Pascal Documentation

Release 1.0

Riguzzi Fabrizio

Dec 23, 2021

CONTENTS

CHAPTER ONE

INTRODUCTION

Pasccal is an algorithm for learning probabilistic integrity constraints. It was proposed in [BARZ20]. It contains modules for both structure and parameter learning.

Pascal is also available in the cplint on SWISH web application at http://cplint.eu.

CHAPTER

TWO

INSTALLATION

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

```
.. code:: prolog
```

?- pack_install(pack).

2.1 Requirements

It requires the pack

- lbfgs

It is installed automatically when installing pack *pascal* or can be installed manually as

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

lbfgs uses a foreign library and contains the library binaries for 32 and 64 bits Linux. If you want to recompile the foreign library you can use

```
?- pack_rebuild(lbfgs).
```

On 32 and 64 bits Linux this should work out of the box.

You can upgrade the pack with

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

Note that the pack on which *pascal* depends is not upgraded automatically in this case so it needs to be upgraded manually.

2.2 Example of use

```
$ cd <pack>/pascal/prolog/examples
$ swipl
?- [bongardkeys].
?- induce_pascal([train]),T).
```

2.3 Testing the installation

```
$ swipl
?- [library(test_pascal)].
?- test_pascal.
```

2.4 Datasets

Other machine learning datasets are available in pack cplint_datasets.

2.5 Support

Use the Google group https://groups.google.com/forum/#!forum/cplint.

CHAPTER

THREE

LANGUAGE

A Probabilistic Constraint Logic Theory (PCLT) is a set of Probabilistic Integrity Constraints (PIC) of the form

$$p\ ::\ L_1, \ldots, L_b \rightarrow \exists(P_1); \ldots; \exists(P_n); \forall\neg(N_1); \ldots; \forall\neg(N_m)$$

where p is a probability, each L_i is a literal and each P_j and N_j is a conjunction of literals. We call each P_j a *P conjunction* and each N_k an *N conjunction*. We call each $\exists(P_j)$ a *P disjunct* and each $\forall\neg(N_k)$ an *N disjunct*.

The variables that occur in the body are quantified universally with scope the PIC. The variables in the head that do not occur in the body are quantified existentially if they occur in a P disjunct and universally if they occur in an N disjunct, with scope the disjunct they occur in.

An example of a PIC for the Bongard problems of [DRVL95]

$$0.5\ ::\ triangle(T), square(S), in(T,S) \rightarrow \exists(circle(C), in(C,S)); \forall\neg(circle(C), in(C,T))$$

which states that if there is a triangle inside a square then either there exists a circle inside the square or there doesn't exist a circle inside the triangle. This constraint has probability 0.5.

CHAPTER

FOUR

USE

The following learning algorithms are available:

- Parameter learning
- Structure learning

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
- PCLT 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 [DRVL95]. bongardkeys.pl represents a Bongard problem for Pascal.

4.1.1 Preamble

In the preamble, the Pascal library is loaded with (bongardkeys.pl):

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

Now you can initialize pascal with

```
:- pascal.
```

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

```
:-set_pascal(examples,keys(pos)).
:-set_pascal(learning_algorithm,gradient_descent).
:-set_pascal(learning_rate,fixed(0.5)).
:-set_pascal(verbosity,1).
```

We will see later the list of available parameters.

A parameter that is particularly important for Pascal 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 PCLT

Now you can specify the background knowledge by including a set of Prolog clauses in a section between `:- begin_bg.` and `:- end_bg.` For example

```
:- begin_bg.
in(A,B) :- inside(A,B).
in(A,D) :- inside(A,C),in(C,D).
:- end_bg.
```

Moreover, you can specify an initial PCLT in a section between `:- begin_in.` and `:- end_in.`. The initial program is used in parameter learning for providing the structure. In the section, facts for the predicates `rule/2` or `ic/1` can be given.

Facts for `rule/2` take the form `rule(ic,prob)` where `prob` is a probability and `ic` is a term of the form `head:-body`. In it, `body` is a list of literals and `head` is a list of head disjuncts. Each head disjunct is couple `(sign,conjunction)` where sign is either `(+)` for P disjuncts or `(-)` for N disjuncts and `conjunction` is a list of literals.

The *example of a PIC* above can be expressed as

```
:- begin_in.
rule(([((+),[circle(C),in(C,S)]),((-),[circle(C),in(C,T)])]:-
  [triangle(T),square(S),in(T,S)]),0.5).
:- end_in.
```

Facts for the predicate `ic/1` take the form `ic(string)` where `string` is a Prolog string wher a constraint is encoded as `prob::body--->head`. `body` is a conjuntion of literals where the conjunction symbol is `/\` . `head` is a disjunction where the disjunction symbol is `\/`. Each disjunct is either a conjunction of literals, in the case of a P disjunct, or of the form `not(conjunction)` where `conjunction` is a conjunction of literals.

The *example of a PIC* above can be expressed as

```
:- begin_in.
ic("0.5 :: triangle(T)/\\square(S)/\\in(T,S)
--->
circle(C)/\\in(C,S)
\\/
not(circle(C)/\\in(C,T)).").
:- end_in.
```

4.1.3 Language Bias

The language bias part is specified 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 considered. * stands for all those that are found. Otherwise the indicated number is 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 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.

An example of language bias for the Bongard domain is

```
modeh(*,triangle(+obj)).
modeh(*,square(+obj)).
modeh(*,circle(+obj)).
modeh(*,in(+obj,-obj)).
modeh(*,in(-obj,+obj)).
modeh(*,in(+obj,+obj)).
modeh(*,config(+obj,-#dir)).
modeb(*,triangle(-obj)).
modeb(*,square(-obj)).
modeb(*,circle(-obj)).
```

(continues on next page)

(continued from previous page)

```
modeb(*,in(+obj,-obj)).
modeb(*,in(-obj,+obj)).
modeb(*,config(+obj,-#dir)).
```

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 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.

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.

4.2 Commands

4.2.1 Parameter Learning

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

```
?- induce_par_pascal(<list of folds>,T).
```

where <list of folds> is a list of the folds for training and T will contain the input theory with updated parameters.

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

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

4.2.2 Structure Learning

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

```
induce(+List_of_folds:list,-T:list) is det
```

where List_of_folds is a list of the folds for training and T will contain the learned PCLT.

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

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

A PCLT can also be tested on a test set with test_pascal/7 or test_prob_pascal/6 as described below.

4.2.3 Testing

A PCLT can also be tested on a test set with

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

or

```
test_prob_pascal(+T:list,+List_of_folds:list,-NPos:int,-NNeg:int,-LL:float,-
↪ExampleList:list) is det
```

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

test_pascal/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.

test_prob_pascal/6 returns the log likelihood of the test examples in LL, the numbers of positive and negative examples in NPos and NNeg and the list ExampleList containing couples Prob-Ex where Ex is a for a a positive example and \+(a) for a a negative example and Prob is the probability of example a.

Then you can draw the curves in cplint on SWISH using C3.js using

```
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 of the form that is returned by test_prob_pascal/6.

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

```
.. code:: prolog
```

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

In cplint on SWISH, by including

```
.. code:: prolog
```

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

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

4.3 Parameters for Learning

Parameters are set with commands of the form

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

The available parameters are:

- `specialization`: (values: `{bottom,mode}`, default value: `bottom`, valid for SLIPCOVER) 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 and LEMUR): 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, SLIPCOVER and LEMUR): number of random restarts of parameter EM learning
- `random_restarts_REFnumber` (values: integer, default value: 1, valid for SLIPCOVER and LEMUR): number of random restarts of parameter EM learning for refinements
- `seed` (values: seed(integer) or seed(random), default value `seed(3032)`): seed for the Prolog random functions, see SWI-Prolog manual
- `c_seed` (values: unsigned integer, default value 21344)): seed for the C random functions
- `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 and LEMUR): 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
- `mcts_beamsize` (values: integer, default value: 3, valid for LEMUR): size of the Monte-Carlo tree search beam
- `mcts_visits` (values: integer, default value: +1e20, valid for LEMUR): maximum number of visits

- `max_iter_structure` (values: integer, default value: 10000, valid for SLIPCOVER): maximum number of theory search iterations
- `mcts_max_depth` (values: integer, default value: 8, valid for LEMUR): maximum depth of default policy search
- `mcts_c` (values: real, default value: 0.7, valid for LEMUR): value of parameter C in the computation of UCT
- `mcts_iter` (values: integer, default value: 20, valid for LEMUR): number of Monte-Carlo tree search iterations
- `mcts_maxrestarts` (values: integer, default value: 20, valid for LEMUR): maximum number of Monte-Carlo tree search restarts
- `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 `modebb/2` also for parameter learning.
- `alpha` (values: floating point ≥ 0, default value: 0): parameter of the symmetric Dirichlet distribution used to initialize the parameters. If it takes value 0, a truncated Dirichlet process is used to sample parameters: the probability of being true of each Boolean random variable used to represent multivalued random variables is sampled uniformly and independently in [0,1]. If it takes a value ≥ 0, the parameters are sampled from a symmetric Dirichlet distribution, i.e. a Dirichlet distribution with vector of parameters $\alpha, \ldots, \alpha$.
- `verbosity` (values: integer in [1,3], default value: 1): level of verbosity of the algorithms.

CHAPTER

FIVE

EXAMPLE FILES

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

CHAPTER SIX

MANUAL IN PDF

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

CHAPTER SEVEN

LICENSE

Pascal follows the BSD 2-Clause License that you can find in the root folder. The copyright is by Fabrizio Riguzzi.

CHAPTER

EIGHT

REFERENCES

BIBLIOGRAPHY

[BARZ20] Elena Bellodi, Marco Alberti, Fabrizio Riguzzi, and Riccardo Zese. MAP inference for probabilistic logic programming. *Theory and Practice of Logic Programming*, 20(5):641–655, 2020. URL: https://arxiv.org/abs/2008.01394, doi:10.1017/S1471068420000174.

[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.