Notes on the implementation of FAM

Nicos Angelopoulos

The Welcome Trust Sanger Institute, Hinxton, Cambridgeshire, UK
nicos.angelopoulos@sanger.ac.uk

Abstract. We revisit the FAM algorithm with the aim of clarifying the inner working of the algorithm with respect to implementing it. We provide a step through the original presentation, clarifying issues that are of interest to implementors of the specific algorithm as well as to implementors of other EM algorithms in PLP frameworks. In addition, this paper presents *P*epl, an implementation of the algorithm in Prolog. We describe three different alternatives to dealing with the expressions needed at the core of the algorithm. A number of example programs provided with *P*epl are explained and the application of *P*epl to user specified programs is discussed.

1 Introduction

Failure adjusted maximization (*F*AM) [4] is an expectation-maxi-mization (*EM*) algorithm originally proposed in the context of stochastic logic programs (*S*LPs), [7,8]. As the name suggests, *F*AM extends the *EM* framework to account for failed derivation paths in *S*LPs [4]. The algorithm provides a closed-form formulation for computing the parameter weights within *EM*'s iterative maximization approach. It has been shown to be applicable to normalised *S*LPs, [4], which is a wide class of stochastic programs. The principle of adjusting for failure as introduced in *F*AM, has also been applied to the PRISM system [9].

*P*epl, which stands for parameter estimation in Prolog, is an implementation of the failure adjusted maximisation algorithm for *S*LPs. *S*LPs are an extension of logic programs where arithmetic labels are attached to clausal definitions. *S*LPs have well defined log linear semantics and an extensive analysis of the effect of backtracking strategies on these semantics, [2,3].

The original formulation of *F*AM although presented in a clear and concise mathematical language can be challenging for implementors, particularly for programmers who are new to probabilistic logic programming. Here, we expose some of the intricacies of the algorithm with emphasis on practical challenges in implementing it.

Furthermore, we present a specific implementation of *F*AM, *P*epl, which has been implemented in Prolog. Key aspects of the implementation such as the transformation of labelled clauses to Prolog and alternative methods for computing the counts used in the closed-form calculation, are also described. The implementation is tested against the examples from the *F*AM paper as well as on other probabilistic programs.

The remainder of the paper is structured as follows: Section 2 presents the algorithm with emphasis on implementation details, Section 3 describes *P*epl and Section 4 discusses a number of examples. Section 5 concludes the paper.

2 Programming *F*AM

Here we present some reading notes for Failure Adjusted Maximisation (FAM) algorithm on Stochastic Logic Programs as presented in [4]. The emphasis is in making some of the intricacies more apparent with respect to implementing the algorithm. The reader is expected to be familiar with Logic Programming jargon and have some appreciation of the SLPs syntax. Here we briefly introduce some basic SLP terminology.

SLPs are logic programs which include clauses that are labelled with arithmetic values. Predicates are defined as either logical, or as labelled in which case all the clauses in the predicate's definition should be labelled. The following two examples show how an unbiased coin (left) and a recursive predicate (right) can be coded.

```
1/2 : coin(head).          1/3 : member( H, [H|T] ).
1/2 : coin(tail).          2/3 : member( El, [H|T] ) :-
                                      member( El, T ).
```

A *pure* SLP is stochastic logic program that does not contain non-labelled predicates. In contrast, an *impure* program contains, and potentially has call dependencies between, labelled and non-labelled predicates. A normalised predicate is a predicate defined by labelled clauses for which the sum of the labels sum to one. FAM is a parameter estimation algorithm that learns the best values of the clausal labels from a data.

We prepend objects in the original FAM paper by *ML-*. Also, we use a top-down approach in presenting the algorithm. Thus we start with the definition in *ML-Definition 10* repeated here in Figure 1. (Slightly modified from the original.)

The objective of this algorithm is, when given the SLP program ($\mathcal{S}$) in Fig. 2 (*ML-Fig.9*) and data (y) Fig. 3 (part of *ML-Table II*) to produce the best set of $\lambda_1, \ldots \lambda_6$ (where $\lambda_i = (logl_i)$).

λ_i is the label, or parameter of clause C_i and a positive number (here we mainly consider $0 < \lambda_i \leq 1$).

Intuitively speaking, we want to find $\hat{\lambda} = \langle \hat{\lambda_1}, \ldots, \hat{\lambda_6} \rangle$ which will reproduce the data according to the input frequencies. For example, if we run goal $|? - s(A, B)$ 120 times, and against the program in Fig. 2 (its parameters changed to $\hat{\lambda}$) then we would like the results to have frequencies close to the ones in Table 1.

Assuming of course that we replace the top-to-bottom exhaustive backtrackable clause matching of Prolog by a proportional to label, non-backtrackable, choice amongst the matching clauses.

0. Let h = 0, and $\lambda^{(0)}$ such that $Z_{\lambda^{(0)}} > 0$.
1. For each parameterised clause C_i, compute $\psi_{\lambda^{(h)}}[\nu_i \mid y]$ $\psi_{\lambda^{(h)}}[\nu_i \mid y]$ using 1 (*ML-Eq.8*).
2. For each parameterised clause C_i let $S_i^{(h)}$ be the sum of the expected counts $\psi_\lambda^{(h)}[\nu_{i'} \mid y]$ for all the clauses $C_{i'}^+$ such that $C_{i'}^+$ shares the predicate symbol as C_i.
3. For each parameterised clause C_i, if $S_i^{(h)} = 0$ then $l_i^{h+1} = l_i^{(h)}$ otherwise

$$l_i^{(h+1)} = \frac{\psi_\lambda^{(h)}[\nu_i \mid y]}{S_i^{(h)}}$$

4. Set $h \leftarrow h + 1$ and go to 1 unless $\lambda(h + 1)$ has converged.

Fig. 1. The FAM Algorithm.

l_1 : s(X,p) :- p(X), p(X).	l_3 : p(a).	l_5 : q(a).
l_2 : s(X,q) :- q(X).	l_4 : p(b).	l_6 : q(b).

Fig. 2. The Example *S*LP $\mathcal{S}$.

2.1 Expectations

FAM is an EM algorithm. An EM algorithm tries to maximise the *likelihood of the data* by selecting the appropriate parameters. In SLPs the probability we try to maximise is $P(y|\lambda^{(h)})$. This is, the probability of the observed data (in the given frequencies) given the current parameters.

Since an SLP's parameters are its clausal probabilities, FAM works on the expected *contribution* a particular clause has in derivations, with relation to the data at hand. This is $\psi_{\lambda^{(h)}}[\nu_i \mid y]$ and was decomposed in *ML-Eq.8* (here [1] in Fig. 1)

$$\psi_{\lambda^{(h)}}[\nu_i \mid y] = \sum_{k=1}^{t-1} N_k \psi_{\lambda^{(h)}}[\nu_i \mid y_k] + N(Z_{\lambda^{(h)}}^{-1} - 1)\psi_{\lambda^{(h)}}[\nu_i \mid fail] \qquad (1)$$

The first part corresponds to refutations while the second term to failed derivations. Broadly speaking Eq. 1 gathers together the contributions of a par-

[1] the first part of the LHS of (1) has been removed here, since it was never used in the implementation, where we view unambiguous yields as a standard sub-case of the ambiguous case.

	k	1	2	3	4
Atom	y_k	s(a,p)	s(b,p)	s(a,q)	s(b,q)
Count	N_k	4	2	3	3

Fig. 3. The Example Data.

Atom	s(a,p)	s(b,p)	s(a,q)	s(b,q)
Should appear	40	20	30	30

Table 1. Example Distribution for $|? - s(A, B)$.

ticular clause (C_i) to derivations against (a) the program, (b) the current parameters and (c) the data. All the constituents of Eq. 1 are listed in the Glossary. The most important components are (a) $Z_{\lambda^{(h)}}$ the probability of success, (b) $\psi_{\lambda^{(h)}}[\nu_i \mid y_k]$ the expected number of times C_i was used in refutations yielding y_k, and (c) $\psi_{\lambda^{(h)}}[\nu \mid fail]$ the expected contribution of the clause to failed derivations. Let R the set of refutation, with R_k the refutations producing y_k and F the set of failed derivations. Then (b) and (c) are :

$$\psi_{\lambda^{(h)}}[\nu_i \mid y_k] = \sum_{r \in R_k} \psi_{\lambda^{(h)}}(r) \cdot \nu_i(r) / \sum_{r \in R_k} \psi_{\lambda^{(h)}}(r)$$

$$\psi_{\lambda^{(h)}}[\nu_i \mid fail] = \sum_{f \in R_{fail}} \psi_{\lambda^{(h)}}(f) \cdot \nu_i(f) / \sum_{f \in R_{fail}} \psi_{\lambda^{(h)}}(f)$$

We note that

$$\sum_{r \in R_k} \psi_{\lambda^{(h)}}(r) = \psi_{\lambda^{(h)}}(y_k)$$

$$\sum_{f \in R_{fail}} \psi_{\lambda^{(h)}}(f) = 1 - Z$$

2.2 Sampling versus exact counting

There are two main methods for computing (a) and R, R_k and F. Firstly, with the *exact* method, and secondly with sampling.

Exact computations involve finding all derivations and calculating the various counts to exact figures. Probability of success in this case,

$$Z_{\lambda^{(h)}} = \sum_{r \in R} \psi(r) / \sum_{r \in R} \psi(r) + \sum_{f \in F} \psi(f)$$

Using this formulation gives us some extra millage, when it comes to impure SLPs, which mix labelled and unlabelled (non-probabilistic) clauses. Strictly speaking, *ML-Sect.3., Defn.6* states that derivations in same equivalence class should have the same yield. This ensures that $\sum_{r \in R} \psi(r) + \sum_{f \in F} \psi(f) = 1$. In such cases $\sum_{r \in R} \psi(r) / \sum_{r \in R} \psi(r) + \sum_{f \in F} \psi(f) = \sum_{r \in R} \psi(r)$ which is what would FAM normally use.

A problem arises when considering infinite computations. When there is at least one infinite branch then it is impossible to manipulate all derivations. One possible solution is to place a significance limit (ϵ) deeming any derivation with probability $< \epsilon$ as failure. In the absence of infinite branches, one might

be able to generate graphs representing all possible derivations, thus making substantial time savings as there will not be a need to revisit derivations at each iteration. Note that we expect it will be very hard to combine significance limit and derivation as graphs.

Sampling, on the other hand, estimates the various expectations, by means of running a number of independent trials and noting the results. The process is repeated at each iteration with the updated λ. [2] In this case,

$$Z_{\lambda^{(h)}} = \frac{\sum_{r \in R} P(r)}{\sum_{r \in R} P(r) + \sum_{f \in F} P(f)} \tag{2}$$

Similarly to the exact case, a significance limit can be set to deal with infinite computation. On the other hand, it is not likely that we can efficiently employ graphs.

With sampling it is more straight forward to use the bag of samples to calculate $\psi_{\lambda^{(h)}}[\nu_i \mid y_k]$. Let S_k be the set of all the samples yielding y_k, then

$$\psi_{\lambda^{(h)}}[\nu_i \mid y_k] = \frac{\sum_{s \in S_k} \nu_i(s)}{\mid S_k \mid}$$

Since for $s \in S_k$,

$$\frac{1}{\mid S_k \mid} = \psi_{\lambda^{(h)}}[s \mid y_k]$$

Note that sampling seems to only work for pure, normalised SLPs. In contrast by using Z of 2 exact counting seems to also cope with impure SLPs.

Finally, one can store expressions rather than re-proving each y_k at every iteration. Storing the expressions is an alternative backend, that provides the same values. In terms of the theory of FAM, there is no difference to when the values are provided by exact counting.

In particular, we can store the expressions for $\psi_{\lambda^{(h)}}[\nu_i \mid y], \psi_{\lambda^{(h)}}(fail)$, and $\sum_{r \in R} \psi(r)$. The objectives of such an approach are:

- immediate gains of not having to prove each y_k every time,
- to use either graph reduction techniques over the SLD-tree, or polynomial simplifications, to derived expressions, that are faster to compute.

2.3 Log Likelihood

The FAM algorithm of Fig. 1 terminates when there is no substantial progress to be had. This is the case when the log-likelihood of the data given the current parameters is improved less than some ϵ within two iterations. The log-likelihood is [3]

$$logL_\lambda(y) = \sum_k N_k log(\psi_\lambda(y_k \mid true))$$

[2] strictly speaking, in our sampling R, R_k and F are not sets but bug collections
[3] it is also worth considering $\psi_\lambda(y_k \mid y)$

In the case of sampling we have :

$$\psi_\lambda(y_k \mid true) = \mid R_k \mid / \mid R \mid$$

since sampling may not yield all y_k, the above will lead to $logL_\lambda(y) = -inf$ when $\mid R_k \mid = 0$. Thus in that case we let $\psi_\lambda(y_k \mid true) = 1/ \mid R \mid^2$

For the exact case

$$\psi_\lambda(y_k \mid true) = \frac{\sum_{r \in R_k} \psi_\lambda(r)}{Z_\lambda}$$

3 Pepl

*P*epl is an implementation of the *F*AM algorithm. It consists of a set of Prolog predicates that can be used on *S*LPs to learn the clausal parameters from data and do sampling from *S*LPs. It is available for two Prolog systems: Yap [1] and SWI-Prolog [10]. For the latter system, *P*epl is provided as a prepackaged library [4] that can be easily installed from within *SWI-Prolog*.

?- pack_install(pepl).
?- library(pepl).
A simple example is provided which can be ran with:

?- [main].
?- main.

This example runs five iterations on the example presented in [4] (sources in `slp/jc_ml_pe.slp`). The default is to use exact counting (equiv. *?- main_exact.*). To run the same example with sampling or stored expressions counting, use *?- main_sample.* and *?- main_store.* respectively.

Stochastic clauses are term expanded to standard Prolog ones. Unique identifiers and a path argument are added to the transformation of stochastic clauses. These are used to identify the path of each derivation. In addition failure paths are also recorded by term expansion techniques. The system provides three ways for computing the counts needed for the closed-form calculation: exact, sample and store. The first method is the straight forward approach where all solutions to the target goal are quarried at each iterative step. Sampling approximates the counts by only sampling from the target. The expressions associated with the exact computation can be stored as term structures of arithmetic expression that can be evaluated at each iteration with fresh instantiations of the labels. This trades space for speed, making the computation much faster by requiring larger amounts of memory.

[4] `http://www.swi-prolog.org/pack/list?p=pepl`

3.1 Available predicates

sload_pe(*SlpSource***)**,
Load an SLP to memory. If the source file has .slp as its file extension then this may be omitted. *P*epl looks for *SlpSource* in directories `.`, and `./slp/` In SWI-Prolog it also looks in `pack(pepl/slp/)`.

sls/0
Listing of the stochastic program currently in memory.

ssave(*FileName***)**
Save the stochastic program currently in memory to a file.

fam(*Options***)**
Run *F*AM on the program loaded in memory, with a number of options passed as a list that may include the following terms:

- *count*(*CountMeth*), *CountMeth* in {*`exact`*, `store`, `sample`};
- *times*(*Tms*), default is *Tms* = `1000` (only relevant with *CountMeth*=`sample`);
- *termin*(*TermList*), currently *TermList* knows about the following terms
  - *interactive*- ask user if another iteration should be run,
  - `iter(I)`- *I* is the number of iterations,
  - `prm_e`(ϵ_p)- parameter difference between iterations. If the change in each and all pararameters between two iterations is less than ϵ_p then the algorithm terminates due to convergence of the parameters
  - `ll_e`(ϵ_λ)- likelihood convergence limit;
- *goal*(*Goal*), the top goal, defaults to an version of the data predicate with all its arguments replaced by free variables;
- *pregoal*(*PreGoal*), a goal that is called only once, before experiments are run. The intuition is that *PreGoal* will partially instantiate *Goal*.
- *data*(*Data*), the data to use, overrides `datafile/1`. Data should be a list of *Yield*-*Times* pairs. (All *Yields* of *Goal* should be included in *Data*, even if that means some get *Times* = `0`.)
- *prior*(*Prior*), the distribution to replace the probability labels with. Default is that no prior is used, Prior=none and input parameters are used as given in Slp source file. System also knows about `uniform` and `random`. Any other distribution should come in Prolog source file named *Prior*.pl and define *Prior*/3 predicate. First argument is a list of ranges (Beg-End) for each stochastic predicate in source file. Second argument, is the list of actual probability labels in source file. Finally, third argument should be instantiated to the list of labels according to *Prior*.
- *datafile*(*DataFile*), the data file to use, default is SLP_data.pl. *DataFile* should have either a number of atomic formulae or a single formula of the form: *frequencies(Data)*.+
- *complement*(*Complement*), one of : `none` (with $PrbSc = PrbTrue$, the default), `success` (with $PrbSc = 1 - PrbFail$), or `quotient` (with $PrbSc = PrbTrue/(PrbTrue + PrbFail)$).

- *setrand*(*SetRand*), sets random seeds. *SetRand* = `true` sets the seeds to some random triplet while the default *SetRand* = `false`, does not set them. Any other value for *SetRand* is taken to be of the form `rand(S1,S2,S3)` as expected by system predicate random of the supported Prolog systems.
- *eps*(*Eps*), the depth Epsilon. Sets the probability limit under which *P*epl considers a path as a failed one.
- *write_iterations*(*Wrt*) indicates which set of parameters to output. Values for *Wrt* are: `all`, which is the default, `last`, and `none`.
- *write_ll*(*Bool*) takes a boolean argument, indicating where log-likelihoods should be printed or not. Default is `true`.
- *debug*(*Dbg*) should be set to on or off (later is the default). If on, various information about intermediate calculations will be printed.
- *return*(*RetOpts*), a list of return options, default is the empty list. The term *RetOpts* contain variables which will be instantiated to the appropriate values signified by the name of each corresponding term. Recognised are, `initial_pps/1` for the initial parameters, `final_pps/` for the final/learned parameters, `termin/1` for the terminating reason, `ll/1` for the last log-likelihood calculated, `iter/1` for the number of iterations performed, and `seeds/1` for the seeds used.
- *keep_pl*(*KeepBool*, if `true`, the temporary Prolog file that contains the translated SLP, is not deleted. Default is `false`.
- *exception*(*Handle*), identifies the action to be taken if an exception is raised while running *fam/1*. The default value for *Handle* is `rerun`. This means the same Fam call is executed repeatedly. Any other value for *Handle* will cause execution to abort after printing the exception raised.

switch_dbg(+Switch)

Switch debugging of *fam/1* to either `on` or `off`.

scall(Goal,Eps,Meth,Path,Succ,BrPrb)

This is a predicate for people interested in the internals, and should only be used by experienced users. The following are the arguments to this call:

- The vanilla Prolog *Goal* to call.
- The value of *Eps(ilon)* at which branches are to be considered as failures.
- The search *Meth(od)* to be used, i.e. `all` for all solutions or `sample` for a single solution.
- The *Path(s)* of the derivation(s).
- A flag indicating a *Succ(essful)* derivation or otherwise-*Succ* is bound to the atom `fail` if this was a failed derivation and remains unbound otherwise.
- *BrPrb* the branch probability of the derivation.

See predicate main_gen/1, in examples/main_scfg.pl for example usage.

all_paths(+SlpFile,+Call)
Display to standard output all derivation paths and plenty of information associated with calling stochastic goal *Call* on the Slp defined in *+SlpFile*.

3.2 Running PE on your own SLPs

Since FAM is an instance of the EM algorithm, initial values for the parameters must be supplied. Also, since FAM is only a parameter estimation algorithm, the structure of the SLP must be given. In our implementation the user composes an SLP with the appropriate structure and labels the clauses in this SLP with the initial values of the parameters. The first step in running FAM is to load this SLP. Suppose the SLP with the initial parameters were saved in the file `foo.slp`; this SLP is loaded using *sload_pe/1* (detailed description in Section 3.1):
?- [pepl].
...
?- sload_pe(foo).
yes
?-

To run the EM algorithm we need a target sample space, comprising from observables, and the expected number each observable should appear. Data should be represented by a Prolog file of atomic formulae or a single formula of the form : `frequencies( Freqs ).` where Freqs is a list of Datum-Times pairs. In the former case atomic formulae can be of arbitrary format but they should share common predicate name and arity. This is the same predicate name and arity for the top goal in the user defined SLP. The intuition is that each formula is a point sample in the target sample space to which we wish to fit the parameters of a given SLP. When *frequencies(Freqs)* is used, *Datum* should be as the formulae just described, while *Times* should be the times each *Datum* appears in the target sample space. The target sample space can be passed to *fam/1* either with the option `data_file/1`, with its argument pointing to a data file as described above, or with the option `data/1`, with its argument being a frequencies list (*Freqs*, above). The two different ways to pass the target sample space and the two formats of the data file option are shown in Table 2

Assume `foo_data.pl` is the Prolog file containing the training data. To run FAM with default settings, do:
?- fam([]).
(see Section 3.1 for how to change these settings). To save the *S*LP with the estimated parameters to a file `fitted_foo.slp`, just do:
?- ssave(fitted_foo).
File `fitted_foo.slp` is created in current directory.

To run *fam* without first loading the *S*LP to memory, call
?- fam([slp(foo)]).

data([s(a,p)-4,s(a,q)-3,s(b,p)-2,s(b,q)-3])
(a)

s(a,p). s(a,q).
s(b,p). s(b,q).
s(a,p). s(a,q). frequencies([s(a,p)-4,s(a,q)-3,s(b,p)-2,s(b,q)-3]).
s(b,p). s(b,q).
s(a,p). s(a,q).
s(b,q). s(a,p).

(b) (c)

Table 2. Three alternative ways to pass the target sample space to FAM. Top, (a), using the `data/1` option. Bottom left, (b), one format for datafiles in `data_file/1` option; order of terms is not important. Bottom right, (c), the alternative format, using a single term.

4 Examples

A number of pre-canned examples are included in the *P*epl sources within the directory `examples`. Standard runs of *F*AM are wrapped in simple calls that can be invoked from the Prolog prompt.

4.1 Palindrome context free grammar

0.3 :: s $\longrightarrow$ [a], s, [a].
0.2 :: s $\longrightarrow$ [b], s, [b].
0.1 :: s $\longrightarrow$ [a],[a].
0.4 :: s $\longrightarrow$ [b],[b].

Fig. 4. The palindromic grammar example.

A usual trick to check that the parameter estimation software works is to generate (sample) N data points from an slp according to some initial set of parameters, then change this set to some generic values (often setting these to a uniform distribution) and then proceed to guessing the original parameters. An example of how this can be done in this implementation is in `run/main_scfg.pl`

For Yap:

% cd examples
% yap
?- [main_csfg].
?- main.

For Swi:

% swipl
?- [pack('pepl/examples/main_csfg')].

This example also illustrates :

(a) a situation where failure ϵ is important.
(b) how to produce N samples (*main_gen/1*).

Again, default is `main_exact` and `main_store` with `main_sample` are also provided.

4.2 Bloodtype PRISM example

```
 bloodtype(a)  :- genotype(a,a).
 bloodtype(a)  :- genotype(a,o).
 bloodtype(a)  :- genotype(o,a).
 bloodtype(b)  :- genotype(b,b).
 bloodtype(b)  :- genotype(b,o).
 bloodtype(b)  :- genotype(o,b).
 bloodtype(o)  :- genotype(o,o).
bloodtype(ab)  :- genotype(a,b).
bloodtype(ab)  :- genotype(b,a).
genotype(X,Y) :- gene(X),gene(Y).

1/3 :: gene( a ).
1/3 :: gene( b ).
1/3 :: gene( o ).
```

Fig. 5. Blood type example.

The example from the Prism web-site[5] can also be seen in action. The corresponding *S*LP can be found in the `slp` directory of the distribution: `slp/prism_bt`.

% cd run
% prolog (where prolog is in {yap,swipl}).
?- [main_prism_bt].
?- main_exact.
or
?- main_store.
or
?- main_sample.
after each of the above *main* calls, you can test accuracy by
?- test(10000).

Frequency of ground successful goals should match the frequency of *Data*. Note that FAM, in theoretic terms, is not strictly speaking applicable to this example since it does not observe the equivalence class criterion. However, in practice, the results of the algorithm are correct.

[5] `http://sato-www.cs.titech.ac.jp/prism/overview-e.html`

5 Conclusions

We presented an extensive new reading of the FAM algorithm which may be of value to implementors of EM algorithms in PLP languages. In tandem we detailed the inner workings of an easy-to-install Prolog based implementation of the algorithm. *P*epl comes with a number of canned examples that are collected from the original FAM paper and other literature sources. Furthermore we detailed a novel approach to re-calculating the arithmetic expressions necessary in the estimation calculations. This is via storing variable versions of the expressions, copies of which are instantiated at each iteration. Future work on the system can include tabled inference which has been shown to be successful in other PLP systems, [5,6]. The system described here is available as open source from: `http://stoics.org.uk/~nicos/sware/` and as an SWI-Prolog package at `http://swi-prolog.org/pack/list?p=pepl`.

Acknowledgements

Dr James Cussens helped with many clarifications and endless iterations of explanations while working on Pepl.

Glossary

Logic Programming

derivation Sequence of goals produced by applying resolution steps on an initial goal G_0 and program P, ending on either *true* or *false*, p. 48.

refutation A derivation ending in *true.*, p. 48.

Indices

h	Scripts the algorithm's current iteration.
i	Index for clauses.
k	Index for data.

Symbols

C_i	The ith clause. For example C_4 in 2 is l_4 : p(b), p. 48.
F	Set (or list) of all failed derivations, p. 49.
N_k	The number of times datum k occured in the observed data. For example in Fig. 3 $N_2 = 2$.
N	The number of observed data ($= \sum_k N_k$). For example in Fig. 3 N = 12.
R_k	Set (or list) of all refutations, yielding y_k, p. 49.
R	Set (or list) of all refutations, p. 49.
Z	Probability of success.

$\nu_i(d)$	Times C_i appeared in derivation (d).
l_i	The parameter of the ith clause. Also referred to, as clausal probability or label.
λ_i	The log of the ith clause parameter.
λ	The set of the clause log parameters $(= \langle \lambda_1, \ldots, \ldots, \lambda_n \rangle)$., p. 50.
y	The observed data against which FAM tries to learn the *true* parameters., p. 47.
$x_{0\iota}$	The ιth unabiguous refutation.
$\psi_{\lambda^{(h)}}[\nu_i \mid y]$	Expected times C_i appears in R, p. 48.
ϵ	Significance level probability., p. 49.

References

1. Vítor Santos Costa, Ricardo Rocha, and Luís Damas. The YAP Prolog system. *Theory and Practice of Logic Programming*, 12:5–34, 1 2012.
2. James Cussens. Loglinear models for first-order probabilistic reasoning. In *UAI-99*, 1999.
3. James Cussens. Stochastic logic programs: Sampling, inference and applications. In *UAI'2000*, pages 115–122, 2000.
4. James Cussens. Parameter estimation in stochastic logic programs. *Machine Learning*, 44(3):245–271, 2001.
5. Yoshitaka Kameya, Taisuke Sato, and Neng-Fa Zhou. Yet more efficient EM learning for parameterized logic programs by inter-goal sharing. In *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22-27, 2004*, pages 490–494, 2004.
6. Angelika Kimmig, Vítor Santos Costa, Ricardo Rocha, Bart Demoen, and Luc De Raedt. On the efficient execution of problog programs. In *Logic Programming, 24th International Conference, ICLP 2008, Udine, Italy, December 9-13 2008, Proceedings*, pages 175–189, 2008. doi: 10.1007/978-3-540-89982-2_22. URL `http://dx.doi.org/10.1007/978-3-540-89982-2_22`.
7. Stephen Muggleton. Stochastic logic programs. In L. de Raedt, editor, *Advances in Inductinve Logic Programming*, pages 254–264. IOS Press, 1996.
8. Stephen Muggleton. Semantics and derivations for SLPs. In *Workshop on Fussion of Domain Knowledge with Data for Decision Support*, 2000.
9. Taisuke Sato, Yoshitaka Kameya, and Neng-Fa. Zhou. Generative modeling with failure in PRISM. In *IJCAI'2005*, page 847852, 2005.
10. Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012.