Pepl v.2
An implementation of the FAM algorithm.
User's Guide

Nicos Angelopoulos
Imperial College
nicos.angelopoulos@imperial.ac.uk

James Cussens
University of York
jc@cs.york.ac.uk

http://stoics.org.uk/~nicos/sware/pepl

January 26, 2014

Contents

1 Introduction

Pepl is an implementation of the failure adjusted maximisation (FAM) algorithm for Stochastic Logic Programs (Slps). The algorithm was introduced by James Cussens in [Cus01]. An incomplete programmer's view of the algorithm can be found in doc/pfam.pdf [Ang01]. FAM is a Expectation-Maximisation algorithm that allows one to estimate the probability labels of Slps from data.

Pepl comprises of a set of Prolog programs that can be used without compilation. The supported systems are:

Yap (tested on 6.3.4)
http://www.cos.ufrj.br/~vitor/Yap/

SWI Prolog (tested on 7.1.4)
http://www.swi-prolog.org

Older versions worked on:

SICStus (tested on 3.8.5) http://www.sics.se/isl/sicstus.html

In what follows we use the following terms:

Data the observations we want to learn from.

Label or parameter, the probability ascribed to a clause.

Sampling the non SLD, but label driven, derivation of a goal against an SLP.

Samples the observations of a sampling.

2 Installation and sanity check

In SWI you can install via the package manager (otherwise follow Yap instructions).
% swipl
?- pack_install(pepl).
?- [library(pepl)].
?- [pack('pepl/examples/main')].
?- main.

YAP instructions, download latest sources from http://stoics.org.uk/~nicos/sware/pepl
*% gunzip pepl-*tgz*
*% tar xf pepl-*tar*
*% cd pepl-**
% yap
?- [main].
?- main.

This will run five iterations on the example presented in Cussens2001 (sources in `slp/jc_ml_pe.slp`). The default is to use exact counting (equiv. *?- main_exact.*). To run the same example with sample or stored expressions counting, use *?- main_sample.* and *?- main_store.* respectively.

3 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 5):

?- [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 1

Assume, `foo_data.pl` is the Prolog file containing the training data. To run FAM with default settings, do:

?- fam([]).

(see Section 5 for how to change these settings). To save the SLP 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 Slp 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 1: 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 More canned examples

4.1 palindrome context free grammar

A usual trick to see if 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

The example from http://sato-www.cs.titech.ac.jp/prism/overview-e.html can also be seen in action. (Corresponding SLP can be found in `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 is not strictly speaking applicable to this example since it does not observe the equivalence class criterion of Section 6.3. Furthermore, since this is an impure SLP, sampling does not work properly.

5 Available predicates

sload_pe(*SlpSource***)**,

Load an SLP to memory. If the source file has an slp extension the extension may be omitted. Pepl looks for *SlpSource* in directories `.`, and `./slp/` In SWI it also looks in `pack(pepl/slp/)`.

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

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

fam(*Options***)** where its argument is a list that may include the following options:

- 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 iteration, that renders termination due to convergence of all parameters, between two iterations,
  - `ll_e`(ϵ_λ)- likelihood convergence limit;
- goal(*Goal*), the top goal, defaults to an all vars version of data predicate;
- pregoal(*PreGoal*), a goal that 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, 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 Pepl 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 loglikelihoods 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 terms RetOpts contain variables. These 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 loglikelihood 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. 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) Note: this is a predicate for people interested in the internals. Use at your own peril. The following describes 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 associated information associated with calling stochastic goal *Call* on the Slp defined in *+SlpFile*.

6 On FAM

6.1 normalisation

Since FAM works on normalised SLPs, Pepl automatically normalises the labels of all stochastic clauses, at read-in time.

6.2 stored expression

Currently we effect some straight forward simplifications such as :

- $0 + A \rightarrow A$,
- $1 * B \rightarrow B$,
- $A * B / A * C \rightarrow B / C$,

Our plans are to provide more interesting simplifications, either by considering graph operations on some representation of the SLD-tree, or reductions of the polynomial expressions derived from the tree.

6.3 equivalence class criterion

FAM works on equivalence classes of derivations rather than on instances. This allows us to deal with impure SLPs, since multiple proofs due to non-stochastic clauses are represented as a single probability point. This approach is applicable to cases where all members of the equivalence class have the same yield.

Since this is a severe restriction in Pepl we have relax this condition by allowing :

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

rather than

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

The two equations are equivalent when $\sum_{r \in R} \psi(r) + \sum_{f \in F} \psi(f) = 1$ which is the case for, equivalent class having same yield, programs.

Although initial experiments (e.g. blood type example 4.2) allows to be optimistic, there is no guarantee that FAM will perform over programs that do not observe the equivalence class criterion.

7 Web-page

The main page for Pepl is: http://stoics.org.uk/~nicos/sware/pepl/.

References

[Ang01] Nicos Angelopoulos. Programming FAM, May 2001. In doc/pfam.ps.

[Cus01] James Cussens. Parameter estimation in stochastic logic programs. *Machine Learning*, 2001. to appear.