phil Documentation

Release 1.0.0

Arnaud Nguembang Fadja, Fabrizio Riguzzi

Mar 12, 2020

CONTENTS

CHAPTER

ONE

INTRODUCTION

phil is a suite of programs for reasoning with hierarchical probabilistic logic programs (HPLP) *[NguembangFadjaLR17]*. It allows both inference and learning. phil is available only for SWI-Prolog.

phil is part of cplint on SWISH web application available at http://cplint.eu.

CHAPTER

TWO

INSTALLATION

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

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

2.1 Requirements

It requires the packs

- auc
- matrix

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

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

You can upgrade the pack with

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

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

2.2 Example of use

```
$ cd <pack>/phil/prolog/examples
$ swipl
?- [uwcse].
?- inference_hplp(advisedby(harry,ben),ai,Prob).
```

2.3 Testing the installation

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

2.4 Datasets

Other machine learning datasets are available in pack phil_datasets.

CHAPTER

THREE

SYNTAX

phil allows the definition of discrete probability distributions. It performs inference, parameter and structure learning on Hierarchical Probabilistic Logic Programs (HPLPs) *[NguembangFadjaLR17]*.

HPLPs are restrictions 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 general HPLP clause has the following form:

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

where `BodyInput` is a conjunction of predicates called input predicates, that are defined by certain facts and rules, and the `bi` are predicates non explicitly defined in the data called hidden predicates. Each hidden predicate is described by a set of clauses in a level immediately bellow. e.g `b1` can be described as follows:

```
b1:p1_1:- BodyInput1, b1_11,b1_12...,b1_1m.
b1:p1_2:- BodyInput2, b1_21,b1_22...,b1_2m.
...

b1:p1_k:- BodyInputk, b1_k1,b1_k2...,b1_km.
```

Each `b1_ij` is also described by a set clauses in the layer immediately bellow. This creates an hierarchy among predicates and clauses. Programs having this structure are called hierarchical probabilistic logic programs. The predicate in the head of clauses in the first layer is called `target` or `output` predicate which is the predicate we are interested in predicting. The `p1_i` are numeric expressions representing probabilities. It is up to the user to ensure that their are legal, i.e. they are greater than zero and less than one. The first clause above states that if the conjunction of the input and the hidden predicates in the body is true, then the head is true is true with probability `p` otherwise, the head is false with probability `1-p`. Note that, hidden predicates in the body of clauses in the programs are all different. If the clause has an empty body, it can be represented like this:

```
h:p.
```

If the clause is annotated with probability 1, the annotation can be omitted and the clause takes the form of a normal prolog clause, i.e.:

```
h :- Body.
```

stands for:

```
h:1 :- Body.
```

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),hid_1(A,B,C).
advisedby(A,B):0.6 :-student(A),professor(B),ta(C,A),taughtby(C,B).
```

```
hid_1(A,B,C): 0.2 :- publication(D,A,C),publication(D,B,C).

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).
publication(p1, harry, pr1). publication(p2, harry, pr1).
publication(p3, harry, pr2). publication(p4, harry, pr2).
publication(p1, ben, pr1). publication(p2, ben, pr1).
publication(p3, ben, pr2). publication(p4, ben, pr2).
```

where publication(A,B,C) means that A is a publication with author B produced in project C. advisedby/2 is the target predicate, student/1, professor/1, project/2, ta/2, taughtby/2 and publication/3 are input predicates and hid_1/3 is a hidden predicate.

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 and the predicate hid_1(A,B,C) is true. 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 third clause is a hidden clause which acts as an aggregation of publications coauthored by the student and his/her professor from a certain project.

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, two joint publications p1 and p2 from project pr1 and two joint publications p3 and p4 from project pr2.

CHAPTER

FOUR

SEMANTICS

The semantics of HPLPs directly inherits the semantics of LPADs. In the case of programs without function symbols, the semantics of HPLPs is given as follows: an HPLP defines a probability distribution over normal logic programs called *worlds*. A world is obtained from the HPLP by first grounding it, and by including in the world or no each ground clause (without the probability annotation). The probability of a world is the product of the probabilities associated to the heads of each selected ground clause. 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 HPLP contains function symbols, the definition is more complex, see *[Poo97][SK01]*.

CHAPTER

FIVE

INFERENCE

phil answers queries using the module phil. It performs exact inference using a program transformation technique. Each program is converted to a set of arithmetic circuits (ACs) or deep neural networks using a modified version of PITA *[Rig14]* with tabling and answer subsumption *[RS10]* as described in *[NguembangFadjaLR17]*.

For answering queries, you have to prepare a Prolog file where you first load the inference module `phil`, initialize it with the directive `:- phil` and then enclose the hierarchical program clauses in `:-begin_in.` and `:-end_in.` or enclose the clauses in `in([<clauses>])`. For example, the uwcse program above can be stored in uwcse.pl for performing inference with `phil` as follows

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

:- begin_in.
advisedby(A,B):0.3 :-student(A),professor(B),project(A,C),project(B,C),hid_1(A,B,C).
advisedby(A,B):0.6 :-student(A),professor(B),ta(C,A),taught\_by(C,B).
hid_1(A,B,C): 0.2 :- publication(D,A,C),publication(D,B,C).
:- end_in.

begin(model(ai)).
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).
publication(p1, harry, pr1). publication(p2, harry, pr1).
publication(p3, harry, pr2). publication(p4, harry, pr2).
publication(p1, ben, pr1). publication(p2, ben, pr1).
publication(p3, ben, pr2). publication(p4, ben, pr2).
end(model(ai)).
```

You can also have (non-probabilistic) clauses outside `:-begin/end_in.` which can be enclosed in `bg(<list of terms representing clauses>).` These are considered as database clauses.

To run a query, you can simply load the Prolog file, for example uwcse.pl, as:

```
?- [uwcse].
```

or:

```
$ swipl uwcse.pl
```

and run the query:

```
?- inference_hplp(advisedby(harry,ben),ai,Prob).
```

where `advisedby(harry,ben)` is the atom for the target predicate, `ai` the interpretation or the model where the query is run and `Prob` stores the computed probability. For also generating the arithmetic circuit, run the query:

```
?- inference_hplp(advisedby(harry,ben),ai,Prob,Circuit).
```

where `Circuit` stores the arithmetic circuit as a Prolog term.

CHAPTER

SIX

LEARNING

The following learning algorithms are available:

- PHIL for Parameter learning for HIerarchical probabilistic Logic programs. Given an input HPLP, PHIL learns its parameters from data. Two versions of PHIL are implemented:
  - DPHIL (Deep PHIL) *[NFRL18b]* performs gradient descent over the ACs. Two regularizations of DPHIL are implemented: L1 and L2 regularizations as used in the deep learning community.
  - EMPHIL (Expectation Maximization PHIL) *[NFRL18a]* performs EM. Three regularizations of EMPHIL are implemented: besides the previous two regularizations, a regularization based on Bayesian update of the parameters is also implemented.
- SLEAHP for Structure LEArning of Hierarchical Probabilistic logic programming, learns both the structure and the parameters of HPLPs from data. SLEAHP performs structure learning by parameter learning. From the data, a set of bottoms clauses are generated. Then a tree of literals appearing in the bottom clauses is created. A very large HPLP with initial parameters (randomly chosen) is randomly generated from the tree. Finally, a regularized version of PHIL is performed on the initial HPLP and clauses with very small values of probability are dropped.

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
- hierarchical program for which you want to learn the parameters (optional for SLEAHP)
- 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 SLEAHP.

6.1.1 Preamble

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

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

Now you can initialize with

```
:- phil.
```

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

```
:- set_hplp(megaex_bottom,10).
:- set_hplp(rate,0.95).
:- set_hplp(max_layer,-1).
:- set_hplp(min_probability,0.00001).
:- set_sc(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. Note that the hidden predicates are not declared since there are automatically generated. The output predicate is declared as

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

and indicates the predicate whose atom you want to predict. 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. In hierarchical programs only the ouput and the hidden predicates 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)).
```

SLEAHP 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_hplp(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_hplp(N,List,Sampled,Rest)` exported from

phil 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 PHIL, prepare an input file as indicated above and call

```
?- induce_hplp_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_hplp_par([train],P).
```

6.2.2 Structure Learning

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

```
?- induce_hplp(+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_hplp([train],P).
```

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

Between two executions of induce_hplp/2 you should exit SWI-Prolog to have a clean database.

6.2.3 Testing

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

```
test_hplp(+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_hplp/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 in phil 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_hplp_par([train],P),
test_hplp(P,[test],LL,AUCROC,ROC,AUCPR,PR).
```

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

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

In `phil` on SWISH, by including

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

in the code before `:- phil.` 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_hplp(<parameter>,<value>).
```

They are available both for PHIL and SLEAHP:

- PHIL hyper-parameters
  - `algorithmType` (values: `{dphil,emphil}`, default value: `dphil`): defines the parameter learning algorithm.
  - `maxIter_phil` (values: integer, default value: 1000): maximum number of iteration of PHIL. If set to -1, no maximum number of iterations is imposed.
  - `epsilon_deep` (values: real, default value: 0.0001): if the difference in the log likelihood in two successive PHIL (DPHIL and EMPHIL) iteration is smaller than `epsilon_deep`, then PHIL stops.
  - `epsilon_deep_fraction` (values: real, default value: 0.00001): if the difference in the log likelihood in two successive PHIL iteration is smaller than `epsilon_deep_fraction*(-current log likelihood)`, then PHIL stops.
  - `seed` (values: seed(integer) or seed(random), default value `seed(3032)`): seed for the Prolog random functions, see SWI-Prolog manual .
  - `setSeed` (values: `{no,yes}`, default value: `no`): if `yes`, the value of `c_seed` is used as seed for randomly initializing the probabilities in PHIL otherwise the actual clock value is used.
  - `c_seed` (values: unsigned integer, default value 21344)): seed for the C random functions if `setSeed` is set to `yes`.
  - `useInitParams` (values: `{no,yes}`, default value: `no`): if `yes` the initial parameters during the learning are the ones indicated in the program. Otherwise, the initial parameters are generated randomly using the clock seed.
  - `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.
  - `max_initial_weight` (values: real number , default value: 0.5): weights in DPHIL are randomly initialized with values in the interval [-max_initial_weight, max_initial_weight].

  - `adam_params` (values: list of four values [learningRate, beta1, beta2, epsilon], default value: [0.001,0.9,0.999,1e-8]): provides Adam's hyper-parameters.
  - `batch_strategy` (values: `minibatch(size)`, `stoch_minibatch(size)`, `batch`, default value: `minibatch(10)`): defines the gradient descent strategy in DPHIL. If set to `minibatch(size)` the mini batch strategy with batch size `size` is used i.e at each iteration, `size` arithmetic circuits (ACs) are used to compute the gradients. If set to `batch` the whole set of ACs are used to compute the gradients. `stoch_minibatch(size)` is a modified version of mini batch in which at each iteration, `size` ACs are randomly selected for computing the gradients.
  - `saveStatistics` (values: `{no,yes}`, default value: `no`): if `yes`, the statistics (evolution of the likelihood and other values) during parameter learning are saved in a folder whose name is the value of `statistics_folder` hyper-parameter.
  - `statistics_folder` (values: string , default value: `statistics`): folder where to save the statistics if `saveStatistics` is set to `yes`.
  - `regularized` (values: `{no,yes}` , default value: `no`): if set to `yes` regularization is enabled.
  - `regularizationType` (values: integer in `[0,3]` , default value: `0`): `0` to disable regularization, `1`, `2` and `3` for L1, L2 and Bayesian regularization respectively.
  - `gamma` (values: real number, default value: `10`): regularization coefficient for L1 and L2. Is equal to the Dirichlet coefficient `a` for Bayesian regularization.
  - `gammaCount` (values:real number , default value: 0): Dirichlet coefficient `b` for Bayesian regularization and typically 0 for L1 and l2 regularization.

- SLEAHP hyper-parameters
  - `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.
  - `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.
  - `probability` (values:real number in `[0,1]` , default value: `1.0`): initial value which indicates the probability to go from the current layer to the next when generating the initial HPLP.
  - `rate` (values: real number in `[0,1]`, default value: `0.95`): at each layer during the tree generation, the probability to go deeper is equal to `probability=probability*rate`. The deeper the layer the lower is the probability to go deeper.
  - `min_probability` (values: real number in `[0,1]`, default value: `1e-5`): probability threshold under which a clause is dropped out.
  - `use_all_mega_examples` (values: `{no,yes}` , default value:`yes`): if set to `yes` all the the bottom clauses are used to create the tree otherwise only one of them is used (typically the first).
  - `saveHPLP` (values: `{no,yes}`, default value: `no`): if set to `yes` the initial and the learned hierarchical programs are saved in the current folder.

- `saveFile` (values: string, default value: "hplp"): file name where to save the initial hierarchical programs if `saveHPLP` is set to `yes`. If the file name is `hplp`, the learned program is saved in file `hplp_learned`.
- `max_layer` (values: integer, default value: -1): maximum layer of clauses. If set to -1, no maximum is imposed.
- `verbosity` (values: integer in `[1,3]`, default value: `1`): level of verbosity of the algorithms.

CHAPTER

SEVEN

EXAMPLE FILES

The `pack/phil/prolog/examples` folder in SWI-Prolog home contains some example programs. The subfolder `learning` contains some learning examples. The `pack/phil/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://arnaudfadja.github.io/phil/_build/latex/phil.pdf.

CHAPTER

NINE

LICENSE

phil follows the Artistic License 2.0 that you can find in phil root folder. The copyright is by Arnaud Nguembang Fadja.

CHAPTER

TEN

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.

[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.

[NFRL18a] Arnaud Nguembang Fadja, Fabrizio Riguzzi, and Evelina Lamma. Expectation maximization in deep probabilistic logic programming. In *International Conference of the Italian Association for Artificial Intelligence*, 293–306. Springer, 2018.

[NFRL18b] Arnaud Nguembang Fadja, Fabrizio Riguzzi, and Evelina Lamma. Learning the parameters of deep probabilistic logic programs. In Elena Bellodi and Tom Schrijvers, editors, *Probabilistic Logic Programming (PLP 2018)*, volume 2219 of CEUR Workshop Proceedings, 9–14. Aachen, Germany, 2018. Sun SITE Central Europe. URL: http://ceur-ws.org/Vol-2219/paper2.pdf.

[Poo97] David Poole. The independent choice logic for modelling multiple agents under uncertainty. *Artificial Intelligence*, 94(1-2):7–56, 1997.

[Rig14] Fabrizio Riguzzi. Speeding up inference for probabilistic logic programs. *CJ_J*, 57(3):347–363, 2014. doi:10.1093/comjnl/bxt096.

[RS10] 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), 162–171. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010. doi:10.4230/LIPIcs.ICLP.2010.162.

[SK01] Taisuke Sato and Yoshitaka Kameya. Parameter learning of logic programs for symbolic-statistical modeling. *J. Artif. Intell. Res.*, 15:391–454, 2001.

[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.

[NguembangFadjaLR17] Arnaud Nguembang Fadja, Evelina Lamma, and Fabrizio Riguzzi. Deep probabilistic logic programming. In Christian Theil Have and Riccardo Zese, editors, *PLP17_B*, volume 1916 of CEUR_S, 3–14. Sun SITE Central Europe, 2017.