LIFTCOVER Documentation

Release 1.0.0

Fabrizio Riguzzi

Jan 03, 2024

CONTENTS

CHAPTER ONE

INTRODUCTION

LIFTCOVER is a system for learning simple probabilistic logic programs with scalability in mind [GBA+23, NFR19].

CHAPTER TWO

PREDICATE REFERENCE

CHAPTER

THREE

INSTALLATION

LIFTCOVER is distributed as a pack of SWI-Prolog. To install it, use

```
$ swipl
?- pack_install(liftcover).
```

3.1 Requirements

It uses the packs

- lbfgs
- auc

They are installed automatically when installing pack *liftcover* or can be installed manually as follows

```
$ swipl
?- pack_install(lbfgs).
?- pack_install(auc).
```

Pack *lbfgs* is optional, if absent the versions of the algorithms that use *lbfgs* do not work but the other versions work.

Parameter learning can be performed using Python and the Python packages numpy, torch and cupy, either on CPU or GPU. If they are not installed, the algorithms that use them do not work, but you can still run the Prolog version of parameter learning.

You can upgrade the pack with

```
$ swipl
?- pack_upgrade(liftcover).
```

Note that the packs on which *liftcover* depends are not upgraded automatically. In this case, they need to be upgraded manually.

3.2 Example of use

```
$ cd <pack>/liftcover/prolog/examples
$ swipl
?- [muta].
?- induce_par_lift([1,2,3,4,5,6,7,8,9],P).
```

3.3 Testing the installation

```
$ swipl
?- [library(test_liftcover)].
?- test.
```

CHAPTER FOUR

SYNTAX

LIFTCOVER learns *liftable probabilistic logic programs* a restricted version of Logic programs with annotated disjunctions LPADs ([VV03, VVB04]). A HPLP is a set of annotated clauses whose head contain a single atom annotated with a probability. For the rest, the usual syntax of Prolog is used.

A clause in such a program has a single head with the *target predicate* (the predicate you want to learn) and a body composed of *input predicates* or predicates defined by deterministic clauses (typically facts).

A general HPLP clause has the following form:

```
h:p:- b1,b2...,bn.
```

The following example inspired from the UWCSE dataset used in [KD05] is represented as (file uwcse.pl)

```
advisedby(A,B):0.3 :-student(A),professor(B),project(A,C),project(B,C).
advisedby(A,B):0.6 :-student(A),professor(B),ta(C,A),taughtby(C,B).

student(harry). professor(ben).
project(harry,pr1). project(harry,pr2).
project(ben,pr1). project(ben,pr2).
taughtby(c1,ben). taughtby(c2,ben).
ta(c_1,harry). ta(c_2,harry).
```

where `publication(A,B,C)` means that A is a publication with author B produced in project C. `advisedby/2` is the target predicate and `student/1, professor/1, project/2, ta/2, taughtby/2` are input predicates.

The first clause states that a student A is advised by a professor B with probability 0.3 if they both work on the same project C.

The second states that a student A is advised by a professor B with probability 0.6 if the student is a teacher assistant in a course taught by the professor.

The facts in the program state that harry is a student and ben a professor. They have two joint courses c1 and c2, two joint projects pr1 and pr2.

CHAPTER FIVE

SEMANTICS

The semantics of liftable PLP directly inherits the semantics of LPADs.

CHAPTER

SIX

LEARNING

6.1 Input

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

- preamble
- background knowledge, i.e., knowledge valid for all interpretations
- liftable PLP for 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 [DRVL95]. bongard.pl and bongardkeys.pl represent a Bongard problem for LIFTCOVER.

6.1.1 Preamble

In the preamble, the LIFTCOVER library is loaded with (bongard.pl):

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

Now you can initialize with

```
:- lift.
```

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

```
:- set_lift(megaex_bottom,10).
:- set_lift(min_probability,0.00001).
:- set_lift(verbosity,1).
```

We will later see the list of available parameters.

6.1.2 Background and Initial hierarchical program

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

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

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

The annotations of the head atoms of the initial program can be probabilities, as in the example above. In parameter learning, the learning procedure can start with the initial parameters in the program. In this case, it is up to the user to ensure that there are values between 0 and 1. In the case of structure learning, the initial program is not necessary.

6.1.3 Language Bias

The language bias part contains the declarations of the input and output predicates. The output predicate is declared as

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

and indicates the predicate whose atom you want to predict (target predicate). Derivations for the atoms for this predicate in the input data is built by the system. Input predicates are those whose atoms you are not interested in predicting. You can declare input predicates with

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

For these predicates, the only true atoms are those in the interpretations and those derivable from them using the background knowledge.

Then, for structure learning 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.

```
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 of atoms are randomly chosen.

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 a clause must appear as output variables in an atom in 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.

An example of language bias for the Bongard domain is

```
output(pos/0).

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

LIFTCOVER also requires facts for the `determination/2` Aleph-style predicate 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`.

6.1.4 Example Interpretations

The last part of the file contains the data. You can specify data with two modalities: models and keys. In the models case, you specify an example model (or interpretation or mega-example) 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 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 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 atoms is the model.

Then you must indicate how 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 intentionally the folds as in

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

`fold/2` is dynamic so you can also write

```
:- fold(all,F),
sample_lift(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_lift(N,List,Sampled,Rest)` exported from `liftcover` 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.

6.2 Commands

6.2.1 Parameter Learning

To execute LIFTCOVER, prepare an input file as indicated above and call

```
?- induce_lift_par(+List_of_folds:list,-P:list).
```

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_lift_par([train],P).
```

The algorithm that is used for parameter learning is specified by the parameter `parameter_learning` that can be set to

- `em`, for Expectation Maximization, in Prolog
- `em_python`, for Expectation Maximization, in Python, using either `cpu` or `gpu`, depending on the value of the `processor` parameter. In the first case, *numpy* is used, in the latter case *cupy* is used.
- `gd`, for Gradient Descent, in Prolog
- `gd_python`, for Gradient Descent, in Python, using Pytorch, using either `cpu` or `gpu`, depending on the value of the `processor` parameter. In both cases, *torch* is used.
- `lbfgs` for Limited-memory Broyden-Fletcher-Goldfarb-Shanno, in Prolog and C

6.2.2 Structure Learning

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

```
?- induce_lift(+List_of_folds:list,-P:list).
```

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_lift([train],P).
```

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

6.2.3 Testing

A program can also be tested on a test set in LIFTCOVER with

```
test_lift(+Program:list,+List_of_folds:list,-LL:float,-AUCROC:float,-ROC:list,-
↪AUCPR:float,-PR:list) is det
```

where `Program` is a list of terms representing clauses and `List_of_folds` is a list of folds.

`test_lift/7` 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`.

Then you can draw the curves using C3.js as follows

```
compute_areas_diagrams(+ExampleList:list,-AUCROC:float,-ROC:dict,-AUCPR:float,-PR:dict)␣
↪is det
```

(from pack auc.pl) that takes as input a list `ExampleList` of pairs probability-literal.

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

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

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

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

In SWISH, by including

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

in the code before `:- lift.` the curves will be shown as graphs using C3.js and the output program will be pretty printed.

6.3 Hyper-parameters for Learning

Hyper-parameters are set with commands of the form

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

and read with commands of the form

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

- hyper-parameters
  - `parameter_learning`: (values: `{em,em_python,gd,gd_python,lbfgs}`, default value: `em`) parameter learning algorithm
  - `processor`: (values: `{cpu,gpu}`, default value: `cpu`) which processor Python will use for parameter learning
  - `regularization`: (values: `{no,l1,l2,bayes}`, default value: `l1`) type of regularization
  - `gamma` (values: real number, default value: `10`): regularization coefficient for L1 and L2
  - `ab` (values: list of two real numbers, default value: `[0,10]`): values of a and b for bayesian regularization
  - `eta` (values: real number, default value: `0.01`): eta parameter in gradient descent (the parameters are updated as par=par+eta*gradient)
  - `max_initial_weight` (values: real number , default value: 0.5): weights in lbfgs and gd are randomly initialized with values in the interval [-max_initial_weight, max_initial_weight].
  - `min_probability` (values: real number in `[0,1]`, default value: `1e-5`): probability threshold under which a clause is dropped out.
  - `eps` (values: real, default value: 0.0001): if the difference in the log likelihood in two successive parameter learning iterations is smaller than `eps`, then parameter learning stops.

- eps_f (values: real, default value: 0.00001): if the difference in the log likelihood in two successive parameter learning iterations is smaller than eps_f*(-current log likelihood), then LIFTCOVER stops.
- random_restarts_number (values: integer, default value: 1): number of random restarts of parameter learning algorithms
- random_restarts_number_str_learn (values: integer, default value: 1): number of random restarts during structure learning for learning the parameter of single clauses
- iter (values: integer, default value: -1): maximum number of parameter learning iterations (-1 means not limits)
- max_iter (values: integer, default value: 10): iterations of clause search.
- beamsize (values: integer, default value: 100): size of the beam in the search for clauses
- neg_ex (values: given, cw, default value: cw): if set to given, the negative examples in training and testing are taken from the test folds interpretations, i.e., those examples ex stored as neg(ex); if set to cw, the negative examples in training and testing 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, so the target predicates must have at least one fact for modeh/2 or modeb/2 also for parameter learning.
- specialization: (values: {bottom,mode}, default value: bottom) specialization mode.
- megaex_bottom (values: integer, default value: 1, valid for SLEAHP): number of mega-examples on which to build the bottom clauses.
- initial_clauses_per_megaex (values: integer, default value: 1, valid for SLEAHP): number of bottom clauses to build for each mega-example (or model or interpretation).
- d (values: integer, default value: 1, valid for SLEAHP): number of saturation steps when building the bottom clause.
- max_var (values: integer, default value: 4): maximum number of distinct variables in a clause
- maxdepth_var (values: integer, default value: 2): maximum depth of variables in clauses (as defined in [Coh95]).
- max_body_length (values: integer, default value: 100): maximum number of literals in the body of clauses
- max_clauses (values: integer, default value: 1000): maximum number of clauses in the theory
- neg_literals (values: {true,false}, default value: false): whether to consider negative literals when building the bottom clause
- minus_infinity: (values: real, default value: -1.0e20) minus infinity
- logzero (values: negative real, default value $\log(0.000001)$): value assigned to $\log(0)$.
- zero (values: real, default value 0.000001): value assigned to 0.
- seed (values: seed(integer) or seed(random), default value seed(3032)): seed for the Prolog random functions, see SWI-Prolog manual .
- verbosity (values: integer in [1,4], default value: 1): level of verbosity of the algorithms.

CHAPTER SEVEN

EXAMPLE FILES

The `pack/liftcover/prolog/examples` folder in SWI-Prolog home contains some example programs. The `pack/liftcover/docs` folder contains this manual in latex, html and pdf.

CHAPTER

EIGHT

MANUAL IN PDF

A PDF version of the manual is available at https://friguzzi.github.io/liftcover/_build/latex/liftcover.pdf.

CHAPTER NINE

LICENSE

phil follows the MIT License that you can find in phil root folder. The copyright is by Fabrizio Riguzzi and Arnaud Nguembang Fadja.

BIBLIOGRAPHY

[Coh95] William W. Cohen. Pac-learning non-recursive prolog clauses. *Artif. Intell.*, 79(1):1–38, 1995.

[DRVL95] 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, 80–94. Fukuoka, Japan, 1995. Springer.

[GBA+23] Elisabetta Gentili, Alice Bizzarri, Damiano Azzolini, Riccardo Zese, and Fabrizio Riguzzi. Regularization inâ probabilistic inductive logic programming. In Elena Bellodi, Francesca Alessandra Lisi, and Riccardo Zese, editors, *Inductive Logic Programming - ILP 2023*, volume 14363 of Lecture Notes in Artificial Intelligence, 16–29. Cham, 2023. Springer Nature Switzerland. doi:10.1007/978-3-031-49299-0_2.

[KD05] Stanley Kok and Pedro Domingos. Learning the structure of markov logic networks. In *Proceedings of the 22nd international conference on Machine learning*, 441–448. 2005.

[NFR19] Arnaud Nguembang Fadja and Fabrizio Riguzzi. Lifted discriminative learning of probabilistic logic programs. *Machine Learning*, 108(7):1111–1135, 2019. doi:10.1007/s10994-018-5750-0.

[VV03] J. Vennekens and S. Verbaeten. Logic programs with annotated disjunctions. Technical Report CW386, K. U. Leuven, 2003.

[VVB04] J. Vennekens, S. Verbaeten, and M. Bruynooghe. Logic programs with annotated disjunctions. In *International Conference on Logic Programming*, volume 3131 of LNCS, 195–209. Springer, 2004.