule learning. In Frasconi, P. and Lisi, F. A., editors, *Inductive Logic Programming*, pages 47–58, Berlin, Heidelberg. Springer Berlin Heidelberg.

References II

Dempster, A. P., Laird, N. M., and Rubin, D. B. (1977). Maximum likelihood from incomplete data via the em algorithm. *JOURNAL OF THE ROYAL STATISTICAL SOCIETY, SERIES B*, 39(1):1–38.

Di Mauro, N., Bellodi, E., and Riguzzi, F. (2015). Bandit-based Monte-Carlo structure learning of probabilistic logic programs. 100(1):127–156.

Fierens, D., Van den Broeck, G., Renkens, J., Shterionov, D., Gutmann, B., Thon, I., Janssens, G., and De Raedt, L. (2015). Inference and learning in probabilistic logic programs using weighted Boolean formulas. *Theory and Practice of Logic Programming*, 15:358–401.

Gutmann, B., Kimmig, A., Kersting, K., and De Raedt, L. (2008). Parameter learning in probabilistic databases: A least squares approach. In Daelemans, W., Goethals, B., and Morik, K., editors, *Machine Learning and Knowledge Discovery in Databases*, pages 473–488, Berlin, Heidelberg. Springer Berlin Heidelberg.

References III

Gutmann, B., Thon, I., and Raedt, L. D. (2011). Learning the parameters of probabilistic logic programs from interpretations. In *European Conference on Machine Learning and Knowledge Discovery in Databases*, volume 6911 of *LNCS*, pages 581–596. Springer.

Ishihata, M., Kameya, Y., Sato, T., and Minato, S. (2008). Propositionalizing the em algorithm by bdds. In *Late Breaking Papers of the 18th International Conf. on Inductive Logic Programming*, pages 44–49.

Kimmig, A., Demoen, B., De raedt, L., Costa, V. s., and Rocha, R. (2011). On the implementation of the probabilistic logic programming language problog. *Theory Pract. Log. Program.*, 11(2-3):235–262.

Muggleton, S. (1995). Inverse entailment and progol. *New Generation Comput.*, 13(3&4):245–286.

References IV

Quinlan, J. R. (1990). Learning logical definitions from relations. *MACHINE LEARNING*, 5:239–266.

Thon, I., Landwehr, N., and Raedt, L. D. (2008). A simple model for sequences of relational state descriptions. In Daelemans, W., Goethals, B., and Morik, K., editors, *Machine Learning and Knowledge Discovery in Databases, European Conference, ECML/PKDD 2008, Antwerp, Belgium, September 15-19, 2008, Proceedings, Part II*, volume 5212 of *Lecture Notes in Computer Science*, pages 506–521. Springer.