1 LP$^{\text{MLN}}$ Learning System Usage

This document provides information needed for install and use the LP$^{\text{MLN}}$ learning system.

The LP$^{\text{MLN}}$ learning system includes the following components:

- Gradient Ascent Based Learning with MC-ASP
- Pseudo-likelihood Based Learning
- Gradient Ascent Based Learning with Gibbs Sampling
- Gradient Ascent Based Learning with Metropolis-Hasting Sampling
- Sampling Based Inference

1.1 Installation

The system requires the following software package:

- Python 2.7.x
- clingo 5.x (The system assumes clingo 5 can be executed with command "`clingo5`")
- sympy
- Boost

To install the system, follow the steps below:

1. Unzip the compressed file

   ```
   unzip lpmln-learn.zip
   ```

2. Go to `code` directory

   ```
   cd code
   ```

3. Compile `lpmlncompiler` with the command

   ```
   g++ -g -std=c++11 lpmlncompiler.cpp -o lpmlncompiler -L /usr/lib -lboost_regex
   ```

4. Compile `negConv` with the command

   ```
   g++ -g -std=c++11 negConv.cpp -o negconv -L /usr/lib -lboost_regex
   ```

5. Compile `clingo3to4` with the command

   ```
   cd Clingo3to4
   make
   cp clingo3to4 ../
   ```

1.2 Input Format

For learning tasks, the input program is a program of the input format of LPMLN2ASP. The weights to be learned needs to be represented with the place-holder `@getWeight(idx)`, where `idx` is the id of the rule (that LPMLN2ASP generates for the rule). For example,

```
#domain person(X).
#domain person(Y).
person(0..9).

@getWeight(1) pf1(X).
cancer(X) :- smokes(X), pf1(X).
@getWeight(2) pf2(X, Y) :- friends(X, Y).
smokes(Y) :- friends(X, Y), smokes(X), pf2(X, Y).
```

Note that the domains of variables that occur in the rules whose weights are to be learned need to be specified with `#domain` statement.

For learning with MC-ASP sampling, the evidence file is simply a list of constraints, specifying truth value of atoms, for example:

```
:- not smokes("edward").
:- not smokes("frank").
:- not smokes("gary").
:- smokes("bob").
:- smokes("chris").
:- smokes("daniel").
:- smokes("helen").

:- not cancer("anna").
:- not cancer("edward").
:- cancer("bob").
:- cancer("frank").
:- cancer("chris").
:- cancer("daniel").
:- cancer("gary").
:- cancer("helen").
```

For learning with pseudolikelihood, Metropolis-Hasting sampling and Gibbs sampling, the evidence file should specify the truth value of all atoms, of the form

```
atom_name argument1;...;argumentN truth_value
```

`truth_value` is 0 (for FALSE) or 1 (for TRUE).

An example is

```
p 0;0 0
p 0;1 1
```

```
q 0;0 0
q 0;1 1
```

which means $p(0,0)$ and $q(0,0)$ are false, $p(0,1)$ and $q(0,1)$ are true.

For sampling based inference, a file specifying the domains of variables is required. The file should be of the form

```
predicate1 comb1_arg1,...,comb1_argN&...&combM_arg1,...,comb1_argN
...
```

For example,

```
pf1 "t";0&"f";0&"t";1&"f";1
pf2 "t";0&"f";0
```

specifies that the first argument of `pf1` and `pf2` is “`t`” or “`f`”, the second argument of `pf1` ranges over $\{0,1\}$ and the second argument of `pf2` can only be `0`.

1.3 Gradient Ascent Based Learning with MC-ASP

Learning parameters such as learning rate and maximum number of iterations can be configured in the file `code/learn-mcsat.py` and `code/learn-mcsat-em.py`.

1.3.1 With Complete Evidence

The command line to run gradient ascent based learning with MC-ASP is

```
python code/learn-mcsat.py path/to/program path/to/evidence
```

For example

```
python code/learn-mcsat.py benchmarks/smoke/smoke.lpmln
    benchmarks/smoke/smoke-evidence.txt
```

will run MC-ASP based learning on the smoke example.

1.3.2 With Incomplete Evidence

When the evidence is incomplete, the command line is

```
python code/learn-mcsat-em.py path/to/program path/to/evidence
```

For example

```
python code/learn-mcsat-em.py benchmarks/smoke/smoke_plg.lpmln
     benchmarks/smoke/smoke-evidence.txt
```

will run MC-ASP based learning on the smoke example with pf atom.

1.4 Pseudo-likelihood Based Learning

Open the file `lpmln-learn.py`. At the beginning of the file, set the value of the variable `learning_alg` to be "`code/learn-pseudolikelihood.py`".

The command to execute pseudo-likelihood based learning is

```
python lpmln-learn.py path/to/program path/to/evidence
```

1.5 Gradient Ascent Based Learning with Gibbs Sampling and Metropolis-Hasting Sampling

Open the file `lpmln-learn.py`. At the beginning of the file, set the value of the variable `learning_alg` to be "`code/learn-mhsampling.py`" for Metropolis-Hasting sampling or "`code/learn-gibbssampling.py`" for Gibbs sampling.

The command to execute gradient ascent based learning with Metropolis-Hasting sampling or Gibbs sampling is

```
python lpmln-learn.py path/to/program path/to/evidence
```

1.6 Sampling Based Inference

To execute Metropolis-Hasting sampling based inference, the input program needs to be turned into an ASP program with LPMLN2ASP first

```
lpmln2asp -i path/to/program
```

then marginal inference can be executed with

```
clingo code/marginal-mhsampling.py program_turned_into_ASP
```

for metropolis-hasting sampling based marginal inference or

```
python code/marginal-mcsat.py program_turned_into_ASP query_atom domain_file
```

for MC-ASP sampling based marginal inference.

For example,

```
lpmln2asp -i benchmarks/smoke/smoke-withweights.lpmln;
clingo code/marginal-mhsampling.py out.txt
```

will execute marginal inference on the program `benchmarks/smoke/smoke-withweights.lpmln`.

2 Examples

2.1 Network Failure

Consider an unstable communication network described by a graph, for example the one shown in Figure 1, where each node represents a signal station that sends and receives signals. A signal station may fail, making it impossible for signals to go through the station. The graph is defined by a set of edge relation, and each node has a certain chance to fail. The following LP$^{\text{MLN}}$ program describes the graph in Figure 1 and defines the connectivity between nodes:

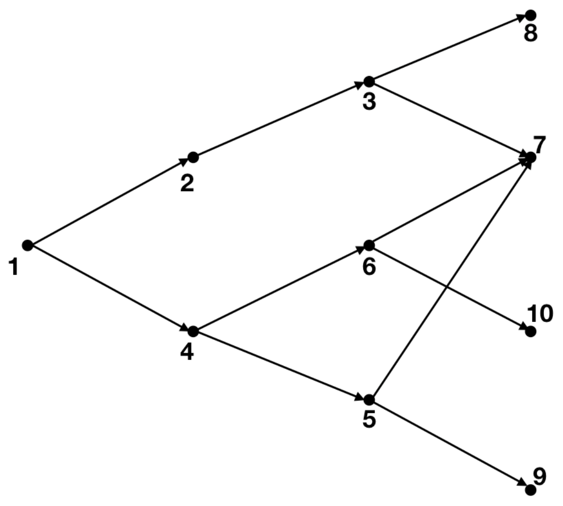

Figure 1: Example Communication Network

```
node(1..10).
session(1..4).
#domain session(T).
@getWeight(1) fail(1, T).
@getWeight(2) fail(2, T).
@getWeight(3) fail(3, T).
@getWeight(4) fail(4, T).
@getWeight(5) fail(5, T).
@getWeight(6) fail(6, T).
@getWeight(7) fail(7, T).
@getWeight(8) fail(8, T).
@getWeight(9) fail(9, T).
@getWeight(10) fail(10, T).

edge(1, 2).
edge(1, 4).
edge(2, 3).
edge(4, 5).
edge(4, 6).
edge(3, 7).
edge(6, 7).
edge(5, 7).
edge(3, 8).
edge(6, 10).
edge(5, 9).

connected(X, Y, T) :- edge(X, Y), not fail(X, T), not fail(Y, T).
connected(X, Y, T) :- connected(X, Z, T), connected(Z, Y, T).
```

We have the following stable model showing the connectivity between two specific stations at several sessions, as the evidence:

```
:- not connected(1, 7, 1).
:- connected(1, 8, 1).
:- not connected(1, 9, 1).
:- connected(1, 10, 1).

:- not connected(1, 7, 2).
:- not connected(1, 8, 2).
:- connected(1, 9, 2).
:- not connected(1, 10, 2).

:- not connected(1, 7, 3).
:- connected(1, 8, 3).
:- connected(1, 9, 3).
:- not connected(1, 10, 3).

:- connected(1, 7, 4).
:- not connected(1, 8, 4).
:- not connected(1, 9, 4).
:- not connected(1, 10, 4).
```

To find out the proper weights for each probabilistic fact $edge(X, Y)$, we execute the following command

```
python code/learn-mcsat-em.py network-prog.lpmln network-evid.txt
```

By default, this executes gradient ascent based learning with MC-ASP sampling for 50 iterations, with 50 MC-ASP iterations each learning iteration. The output looks like

```
Rule 1:  -3.894
Rule 2:  0.648
Rule 3:  1.992
Rule 4:  -3.82
Rule 5:  0.356
Rule 6:  -2.43
Rule 7:  0.186
Rule 8:  0.09
Rule 9:  0.18
Rule 10: -1.15
```

2.2 Smokers

The following program says that smokers influence their friends and defines a social network by specifying friend relations:

```
#domain person(X).
#domain person(Y).

person(0..9).

@getWeight(1) cancer(X) :- smokes(X).
@getWeight(2) smokes(Y) :- friends(X, Y), smokes(X).

friends("anna", "bob").
friends("bob", "anna").
friends("anna", "edward").
friends("edward", "anna").
friends("anna", "frank").
friends("frank", "anna").
friends("bob", "chris").
friends("chris", "bob").
friends("chris", "daniel").
friends("daniel", "chris").
friends("edward", "frank").
friends("frank", "edward").
friends("gary", "helen").
friends("helen", "gary").
friends("gary", "anna").
friends("anna", "gary").

smokes("anna").
```

We have the following stable model specifying whether each person is a smoker and whether each person has cancer:

```
:- not smokes("edward").
:- not smokes("frank").
:- not smokes("gary").
:- smokes("bob").
:- smokes("chris").
:- smokes("daniel").
:- smokes("helen").

:- not cancer("anna").
:- not cancer("edward").
:- cancer("bob").
:- cancer("frank").
:- cancer("chris").
:- cancer("daniel").
:- cancer("gary").
:- cancer("helen").
```

To learn the weights of the rule saying smokers influence their friends and the rule saying smokers tend to have cancer, we execute

```
python code/learn-mcsat-em.py smoker-prog.lpmln smoker-evid.txt
```

By default, this executes gradient ascent based learning with MC-ASP sampling for 50 iterations, with 50 MC-ASP iterations each learning iteration. The output looks like

```
New weights:
Rule 1: 0.66
Rule 2: 0.52
max_diff 0.14
```

The quality of the learned weights (i.e., how close they fit to the evidence) can be checked by comparing the number of ground instances not satisfied by the evidence and the approximated expectation of the number of false ground instances, for each rule. This can be seen in the output of the learning program. The weights tell us that the evidence positive supports both the hypothesis, but not very strong.

2.3 Robot

Consider a robot located in a building with 2 rooms `r1` and `r2` and a book that can be picked up. The robot can move to rooms, pick up the book and put down the book. Sometimes actions may fail: the robot may fail to enter the room or pick up the book, and may drop the book when it has the book. The domain can be modeled in LP$^{\text{MLN}}$ as follows:

```
astep(0).
step(0..1).
boolean("t"; "f").
room("r1"; "r2").
instance(1..12).

#domain astep(AI).
#domain instance(ID).

% Probability Distribution
%% Entering a room fails at probability 0.2
@getWeight(1) pf1(AI, ID).
ab("enter_failed", I, ID) :- pf1(I, ID), ab(I, ID).
%% The robot drops the book at probability 0.1
@getWeight(2) pf2(AI, ID).
ab("drop_book", I, ID) :- pf2(I, ID), ab(I, ID).
%% Picking up fails at probability 0.3
@getWeight(3) pf3(AI, ID).
ab("pickup_failed", I, ID) :- pf3(I, ID), ab(I, ID).
```

```
% UEC
%% Fluents
:- not loc_robot("r1", I, ID), not loc_robot("r2", I, ID), step(I), instance(ID).
:- loc_robot("r1", I, ID), loc_robot("r2", I, ID), step(I), instance(ID).
:- not loc_book("r1", I, ID), not loc_book("r2", I, ID), step(I), instance(ID).
:- loc_book("r1", I, ID), loc_book("r2", I, ID), step(I), instance(ID).
:- not hasBook("t", I, ID), not hasBook("f", I, ID), step(I), instance(ID).
:- hasBook("t", I, ID), hasBook("f", I, ID), step(I), instance(ID).
%% Actions
:- not goto(R, "t", I, ID), not goto(R, "f", I, ID), astep(I), room(R), instance(ID).
:- goto(R, "t", I, ID), goto(R, "f", I, ID), astep(I), room(R), instance(ID).
:- not pickup_book("t", I, ID), not pickup_book("f", I, ID), astep(I), instance(ID).
:- pickup_book("t", I, ID), pickup_book("f", I, ID), astep(I), instance(ID).
:- not putdown_book("t", I, ID), not putdown_book("f", I, ID), astep(I), instance(ID).
:- putdown_book("t", I, ID), putdown_book("f", I, ID), astep(I), instance(ID).

% Effect of Actions
loc_robot(R, I+1, ID) :- goto(R, "t", I, ID), not ab("enter_failed", I, ID), instance(ID).
loc_book(R, I, ID) :- loc_robot(R, I, ID), hasBook("t", I, ID), instance(ID).
hasBook("t", I+1, ID) :- pickup_book("t", I, ID), loc_robot(R, I, ID), loc_book(R, I, ID), n
hasBook("f", I+1, ID) :- putdown_book("t", I, ID), instance(ID).
hasBook("f", I+1, ID) :- ab("drop_book", I, ID), instance(ID).

% Frame Axioms
loc_robot(R, I+1, ID) :- loc_robot(R, I, ID), astep(I), instance(ID), not not loc_robot(R, I
loc_book(R, I+1, ID) :- loc_book(R, I, ID), astep(I), instance(ID), not not loc_book(R, I+1,
hasBook(B, I+1, ID) :- hasBook(B, I, ID), astep(I), instance(ID), not not hasBook(B, I+1, II

% No Concurrency
:- goto(R1, "t", I, ID), goto(R2, "t", I, ID), astep(I), instance(ID), R1 != R2.
:- goto(R, "t", I, ID), pickup_book("t", I, ID), room(R), astep(I), instance(ID).
:- goto(R, "t", I, ID), putdown_book("t", I, ID), room(R), astep(I), instance(ID).
:- pickup_book("t", I, ID), putdown_book("t", I, ID), astep(I), instance(ID).

% Initial state and actions are exogenous
loc_robot("r1", 0, ID) :- instance(ID), not loc_robot("r2", 0, ID).
loc_robot("r2", 0, ID) :- instance(ID), not loc_robot("r1", 0, ID).

loc_book("r1", 0, ID) :- instance(ID), not loc_book("r2", 0, ID).
loc_book("r2", 0, ID) :- instance(ID), not loc_book("r1", 0, ID).

hasBook("t", 0, ID) :- instance(ID), not hasBook("f", 0, ID).
hasBook("f", 0, ID) :- instance(ID), not hasBook("t", 0, ID).
```

```
goto(R, "t", I, ID) :- room(R), astep(I), instance(ID), not goto(R, "f", I, ID).
goto(R, "f", I, ID) :- room(R), astep(I), instance(ID), not goto(R, "t", I, ID).

pickup_book("t", I, ID) :- astep(I), instance(ID), not pickup_book("f", I, ID).
pickup_book("f", I, ID) :- astep(I), instance(ID), not pickup_book("t", I, ID).

putdown_book("t", I, ID) :- astep(I), instance(ID), not putdown_book("f", I, ID).
putdown_book("f", I, ID) :- astep(I), instance(ID), not putdown_book("t", I, ID).
```

We provide a list of transitions as the observed data, encoded as follows:

```
:- not loc_robot("r1", 0, 1).
:- not loc_book("r2", 0, 1).
:- not hasBook("f", 0, 1).
:- not goto("r2", "t", 0, 1).
:- not loc_robot("r1", 1, 1).

:- not loc_robot("r1", 0, 2).
:- not loc_book("r2", 0, 2).
:- not hasBook("f", 0, 2).
:- not goto("r2", "t", 0, 2).
:- not loc_robot("r2", 1, 2).

:- not loc_robot("r1", 0, 3).
:- not loc_book("r2", 0, 3).
:- not hasBook("f", 0, 3).
:- not goto("r2", "t", 0, 3).
:- not loc_robot("r2", 1, 3).

:- not loc_robot("r1", 0, 4).
:- not loc_book("r2", 0, 4).
:- not hasBook("f", 0, 4).
:- not goto("r2", "t", 0, 4).
:- not loc_robot("r2", 1, 4).

:- not loc_robot("r1", 0, 5).
:- not loc_book("r1", 0, 5).
:- not hasBook("f", 0, 5).
:- not pickup_book("t", 0, 5).
:- not hasBook("f", 1, 5).

:- not loc_robot("r1", 0, 6).
:- not loc_book("r1", 0, 6).
:- not hasBook("f", 0, 6).
:- not pickup_book("t", 0, 6).
:- not hasBook("f", 1, 6).
:- not loc_robot("r1", 0, 7).
:- not loc_book("r1", 0, 7).
:- not hasBook("f", 0, 7).
:- not pickup_book("t", 0, 7).
:- not hasBook("t", 1, 7).

:- not loc_robot("r1", 0, 8).
:- not loc_book("r1", 0, 8).
:- not hasBook("f", 0, 8).
:- not pickup_book("t", 0, 8).
:- not hasBook("t", 1, 8).

:- not loc_robot("r1", 0, 9).
:- not hasBook("t", 0, 9).
:- not hasBook("f", 1, 9).

:- not loc_robot("r1", 0, 10).
:- not hasBook("t", 0, 10).
:- not hasBook("t", 1, 10).

:- not loc_robot("r1", 0, 11).
:- not hasBook("t", 0, 11).
:- not hasBook("t", 1, 11).

:- not loc_robot("r1", 0, 12).
:- not hasBook("t", 0, 12).
:- not hasBook("t", 1, 12).
```

It describes 12 transitions, where `enter_failed` occurred 1 time out of 4 attempts, `pickup_failed` occurred 2 times out of 4 attempts, and `drop_book` occurred 1 time out of 4 attempts.

With the command

```
python code/learn-mcsat-em.py robot.lpmln robot-evid.txt
```

we execute gradient ascent based learning with MC-ASP sampling for 50 iterations, with 50 MC-ASP iterations each learning iteration. The output looks like

```
Rule 1:  0.536
Rule 2:  0.3752
Rule 3:  -0.5286
```

2.4 Yale Shooting Example

Consider a probabilistic extension of Yale shooting program, where shooting at the turkey does not necessarily kill the turkey. The extension can be modeled in LP$^{\text{MLN}}$ as follows, where the weight indicating how likely the turkey is killed if it's shot is to be learned:

```
#domain astep(AI).

astep(0..5).
step(0..6).
boolean("t";"f").

% Probability Distribution
@getWeight(1) pf1("t", AI) :- astep(AI).
@getWeight(2) pf1("f", AI) :- astep(AI).

% UEC
%% Fluents
:- not alive("t", I), not alive("f", I), step(I).
:- alive("t", I), alive("f", I), step(I).
:- not loaded("t", I), not loaded("f", I), step(I).
:- loaded("t", I), loaded("f", I), step(I).
%% Actions
:- not load("t", I), not load("f", I), astep(I).
:- load("t", I), load("f", I), step(I).
:- not fire("t", I), not fire("f", I), astep(I).
:- fire("t", I), fire("f", I), astep(I).
%% Probablistic Facts
:- not pf1("t", I), not pf1("f", I), astep(I).
:- pf1("t", I), pf1("f", I), astep(I).

% Effect of Actions
loaded("t", I+1) :- load("t", I), astep(I).
alive("f", I+1) :- fire("t", I), loaded("t", I), pf1("t", I).
loaded("f", I+1) :- fire("t", I), loaded("t", I).

% Frame Axioms
0{alive(B, I+1)} :- alive(B, I), astep(I), boolean(B).
0{loaded(B, I+1)} :- loaded(B, I), astep(I), boolean(B).

% Initial state and actions are exogenous
0{loaded(B, 0) : boolean(B)}.
0{alive(B, 0) : boolean(B)}.
0{load(B, I) : boolean(B)} :- astep(I).
0{fire(B, I) : boolean(B)} :- astep(I).
```

We provide the following stable model as training data, which describes a history where the person shot the turkey 3 times and only succeeds at the last attempt:

```
:- not alive("t", 0).
:- not loaded("f", 0).
:- not fire("f", 0).
:- not load("t", 0).
:- not fire("t", 1).
:- not alive("t", 2).
:- not fire("f", 2).
:- not load("t", 2).
:- not fire("t", 3).
:- not alive("t", 4).
:- not load("t", 4).
:- not fire("f", 4).
:- not fire("t", 5).
:- not alive("f", 6).
```

We execute

```
python code/learn-mcsat-em.py yale-shooting.lpmln yale-shooting-evid.txt
```

to run gradient ascent 50 iterations, each with 50 MC-ASP iterations. The output looks like

```
New weights:
Rule 1:  -1.46
Rule 2:  -0.54
max_diff 0.0
```

The learned weights indicate that the success rate of shooting is about $\frac{exp(-1.46)}{exp(-1.46)+exp(-0.54)} \approx$ 0.285.