Pascal Documentation

Release 1.0

Riguzzi Fabrizio

Dec 11, 2023

CONTENTS

CHAPTER

ONE

INTRODUCTION

Pasccal is an algorithm for learning probabilistic integrity constraints. It was proposed in [RBZ+21]. 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

PREDICATE REFERENCE

- pascal

CHAPTER

THREE

INSTALLATION

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

```
?- pack_install(pack).
```

3.1 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.

3.2 Example of use

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

3.3 Testing the installation

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

3.4 Support

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

CHAPTER

FOUR

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

FIVE

USE

The following learning algorithms are available:

- Parameter learning
- Structure learning

5.1 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.

5.1.1 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.

5.1.2 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.
```

5.1.3 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>

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

5.1.4 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.

In order to specify if an interpretation is positive or negative, you should include in the interpretation a fact. The form of the fact dependes on the parameter `examples` whose values are `{auto, keys(pred)}`. If set to `auto`, positive examples in the models format should contain a `pos` fact and in the keys format a `pos(id)` fact, where `id` is the identifier of the interpretation. If set to `keys(pred)`, `pred` or `pred(pos)` (`pred(id)` or `pred(id,pos)` in the keys format) is used instead of `pos` to determine positive exmples.

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).
```

which however must be inserted after the input interpretations otherwise the facts for `int/1` will not be available and fold `all` would be empty.

5.2 Commands

5.2.1 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).
```

5.2.2 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.

5.2.3 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.

5.3 Parameters for Learning

Parameters are set with commands of the form

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

The available parameters are:

- examples: (values: {auto, keys(pred)}, default value: auto) if set to auto, positive examples in the models format should contain a pos fact and in the keys format a pos(id) fact, where id is the identifier of the interpretation. If set to keys(pred), pred is used instead of pos to determine positive exmples
- beamsize (values: integer, default value: 10): size of the beam
- verbosity (values: integer in [1,3], default value: 1): level of verbosity of the algorithms.
- max_nodes (values: integer, defualt value: 10): maximum number of iteration of beam search
- optimal (values: {yes, no}, default value: no): whether ther refinement operator is optimal or not
- max_length (values: integer, defualt value: 4): maximum number of body literals and head disjuncts
- max_lengths (values: list of integers [Body,Disjuncts,LitIn+,LitIn-], defualt value: [1,1,1,0]): maximum number of, respectively, body literals, head disjuncts, literals in P disjuncts and literals in N disjuncts
- max_initial_weight (values: real, default value 0.1): absolute value of the maximum of the initial weights in weight learning.

- `learning_algorithm` (values: `{gradient_descent, lbfgs}`, default value: `gradient_descent`): algorithm for parameter learning
- `random_restarts_number` (values: integer, default value: 1): number of random restarts for gradient descent parameter learning
- `learning_rate` (values: `{fixed(value),decay(eta_0,eta_tau,tau)}`, default value: `fixed(0.01)`): value of the learning rate, either fixed to a value or set with a decay strategy
- `gd_iter` (values: integer, default value: 1000): maximum number of gradient descent iterations
- `epsilon` (values: real, default value: 0.0001): if the difference in the log likelihood in two successive parameter gradient descent iterations is smaller than `epsilon`, then the algorithm stops
- `epsilon_fraction` (values: real, default value: 0.00001): if the difference in the log likelihood in two successive parameter gradient descent iterations is smaller than `epsilon_fraction*(-current log likelihood)`, then the algorithm stops
- `regularization` (values: `{1,2}`, default value: 2): either L1 or L2 regularization in gradient descent and lbfgs
- `regularizing_constant` (values: real, default value: 5): value of the regularizatiom constant in gradient descent and lbfgs
- `max_rules` (values: integer, default value: 10): maximum number of PIC in the final theory
- `logzero` (values: negative real, default value $\log(0.01)$: value assigned to $\log(0)$
- `zero` (values: positive real, default value `0.0001`: value assigned to 0 when computing denominators that are counts
- `minus_infinity` (values: negative real, default value `-1.0e20`: value assigned to $-\infty$, used as the intial value of the log likelihood in parameter learning

CHAPTER

SIX

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 html and pdf.

CHAPTER

SEVEN

MANUAL IN PDF

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

CHAPTER

EIGHT

LICENSE

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

CHAPTER

NINE

REFERENCES

BIBLIOGRAPHY

[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.

[RBZ+21] Fabrizio Riguzzi, Elena Bellodi, Riccardo Zese, Marco Alberti, and Evelina Lamma. Probabilistic inductive constraint logic. *Machine Learning*, 110:723–754, 2021. doi:10.1007/s10994-020-05911-6.