cplint on SWISH Manual

February 6, 2016

1 Syntax

LPAD and CP-logic programs consist of a set of annotated disjunctive clauses. Disjunction in the head is represented with a semicolon and atoms in the head are separated from probabilities by a colon. For the rest, the usual syntax of Prolog is used. A general CP-logic clause has the form

```
h1:p1 ; ... ; hn:pn :- b1,...,bm,\+ c1,....,\+ cl
```

No parentheses are necessary. The `pi` are numeric expressions. It is up to the user to ensure that the numeric expressions are legal, i.e. that they sum up to less than one.

If the clause has an empty body, it can be represented like this

```
h1:p1 ; ... ; hn:pn.
```

If the clause has a single head with probability 1, the annotation can be omitted and the clause takes the form of a normal prolog clause, i.e.

```
h1 :- b1,...,bm,\+ c1,...,\+ cl.
```

stands for

```
h1:1 :- b1,...,bm,\+ c1,...,\+ cl.
```

The coin example of [12] is represented as (file `coin.cpl`)

```
heads(Coin):1/2 ; tails(Coin):1/2 :-
  toss(Coin),\+biased(Coin).

heads(Coin):0.6 ; tails(Coin):0.4 :-
  toss(Coin),biased(Coin).

fair(Coin):0.9 ; biased(Coin):0.1.

toss(coin).
```

The first clause states that if we toss a coin that is not biased it has equal probability of landing heads and tails. The second states that if the coin is biased it has a slightly higher probability of landing heads. The third states that the coin is fair with probability 0.9 and biased with probability 0.1 and the last clause states that we toss a coin with certainty.

Moreover, the bodies of rules may contain the built-in predicates:

```
is/2, >/2, </2, >=/2 ,=</2,
=:=/2, =\=/2, true/0, false/0,
=/2, ==/2, \=/2 ,\==/2, !/0, length/2
```

The bodies may also contain the following library predicates:

```
member/2, max_list/2, min_list/2
nth0/3, nth/3, name/2, float/1,
integer/1, var/1, @>/2,
memberchk/2, select/3, dif/2,
between/3
```

plus the predicate

```
average/2
```

that, given a list of numbers, computes its arithmetic mean.

The body of rules may also contain the predicate `prob/2` that computes the probability of an atom, thus allowing nested probability computations. For example (`meta.pl`)

```
a:0.2:-
  prob(b,P),
  P>0.2.
```

is a valid rule.

Moreover, the probabilistic annotations can be variables, as in (`flexprob.pl`))

```
red(Prob):Prob.

draw_red(R, G):-
  Prob is R/(R + G),
  red(Prob).
```

Variables in probabilistic annotations must be ground when resolution reaches the end of the body, otherwise an exception is raised.

2 Semantics

The semantics of LPADs for the case of programs without functions symbols can be given as follows. An LPAD defines a probability distribution over normal logic programs called *worlds*. A world is obtained from an LPAD by first grounding it, by

selecting a single head atom for each ground clause and by including in the world the clause with the selected head atom and the body. The probability of a world is the product of the probabilities associated to the heads selected. The probability of a ground atom (the query) is given by the sum of the probabilities of the worlds where the query is true.

If the LPAD contains function symbols, the definition is more complex, see [7, 11, 9].

3 Inference

`cplint` answers queries using the module `pita` or `mcintyre`. The first performs the program transformation technique of [10]. Differently from that work, techniques alternative to tabling and answer subsumption are used. The latter performs approximate inference by sampling using a different program transformation technique and is described in [8].

For answering queries, you have to prepare a Prolog file where you first load the inference module (for example `pita`), initialize it with a directive (for example `:- pita`) and then enclose the LPAD clauses in `:-begin_lpad.` and `:-end_lpad.` For example, the coin program above can be stored in `coin.pl` for performing inference with `pita` as follows

```
:- use_module(library(pita)).
:- pita.
:- begin_lpad.
heads(Coin):1/2 ; tails(Coin):1/2:-
toss(Coin),\+biased(Coin).

heads(Coin):0.6 ; tails(Coin):0.4:-
toss(Coin),biased(Coin).

fair(Coin):0.9 ; biased(Coin):0.1.

toss(coin).
:- end_lpad.
```

The same program for `mcintyre` is

```
:- use_module(library(mcintyre)).
:- mc.
:- begin_lpad.
heads(Coin):1/2 ; tails(Coin):1/2:-
toss(Coin),\+biased(Coin).

heads(Coin):0.6 ; tails(Coin):0.4:-
toss(Coin),biased(Coin).
```

```
fair(Coin):0.9 ; biased(Coin):0.1.

toss(coin).
:- end_lpad.
```

You can have also (non-probabilistic) clauses outside `:-begin/end_lpad.` These are considered as database clauses. In `pita` subgoals in the body of probabilistic clauses can query them by enclosing the query in `db/1`. For example (`testdb.pl`)

```
:- use_module(library(pita)).
:- pita.
:- begin_lpad.
sampled_male(X):0.5:-
  db(male(X)).
:- end_lpad.
male(john).
male(david).
```

You can also use `findall/3` on subgoals defined by database clauses (`persons.pl`)

```
:- use_module(library(pita)).
:- pita.
:- begin_lpad.
male:M/P; female:F/P:-
  findall(Male,male(Male),LM),
  findall(Female,female(Female),LF),
  length(LM,M),
  length(LF,F),
  P is F+M.
:- end_lpad.
male(john).
male(david).
female(anna).
female(elen).
female(cathy).
```

Aggregate predicates on probabilistic subgoals are not implemented due to their high computational cost (if the aggregation is over n atoms, the values of the aggregation are potentially 2^n). The Yap version of `cplint` includes reasoning algorithms that allows aggregate predicates on probabilistic subgoals, see `http://ds.ing.unife.it/~friguzzi/software/cplint/manual.html`.

In `mcintyre` you can query database clauses in the body of probabilistic clauses without any special syntax. You can also use `findall/3`.

`cplint` on SWISH allows two types of programs: LPAD and Prolog. In the first, you create a new LPAD in the editor, in the latter a Prolog program.

In LPADs, you write in the editor the program without the library import and the compiler directives. `pita` is used for inference.

You ask queries by writing the atom of which you want to compute the probability. In the coin example, the probability of `heads(coin)` can be obtained with the

```
?- heads(coin).
```

In Prolog programs, you have to enter the program above (`coin.pl`).
You can ask the probability of an atom using `pita` using the predicate using

```
prob(:Query:atom,-Probability:float)|
```

as in

```
?- prob(heads(coin),P).
```

If the query is non-ground, `prob/2` returns in backtracking the succesful instantiations together with their probability.
When using `mcintyre`, the predicate for querying is

```
mc_prob(:Query:atom,-Probability:float)
```

as in

```
?- mc_prob(heads(coin),P).
```

With `mcintyre`, you can also take a given number of sample with

```
mc_sample(:Query:atom,+Samples:int,-Successes:int,-Failures:int,
  -Probability:float).
```

as in (`coinmc.pl`)

```
?- mc_sample(heads(coin),1000,S,F,P).
```

that samples `heads(coin)` 1000 times and returns in `S` the number of successes, in `F` the number of failures and in `P` the estimated probability (`S/1000`).
If you are just interested in the probability, you can use

```
mc_sample(:Query:atom,+Samples:int,-Probability:float)
```

as in (`coinmc.pl`)

```
?- mc_sample(heads(coin),1000,Prob).
```

that samples `heads(coin)` 1000 times and returns the estimated probability that a sample is true (i.e., that a sample succeeds).
Moreover, you can sample arguments of queries with

```
mc_sample_arg(:Query:atom,+Samples:int,?Arg:var,-Values:list).
```

The predicate samples `Query` a number of `Samples` times. `Arg` should be a variable in `Query`. The predicate returns in `Values` a list of couples `L-N` where `L` is the list of values of `Arg` for which `Query` succeeds in a world sampled at random and `N` is the number of samples returning that list of values. If `L` is the empty list, it means that for that sample the query failed. If `L` is a list with a single element, it means that for that sample the query is determinate. If, in all couples `L-N`, `L` is a list with a single element, it means that the clauses in the program are mutually exclusive, i.e., that in every sample, only one clause for each subgoal has the body true. This is one of the assumptions taken for programs of the PRISM system [11]. For example `pfcglr.pl` and `plcg.pl` satisfy this constraint while `markov_chain.pl` and `var_obj.pl` don't.

An example of use of the above predicate is

```
?- mc_sample_arg(reach(s0,0,S),50,S,Values).
```

of `markov_chain.pl` that takes 50 samples of `L` in `findall(S,(reach(s0,0,S),L)`.

You can sample arguments of queries also with

```
mc_sample_arg_first(:Query:atom,+Samples:int,?Arg:var,-Values:list).
```

samples `Query` a number of `Samples` times and returns in `Values` a list of couples `V-N` where `V` is the value of `Arg` returned as the first answer by `Query` in a world sampled at random and `N` is the number of samples returning that value. `V` is failure if the query fails. `mc_sample_arg_first/4` differs from `mc_sample_arg/4` because the first just computes the first answer of the query for each sampled world.

Finally, you can compute expectations with

```
mc_expectation(:Query:atom,+N:int,?Arg:var,-Exp:float).
```

that computes the expected value of `Arg` in `Query` by sampling. It takes `N` samples of `Query` and sums up the value of `Arg` for each sample. The overall sum is divided by `N` to give `Exp`.

An example of use of the above predicate is

```
?- mc_expectation(eventually(elect,T),1000,T,E).
```

of `pctl_slep.pl` that returns in `E` the expected value of `T` by taking 1000 samples.

You can also see the probability of the query being true and being false as a bar chart with `prob_bar(:Query:atom,-Probability:dict)` as in

```
?- prob_bar(heads(coin),P).
```

if you include

```
:- use_rendering(c3).
```

before `:- pita.` `P` will be instantiated with a dict for rendering with `c3`. It will be shown as a bar chart with a bar for the probability of `heads(coin)` true and a bar for the probability of `heads(coin)` false.

When using `mcintyre`, you can use

```
mc_prob_bar(:Query:atom,-Probability:dict).
```

as in

```
?- mc_prob_bar(heads(coin),P).
```

to obtain a chart representation of the probability.

You can obtain a bar chart of the samples with

```
?- mc_sample_bar(heads(coin),1000,Chart).
```

that returns in `Chart` a diagram with one bar for the number of successes and one bar for the number of failures.

You can also graph the results of sampling arguments with

```
mc_sample_arg_bar(:Query:atom,+Samples:int,?Arg:var,-Chart:dict).
mc_sample_arg_first_bar(:Query:atom,+Samples:int,?Arg:var,-Chart:dict) .
```

that return in `Chart` a bar chart with a bar for each possible sampled value whose size is the number of samples returning that value.

An example is

```
?- mc_sample_arg_bar(reach(s0,0,S),50,S,Chart).
```

of `markov_chain.pl`.

3.1 Parameters

The inference modules have a number of parameters in order to control their behavior. They can be set with the directive

```
:- set_pita(<parameter>,<value>).
```

or

```
:- set_mc(<parameter>,<value>).
```

after initialization (`:-pita.` or `:-mc.`) but outside `:-begin/end_lpad.` The current value can be read with

```
?- setting_pita(<parameter>,Value).
```

or

```
?- setting_mc(<parameter>,Value).
```

from the top-level. The available parameters common to both `pita` and `mcintyre` are:

- `epsilon_parsing`: if (1 - the sum of the probabilities of all the head atoms) is larger than `epsilon_parsing`, then `pita` adds the null event to the head. Default value `0.00001`.

- `single_var`: determines how non ground clauses are treated: if `true`, a single random variable is assigned to the whole non ground clause, if `false`, a different random variable is assigned to every grounding of the clause. Default value `false`.

Moreover, `pita` has the parameters

- `depth_bound`: if `true`, the depth of the derivation of the goal is limited to the value of the `depth` parameter. Default value `false`.
- `depth`: maximum depth of derivations when `depth_bound` is set to `true`. Default value `5`.

If `depth_bound` is set to `true`, derivations are depth-bounded so you can query also programs containing infinite loops, for example programs where queries have an infinite number of explanations. However the probability that is returned is guaranteed only to be a lower bound, see for example `markov_chaindb.pl`

`mcintyre` has the parameters

- `min_error`: minimal width of the binomial proportion confidence interval for the probability of the query. When the confidence interval for the probability of the query is below `min_error`, the computation stops. Default value `0.01`.
- `k`: the number of samples to take before checking whether the the binomial proportion confidence interval is below `min_error`. Default value `1000`. `max_samples`: the maximum number of samples to take. This is used when the probability of the query is very close to 0 or 1. In fact `mcintyre` also checks for the validity of the the binomial proportion confidence interval: if less than 5 failures or successes are sampled, even if the width of the confidence interval is less than `min_error`, the computation continues. This would lead to non-termination in cases where the probability is 0 or 1. `max_samples` ensures termination. Default value `10e4`.

The example `markov_chain.pl` shows that `mcintyre` can perform inference in presence of an infinite number of explanations for the goal. Differently from `pita`, no depth bound is necessary, as the probability of selecting the infinite computation branch is 0. However, also `mcintyre` may not terminate if loops not involving probabilistic predicates are present.

If you want to set the seed of the random number generator, you can use SWI-Prolog predicates `setrand/1` and `getrand/1`, see SWI-Prolog manual.

4 Learning

The following learning algorithms are available:

- EMBLEM (EM over Bdds for probabilistic Logic programs Efficient Mining): an implementation of EM for learning parameters that computes expectations directly on BDDs [3], [1], [2]

- SLIPCOVER (Structure LearnIng of Probabilistic logic programs by searChing OVER the clause space): an algorithm for learning the structure of programs by searching the clause space and the theory space separately [4]

4.1 Input

To execute the learning algorithms, prepare a Prolog file divided in five parts

- preamble
- background knowledge, i.e., knowledge valid for all interpretations
- LPAD/CPL-program for you which you want to learn the parameters (optional)
- language bias information
- example interpretations

The preamble must come first, the order of the other parts can be changed.

For example, consider the Bongard problems of [6]. `bongard.pl` and `bongardkeys.pl` represent a Bongard problem. Let us consider `bongard.pl`.

4.1.1 Preamble

In the preamble, the SLIPCOVER library is loaded with

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

Now you can initialize SLIPCOVER with

```
:- sc.
```

At this point you can start setting parameters for SLIPCOVER such as for example

```
:- set_sc(megaex_bottom,20).
:- set_sc(max_iter,2).
:- set_sc(max_iter_structure,5).
:- set_sc(verbosity,1).
```

We will see later the list of available parameters. A particularly important parameter is `verbosity`: if set to 1, nothing is printed and learning is fastest, if set to 3 much information is printed and learning is slowest, 2 is in between. This ends the preamble.

4.1.2 Background and Initial LPAD/CPL-program

Now you can specify the background knowledge with a fact of the form

```
bg(<list of terms representing clauses>).
```

where the clauses must currently be deterministic. Alternatively, you can specify a set of clauses by including them in a section between `:- begin_bg.` and `:- end_bg.` For example

```
:- begin_bg.
replaceable(gear).
replaceable(wheel).
replaceable(chain).
not_replaceable(engine).
not_replaceable(control_unit).
component(C):-
  replaceable(C).
component(C):-
  not_replaceable(C).
:- end_bg.
```

from the `mach.pl` example. If you specify both a `bg/1` fact and a section, the clauses of the two will be combined.

Moreover, you can specify an initial program with a fact of the form

```
in(<list of terms representing clauses>).
```

The initial program is used in parameter learning for providing the structure. The indicated parameters do not matter as they are first randomized. Remember to enclose each clause in parentheses because `:-` has the highest precedence.

For example, `bongard.pl` has the initial program

```
in([(pos:0.197575 :-
       circle(A),
       in(B,A)),
    (pos:0.000303421 :-
       circle(A),
       triangle(B)),
    (pos:0.000448807 :-
       triangle(A),
       circle(B))]).
```

Alternatively, you can specify an input program in a section between `:- begin_in.` and `:- end_in.`, as for example

```
:- begin_in.
pos:0.197575 :-
  circle(A),
  in(B,A).
pos:0.000303421 :-
  circle(A),
  triangle(B).
```

```
pos:0.000448807 :-
  triangle(A),
  circle(B).
:- end_in.
```

If you specify both a `in/1` fact and a section, the clauses of the two will be combined.

4.1.3 Language Bias

The language bias part contains the declarations of the input and output predicates. Output predicates are declared as

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

and indicate the predicate whose atoms you want to predict. Derivations for the atoms for this predicates in the input data are built by the system. These are the predicates for which new clauses are generated.

Input predicates are those whose atoms you are not interested in predicting. You can declare closed world input predicates with

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

For these predicates, the only true atoms are those in the interpretations and those derivable from them using the background knowledge, the clauses in the input/hypothesized program are not used to derive atoms for these predicates. Moreover, clauses of the background knowledge that define closed world input predicates and that call an output predicate in the body will not be used for deriving examples.

Open world input predicates are declared with

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

In this case, if a subgoal for such a predicate is encountered when deriving a subgoal for the output predicates, both the facts in the interpretations, those derivable from them and the background knowledge, the background clauses and the clauses of the input program are used.

Then, you have to specify the language bias by means of mode declarations in the style of Progol.

```
modeh(<recall>,<predicate>(<arg1>,...)).
```

specifies the atoms that can appear in the head of clauses, while

```
modeb(<recall>,<predicate>(<arg1>,...)).
```

specifies the atoms that can appear in the body of clauses. `<recall>` can be an integer or `*`. `<recall>` indicates how many atoms for the predicate specification are retained in the bottom clause during a saturation step. `*` stands for all those that are found. Otherwise the indicated number is randomly chosen.

Two specialization modes are available: `bottom` and `mode`. In the first, a bottom clause is built and the literals to be added during refinement are taken from it. In the latter, no bottom clause is built and the literals to be added during refinement are generated directly from the mode declarations.

Arguments of the form

`+<type>`

specifies that the argument should be an input variable of type `<type>`, i.e., a variable replacing a `+<type>` argument in the head or a `-<type>` argument in a preceding literal in the current hypothesized clause.

Another argument form is

`-<type>`

for specifying that the argument should be a output variable of type `<type>`. Any variable can replace this argument, either input or output. The only constraint on output variables is that those in the head of the current hypothesized clause must appear as output variables in an atom of the body.

Other forms are

`#<type>`

for specifying an argument which should be replaced by a constant of type `<type>` in the bottom clause but should not be used for replacing input variables of the following literals when building the bottom clause or

`-#<type>`

for specifying an argument which should be replaced by a constant of type `<type>` in the bottom clause and that should be used for replacing input variables of the following literals when building the bottom clause.

`<constant>`

for specifying a constant.

Note that arguments of the form `#<type>` `-#<type>` are not available in specialization mode `mode`, if you want constants to appear in the literals you have to indicate them one by one in the mode declarations.

An example of language bias for the Bongard domain is

```
output(pos/0).

input_cw(triangle/1).
input_cw(square/1).
input_cw(circle/1).
input_cw(in/2).
input_cw(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)).
```

SLIPCOVER also requires facts for the `determination/2` predicate Aleph-style that indicate which predicates can appear in the body of clauses. For example

```
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).
```

state that `triangle/1` can appear in the body of clauses for `pos/0`.

SLIPCOVER also allows mode declarations of the form

```
modeh(<r>,[<s1>,...,<sn>],[<a1>,...,<an>],[<P1/Ar1>,...,<Pk/Ark>]).
```

These mode declarations are used to generate clauses with more than two head atoms. In them, `<s1>,...,<sn>` are schemas, `<a1>,...,<an>` are atoms such that `<ai>` is obtained from `<si>` by replacing placemarkers with variables, `<Pi/Ari>` are the predicates admitted in the body. `<a1>,...,<an>` are used to indicate which variables should be shared by the atoms in the head. An example of such a mode declaration (from `uwcselearn.pl`) is

```
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]).
```

If you want to specify negative literals for addition in the body of clauses, you should define a new predicate in the background as in

```
not_worn(C):-
  component(C),
  \+ worn(C).
one_worn:-
  worn(_).
none_worn:-
  \+ one_worn.
```

from `mach.pl` and add the new predicate in a `modeb/2` fact

```
modeb(*,not_worn(-comp)).
modeb(*,none_worn).
```

Note that successful negative literals do not instantiate the variables, so if you want a variable appearing in a negative literal to be an output variable you must instantiate before calling the negative literals. The new predicates must also be declared as input

```
input_cw(not_worn/1).
input_cw(none_worn/0).
```

Lookahead can also be specified with facts of the form

```
lookahead(<literal>,<list of literals>).
```

In this case when a literal matching `<literal>` is added to the body of clause during refinement, then also the literals matching `<list of literals>` will be added. An example of such declaration (from `muta.pl`) is

```
lookahead(logp(_),[(_=_)]).
```

Note that `<list of literals>` is copied with `copy_term/2` before matching, so variables in common between `<literal>` and `<list of literals>` may not be in common in the refined clause.

It is also possible to specify that a literal can only be added together with other literals with facts of the form

```
lookahead_cons(<literal>,<list of literals>).
```

In this case `<literal>` is added to the body of clause during refinement only together with literals matching `<list of literals>`. An example of such declaration is

```
lookahead_cons(logp(_),[(_=_)]).
```

Also here `<list of literals>` is copied with `copy_term/2` before matching, so variables in common between `<literal>` and `<list of literals>` may not be in common in the refined clause.

Moreover, we can specify lookahead with

```
lookahead_cons_var(<literal>,<list of literals>).
```

In this case `<literal>` is added to the body of clause during refinement only together with literals matching `<list of literals>` and `<list of literals>` is not copied before matching, so variables in common between `<literal>` and `<list of literals>` are in common also in the refined clause. This is allowed only with `specialization` set to `bottom`. An example of such declaration is

```
lookahead_cons_var(logp(B),[(B=_)]).
```

4.1.4 Example Interpretations

The last part of the file contains the data. You can specify data with two modalities: models and keys. In the models type, you specify an example model (or interpretation or megaexample) as a list of Prolog facts initiated by `begin(model(<name>)).` and terminated by `end(model(<name>)).` as in

```
begin(model(2)).
pos.
triangle(o5).
config(o5,up).
square(o4).
in(o4,o5).
circle(o3).
triangle(o2).
config(o2,up).
in(o2,o3).
triangle(o1).
config(o1,up).
end(model(2)).
```

The interpretations may contain a fact of the form

```
prob(0.3).
```

assigning a probability (0.3 in this case) to the interpretations. If this is omitted, the probability of each interpretation is considered equal to $1/n$ where n is the total number of interpretations. `prob/1` can be used to set a different multiplicity for the interpretations.

The facts in the interpretation are loaded in SWI-Prolog database by adding an extra initial argument equal to the name of the model. After each interpretation is loaded, a fact of the form `int(<id>)` is asserted, where `id` is the name of the interpretation. This can be used in order to retrieve the list of interpretations.

Alternatively, with the keys modality, you can directly write the facts and the first argument will be interpreted as a model identifier. The above interpretation in the keys modality is

```
pos(2).
triangle(2,o5).
config(2,o5,up).
square(2,o4).
in(2,o4,o5).
circle(2,o3).
triangle(2,o2).
config(2,o2,up).
in(2,o2,o3).
triangle(2,o1).
config(2,o1,up).
```

which is contained in the `bongardkeys.pl` This is also how model 2 above is stored in SWI-Prolog database. The two modalities, models and keys, can be mixed in the same file. Facts for `int/1` are not asserted for interpretations in the key modality but can be added by the user explicitly.

Note that you can add background knowledge that is not probabilistic directly to the file writing the clauses taking into account the model argument. For example (`carc.pl`) contains

```
connected(_M,Ring1,Ring2):-
  Ring1 \= Ring2,
  member(A,Ring1),
  member(A,Ring2), !.

symbond(Mod,A,B,T):- bond(Mod,A,B,T).
symbond(Mod,A,B,T):- bond(Mod,B,A,T).
```

where the first argument of all the atoms is the model.

Example `registration.pl` contains for example

```
party(M,P):-
  participant(M,_, _, P, _).
```

that defines intensionally the target predicate `party/1`. Here `M` is the model and `participant/4` is defined in the interpretations. You can also define intensionally the negative examples with

```
neg(party(M,yes)):- party(M,no).
neg(party(M,no)):- party(M,yes).
```

Then you must indicate how the examples are divided in folds with facts of the form: `fold(<fold_name>,<list of model identifiers>)`, as for example

```
fold(train,[2,3,...]).
fold(test,[490,491,...]).
```

As the input file is a Prolog program, you can define intensionally the folds as in

```
fold(all,F):-
  findall(I,int(I),F).
```

`fold/2` is dynamic so you can also write (`registration.pl`)

```
:- fold(all,F),
   sample(4,F,FTr,FTe),
   assert(fold(rand_train,FTr)),
   assert(fold(rand_test,FTe)).
```

which however must be inserted after the input interpretations otherwise the facts for `int/1` will not be available and the fold `all` would be empty. This command uses `sample(N,List,Sampled,Rest)` exported from `slipcover` that samples `N` elements from `List` and returns the sampled elements in `Sampled` and the rest in `Rest`. If `List` has `N` elements or less, `Sampled` is equal to `List` and `Rest` is empty.

4.2 Commands

4.2.1 Parameter Learning

To execute EMBLEM, prepare an input file in the editor panel as indicated above and call

```
?- induce_par(<list of folds>,P).
```

where `<list of folds>` is a list of the folds for training and `P` will contain the input program with updated parameters.

For example `bongard.pl`, you can perform parameter learning on the `train` fold with

```
?- induce_par([train],P).
```

A program can also be tested on a test set with

```
?- test(<program>,<list of folds>,LL,AUCROC,ROC,AUCPR,PR).
```

where `<program>` is a list of terms representing clauses and `<list of folds>` is a list of folds. This returns the log likelihood of the test examples in `LL`, the Area Under the ROC curve in `AUCROC`, a dictionary containing the list of points (in the form of Prolog pairs `x-y`) of the ROC curve in `ROC`, the Area Under the PR curve in `AUCPR`, a dictionary containing the list of points of the PR curve in `PR`.

For example, to test on fold `test` the program learned on fold `train` you can run the query

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

Or you can test the input program on the fold `test` with

```
?- in(P),
   test(P,[test],LL,AUCROC,ROC,AUCPR,PR).
```

By including

```
:- use_rendering(c3).
:- use_rendering(lpad).
```

in the code before `:- sc.` the curves will be shown as graphs and the output program will be pretty printed.

4.2.2 Structure Learning

To execute SLIPCOVER, prepare an input file in the editor panel as indicated above and call

```
?- induce(<list of folds>,P).
```

where `<list of folds>` is a list of the folds for training and `P` will contain the learned program.

For example `bongard.pl`, you can perform structure learning on the `train` fold with

```
?- induce([train],P).
```

A program can also be tested on a test set with `test/7` as described above.

4.3 Parameters

Parameters are set with commands of the form

```
:- set_sc(<parameter>,<value>).
```

The available parameters are:

- `specialization`: (values: `{bottom,mode}`, default value: `bottom`) specialization mode.
- `depth_bound`: (values: `{true,false}`, default value: `true`) if `true`, the depth of the derivation of the goal is limited to the value of the `depth` parameter.
- `depth` (values: integer, default value: 2): depth of derivations if `depth_bound` is set to `true`
- `single_var` (values: `{true,false}`, default value: `false`): if set to `true`, there is a random variable for each clause, instead of a different random variable for each grounding of each clause
- `epsilon_em` (values: real, default value: 0.1): if the difference in the log likelihood in two successive parameter EM iteration is smaller than `epsilon_em`, then EM stops
- `epsilon_em_fraction` (values: real, default value: 0.01): if the difference in the log likelihood in two successive parameter EM iteration is smaller than `epsilon_em_fraction`*(-current log likelihood), then EM stops
- `iter` (values: integer, defualt value: 1): maximum number of iteration of EM parameter learning. If set to -1, no maximum number of iterations is imposed
- `iterREF` (values: integer, defualt value: 1, valid for SLIPCOVER): maximum number of iteration of EM parameter learning for refinements. If set to -1, no maximum number of iterations is imposed.
- `random_restarts_number` (values: integer, default value: 1, valid for EMBLEM and SLIPCOVER): number of random restarts of parameter EM learning
- `random_restarts_REFnumber` (values: integer, default value: 1, valid for SLIPCOVER): number of random restarts of parameter EM learning for refinements

- `setrand` (values: rand(integer,integer,integer)): seed for the random functions, see SWI-Prolog manual for the allowed values
- `logzero` (values: negative real, default value log(0.000001)): value assigned to $\log 0$
- `max_iter` (values: integer, default value: 10, valid for SLIPCOVER): number of interations of beam search
- `max_var` (values: integer, default value: 4, valid for SLIPCOVER): maximum number of distinct variables in a clause
- `beamsize` (values: integer, default value: 100, valid for SLIPCOVER): size of the beam
- `megaex_bottom` (values: integer, default value: 1, valid for SLIPCOVER): number of mega-examples on which to build the bottom clauses
- `initial_clauses_per_megaex` (values: integer, default value: 1, valid for SLIPCOVER): number of bottom clauses to build for each mega-example (or model or interpretation)
- `d` (values: integer, default value: 1, valid for SLIPCOVER): number of saturation steps when building the bottom clause
- `max_iter_structure` (values: integer, default value: 10000, valid for SLIPCOVER): maximum number of theory search iterations
- `background_clauses` (values: integer, default value: 50, valid for SLIPCOVER): maximum numbers of background clauses
- `maxdepth_var` (values: integer, default value: 2, valid for SLIPCOVER): maximum depth of variables in clauses (as defined in [5]).
- `neg_ex` (values: `given`, `cw`, default value: `cw`): if set to `given`, the negative examples in testing are taken from the test folds interpretations, i.e., those examples `ex` stored as `neg(ex)`; if set to `cw`, the negative examples are generated according to the closed world assumption, i.e., all atoms for target predicates that are not positive examples. The set of all atoms is obtained by collecting the set of constants for each type of the arguments of the target predicate.
- `verbosity` (values: integer in [1,3], default value: 1): level of verbosity of the algorithms

5 Manual in PDF

A PDF version of the manual is available at `https://github.com/friguzzi/cplint/blob/master/doc/help-cplint.pdf`.

6 Bibliography

References

[1] Elena Bellodi and Fabrizio Riguzzi. EM over binary decision diagrams for probabilistic logic programs. In *Proceedings of the 26th Italian Conference on Computational Logic (CILC2011), Pescara, Italy, 31 August 31-2 September, 2011*, 2011.

[2] Elena Bellodi and Fabrizio Riguzzi. EM over binary decision diagrams for probabilistic logic programs. Technical Report CS-2011-01, Dipartimento di Ingegneria, Università di Ferrara, Italy, 2011.

[3] Elena Bellodi and Fabrizio Riguzzi. Expectation Maximization over binary decision diagrams for probabilistic logic programs. *Intel. Data Anal.*, 16(6), 2012.

[4] Elena Bellodi and Fabrizio Riguzzi. Structure learning of probabilistic logic programs by searching the clause space. *Theory and Practice of Logic Programming*, 2013.

[5] William W. Cohen. Pac-learning non-recursive prolog clauses. *Artif. Intell.*, 79(1):1–38, 1995.

[6] L. De Raedt and W. Van Laer. Inductive constraint logic. In *Proceedings of the 6th Conference on Algorithmic Learning Theory (ALT 1995)*, volume 997 of *LNAI*, pages 80–94, Fukuoka, Japan, 1995. Springer.

[7] David Poole. The independent choice logic for modelling multiple agents under uncertainty. *Artificial Intelligence*, 94(1-2):7–56, 1997.

[8] Fabrizio Riguzzi. MCINTYRE: A Monte Carlo system for probabilistic logic programming. *Fundamenta Informaticae*, 124(4):521–541, 2013.

[9] Fabrizio Riguzzi. The distribution semantics is well-defined for all normal programs. In Fabrizio Riguzzi and Joost Vennekens, editors, *Proceedings of the 2nd International Workshop on Probabilistic Logic Programming (PLP)*, volume 1413 of *CEUR Workshop Proceedings*, pages 69–84, Aachen, Germany, 2015. Sun SITE Central Europe.

[10] Fabrizio Riguzzi and Terrance Swift. Tabling and Answer Subsumption for Reasoning on Logic Programs with Annotated Disjunctions. In *Technical Communications of the International Conference on Logic Programming*, volume 7 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 162–171. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010.

[11] Taisuke Sato and Yoshitaka Kameya. Parameter learning of logic programs for symbolic-statistical modeling. *J. Artif. Intell. Res.*, 15:391–454, 2001.

[12] J. Vennekens, S. Verbaeten, and M. Bruynooghe. Logic programs with annotated disjunctions. In *International Conference on Logic Programming*, volume 3131 of *LNCS*, pages 195–209. Springer, 2004.