Sampling and probabilistic inference in D/Slps.

nicos.angelopoulos@pirbright.ac.uk

Nicos Angelopoulos

The Pirbright Institute

Abstract. Stochastic logic programming (*Slp*) and Distributional logic programming (*Dlp*) are two closely related probabilistic logic programming formalisms that have been previously studied in the context of machine learning. The former is a restrictive form which has good parameter estimation results, while the latter is more expressive and has been used in the context of Bayesian machine learning. Although both formalisms have easily installed systems for performing the aforementioned machine learning tasks, these systems have previously no facilities for more general probabilistic inference over these probabilistic languages. Here we describe newly added facilities to both systems for sampling and probabilistic inference. We present unified ways to perform sampling, where SLD resolution is replaced by stochastic clause selection. Probabilistic inference where the probability of goals is calculated using standard SLD on augmented clauses. A number of simple and more complex examples are considered and for sampling we demonstrate stochastic properties by taking advantage a couple of auxiliary libraries that work well with the probabilistic packages considered here. initially, limitations of the expressivity of *Slps* are highlighted by juxtaposing it to the intuitive uniform membership program of *Dlps*.

1 Introduction

Stochastic logic programs were introduced to enable probabilistic reasoning within a labelled clausal context(Muggleton, 1996). They are similar in expressive power to Prism (Sato and Kameya, 1997; Sato et al., 2005) and have a robust algorithm (Failure Adjusted Maximisation, FAM) that can perform parameter estimation from data (Cussens, 2000). *Pepl* is an *SWI-Prolog* pack, that is, a user contributed library that is indexed in the central packs server and can be easily installed within *SWI-Prolog*. *Pepl*, (Angelopoulos, 2016) which implements FAM for *Slps*. Here, we describe new features added to the pack (version 2.3) for performing more general probabilistic inference. An *Slp* extends clausal logic programming by allowing ground arithmetic values as labels to clauses. An example of an *Slp* implementing a coin toss is shown in Figure 1.

Distributional logic programs (*Dlps*) were introduced to overcome limitations in the expressivity of *Slps* (Angelopoulos and Cussens, 2005, 2008). This category of probabilistic logic programs allows the probabilistic labels to be arbitrary arithmetic expressions which are calculated at run-time and typically depend

```
1 :: doubles(Side) :-
        coin(Side),
        coin(Side).

0.5 :: coin(head).
0.5 :: coin(tail).
```

Fig. 1. Programs for coin/1 and doubles/1. Syntax for both *Slp* and *Dlp* are identical for these simple examples.

on the input arguments to concretise the probability labels (Angelopoulos and Cussens, 2017). *Dlps* are a super class of *Slps* thus the *Slp* programs in Figure 1 are also valid *Dlp* syntax. *Bims* is an *SWI-Prolog* pack that implements Bayesian model averaging by means of a Metropolis-Hasting algorithm (Angelopoulos and Cussens, 2001, 2008). The system has been used in a number of publications both on theoretical aspects (Angelopoulos and Cussens, 2005, 2006) and applications (Angelopoulos et al., 2009).

Both *Pepl* and *Bims* compile probabilistic programs to normal *Prolog* clauses with a number of extra parameters and a small number of extra clauses. The extra parameters allow meta information such as the path to a resolution and the probability labels can be returned. This information can be then used in probabilistic inference tasks. The injected extra clauses allow unification and probability label calculation to take place. Probabilistic logic programming is a fertile area of research with a number of different formalisms having being introduced over the years (Riguzzi, 2023).

2 Resolution

As *Pepl* and *Bims* were developed with emphasis on machine learning (parameter estimation and model structure respectively), there have been limited general probabilistic inference over *Slps* and *Dlps*. Standard Prolog uses SLD (selective linear definite clause) resolution to refute goals against a logic program containing clauses. When labeling the clauses with probabilities we can extend resolution in two different distinct ways. First, we can use standard SLD derivations, while holding on to the probabilities of the resolution steps. In this scenario, each set of instantiations at the end of a refutation has an associated probability value assigned to it, which is simply the product of probability labels of the clauses used in the refutation. Furthermore, any specific instantiation will have a total probability ascribed to it, which the sum of the products for all the refutation that derive it.

Second, instead of SLD resolution, probabilistic labels can be used to affect stochastic sampling. Where at each resolution step, we do not choose the first matching clause, but from all matching clauses, we sample in proportion to the relative value of the probabilistic labels. We term this Selective Stochastic Definite clause resolution (SSD). So for instance if at a resolution step we have three matching clauses with labels crystallised to values 1/2, 1/4 and 1/4 then

SSD will select the first clause twice as often as it will select any of the other two clauses. In what follows, we show how these two types of enhancements are implemented in parallel ways for *Slps* and *Dlps* in the *Pepl* and *Bims* packs respectively.

We will use the two programs in Figure 1 to demonstrate the two type of resolution supported in both packs. The clauses for *coin/1* implement the simplest of probabilistic programs, the flipping of an unbiased coin. This program is extremely well behaved, as there no probability mass loss to failure. That is, there is no matching clause that will later lead to a failure. There are only two distinct instantiations to the query variable, each in a branch with probability of 0.5 with a total sum of 1. Program *doubles/1* throws a dice twice, with success only registering in the case where both coin tosses had the same result. Unlikely the *coin/1* predicate, this is less well behaved in terms of probabilistic inference, as there is a probability mass loss equal to a half.

2.1 Probability of goals with SLD

Both *Pepl* and *Bims* transform the probabilistic clauses to standard logic clauses that contain the probabilistic and path information in extra arguments. It is then a matter of injecting extra code that allows backtrackable exploration of the space. The path and label information can then be utilised to calculate the probability of a branch and the probability of specific instantiations.

Pepl supports the SLD-based exploration of *Slps* via predicates *scall/1*, *scall/2* and *scall/5*.

$$\%\ \textit{loads the Slp to memory}$$
$$?-\ \textit{sload\_pe}(\textit{coin}).$$
$$?-\ \textit{scall}(\textit{coin}(\textit{Flip}),\quad \textit{Prb}).$$
$$\textit{Flip}\ = \textit{head},$$
$$\textit{Prb}\ = 0.5\ ;$$
$$\textit{Flip}\ = \textit{tail},$$
$$\textit{Prb}\ = 0.5.$$

scall/5 provides the more generic way to interact with the probability of events:

$$\textit{scall}(+\textit{Goal},\ +\textit{Eps},\ -\textit{Path},\ -\textit{Succ},\ -\textit{Prb}).$$

where, *Eps* is the ϵ value. This provides a threshold, where paths whose probability is below that value are considered as failed refutations. A value of zero, means that all paths are explored, with no threshold based branch pruning being applied. The rationale is that if we are interested in the high probability derivations then we can safely ignore these paths while in the midst of resolution. *Eps* provides a convenient way of reducing the search space without losing high probability branches. *Path* is a list of the indices of clauses used in the derivation. The indices simply number the clauses in their position within the

program, from 1 upwards. *Prb* is the probability of the derivation branch, which is the product of all the probability labels for the clauses used in the resolutions steps. Finally, *Succ*, is either instantiated to *fail* if this is a failure branch and it is a free variable otherwise. The reason for accounting for failure is because *Pepl* implements the Failure Adjusted Maximisation algorithm which needs to know about which branches lead to failure, along with the probabilities of each of the branches. The benefit of having this information is that we can reconstruct the whole of the probability space if so required.

To find the probability of a *tail* coin flip, the following can be used, where 0 means that there will be no pruning of any branches.

$?-\ scall(coin(tail),0,Path,Succ,Prb).$
$Path\ = [2],$
$Succ\ = fail,$
$Prb\ = 0.5\ ;$
$Path\ = [3],$
$Prb\ = 0.5.$

To find the total probability of a goal in *Pepl*:

$?-\ sload\_pe(doubles).$

$?-\ scall\_sum(doubles(head),\ Prb).$
$Prb\ = 0.25.$

$?-\ scall\_sum(doubles(tail),\ \ Prb).$
$Prb\ = 0.25.$

$?-\ scall\_sum(doubles(Side),\ Prb).$
$Prb\ = 0.5.$

Bims provides similar facilities for deriving the probabilities of events using standard SLD resolution. However, unlike *Pepl*, here there is no ϵ threshold or explicit capture of failed branches.

$?-\ dlp\_load(doubles).$

$?-\ dlp\_call(doubles(Side)).$
$Side\ = head\ ;$
$Side\ = tail\ ;$
$false.$

$?-\ dlp\_call(doubles(Side),Path,Prb).$
$Side\ = head,$

Path = [3:1, 1/0.5, 1:0.5],
Prb = 0.25 ;
Side = *tail*,
Path = [3:1, 2/0.5, 2:0.5],
Prb = 0.25 ;
false.

?− *dlp_call_sum*(*doubles*(*Side*),*Sum*).
Sum = 0.5.

?− *dlp_call_sum*(*doubles*(*head*),*Sum*).
Sum = 0.25.

Note, that in the case of *Bims*, the *Path* is a list of term structures representing the derivation path. In the example above, (3:1) indicates the third clause was used, that there was no other clause matching (:) and that the probability of the clause was (1).

2.2 Sampling with SSD

Having enhanced definite clauses with arithmetic values that act as probability labels, we can use this labels to replace the standard SLD resolution. The intuitive operation is that the clauses are selected proportionally to the probability value of their label- namely, Selective Stochastic Definite clause resolution (SSD). Seeking a single branch at a time with such a strategy constitutes stochastic sampling which will in the long run sample from the SLD tree proportionally to the branch probabilities. This type of resolution can also be the basis for other algorithms. For instance, *Bims* implements a stochastic resolution and backtracking to enable model sampling in the context of Bayesian machine learning Angelopoulos and Cussens (2017).

In *Pepl* predicates *sample/1* and *sample/5* can be used for sampling.

% *sets the random seed*
?− *seed_pe*.

?− *sample*(*coin*(*tail*),0,*Path*,*Succ*,*Prb*).
Path = [1],
Succ = *fail*,
Prb = 0.5.

?− *seed_pe*.
?− *sample*(*coin*(*head*),0,*Path*,*Succ*,*Prb*).
Path = [1],
Prb = 0.5

?− *seed_pe*.
?− *sample*(*coin*(*Flip*),0,*Path*,*Succ*,*Prb*).
Flip = *head*,
Path = [1],
Prb = 0.5.

Please note that in the cases above where the instantiation of *Succ* is not shown it is because it is still an unbound variable (it only gets bound if it is a failure branch which is a internal signal for the resolution to stop). The *SWI-Prolog* interpreter does not show unbound variables in the set of instantiations. *Succ* is a free variable if the sample corresponds to a successful derivation while the bound to the atom $fail$.
Similar provisions have been recently added to *Bims* for sampling over *Dlps*. The following example sets the seed so the sequence produces repeatable results, before using SSD to sample *doubles/1* thrice.

?− *dlp_seed*.

?− *dlp_sample*(*doubles*(*Side*)).
Side = *tail*.

?− *dlp_sample*(*doubles*(*Side*)).
Side = *tail*.

?− *dlp_sample*(*doubles*(*Side*)).
Side = *head*.

2.3 Emerging sampling properties

Sampling using SSD will in the long run sample branches of the SLD tree according to the probability of the branches as this is calculated by multiplying together all the probability labels attached to the refutation clauses. To illustrate sampling properties of the programs presented thus far we will lean on some helper predicates from three other *SWI-Prolog* packs: *mlu* (machine learning utilities), *b_real* (Real helper predicates) and *Real* (a low level bridge to the *R* language (Angelopoulos et al., 2016)).

We sample 100 coin tosses using *Pepl* and *Bims* using the code below.

?− *lib*(*mlu*).
?− *seed_pe*.
?− *mlu_sample*(*sample*(*coin*(*Side*)), 100, *Side*, *Freqs*),
mlu_frequency_plot(*Freqs*, [*interface*(*barplot*),*outputs*([*svg*]),*las*=2]).
Freqs = [*head*−53, *tail*−47].

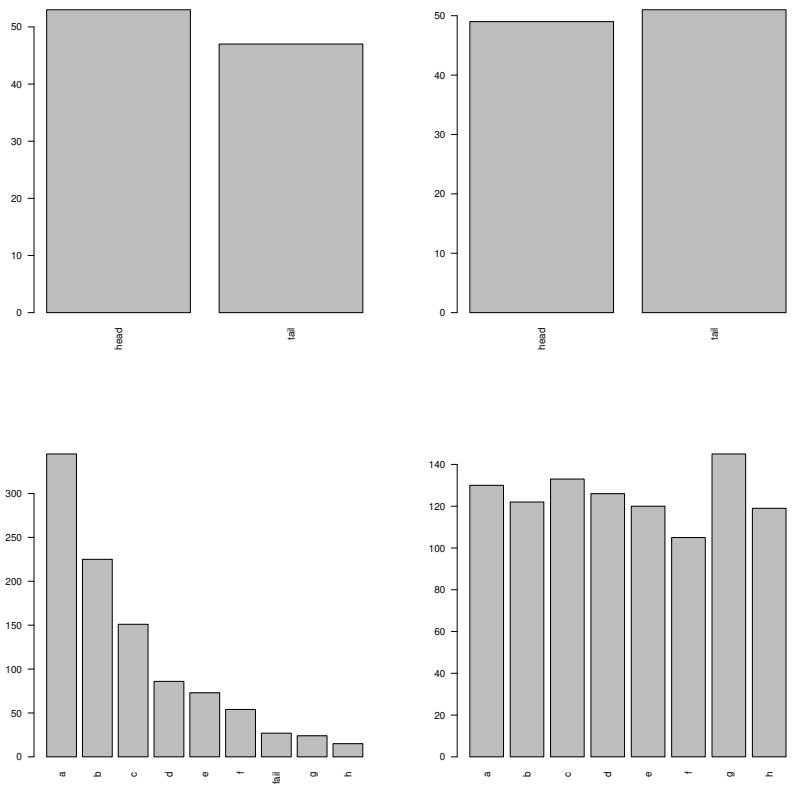

Fig. 2. Top left, sampling 100 coin tosses with *Pepl* over *Slps*. Top right, sampling 100 coin tosses with *Bims*. Bottom left, sampling 1000 draws for a single member from an eight member list using *Slp* predicate *member3/2*. Bottom right, sampling 1000 draws for a single member from an eight member list with *Dlp* predicate *umember/2*.

```
                                        :- pvars( umember(L,_E), [Len—length(L,Len)] ).
1/3 ::  member3( H, [H|T] ).
2/3 ::  member3( Elem, [_H|T] ) :-      1/X :: X ::  umember( [H|_T], H ).
            member3( Elem, T ).         (1 - 1/X) :: X ::  umember( [_H|T], El ) :-
                                                [X - 1] ::  umember( T, El ).
```

Fig. 3. Left, *Slp* which is tuned to cope with selecting uniformly from a 3 member list, but is unable to work on arbitrary length inputs. Right, *Dlp* draws uniformly an *Element* from *List*.

```
?- set_random(seed(1010)).
?- mlu_sample(dlp_sample(coin(Side)), 100, Side, Freqs),
   mlu_frequency_plot(Freqs,  [interface(barplot),outputs([svg]),las=2]).
Freqs  =[head—49, tail—51].
```

The top of Figure 2 includes the two plots generated via sampling and visualised through R as *svg* output plots. In both cases 100 coin tosses will produce a reasonably balanced set of *head* and *tail*.

3 Relative expressivity

Slps have similar expressivity to Prism and a similar focus in parameter estimation. The labels are thus specific arithmetic values that are unable to express complex probabilistic relations at the clausal level as they are unable to adapt to the variety of different data structures the relations hold over. *Slps* were the first modeling language used in *Bims*, but it soon became apparent that they were unable to cope with defining rich priors over complex statistical models such as classification trees and Bayesian networks which need to be held over compound *Prolog* terms. *Dlps* were then introduced to allow modelling of such complex relations (Angelopoulos and Cussens, 2005, 2008). The main difference is that labels are computed at run time and there is a succinct syntax that connects the labels to the relational data at that time.

The *Dlp* in Figure 3 draws uniformly an *Element* from *List*. The labels are calculated on-the-fly and depend on the length of the input list. Predicate *pvars/1* defines the guard part. At runtime the length of the input list L is computed into variable *Len* before it matched to clausal probabilistic variable X. The recursive call to *umember/2* passes the length of the tail (T) into the runtime environment so the length does not have to be re-calculated using the guard (*length/2* defined at top of the *Dlp* in Fig.3). A comprehensive discussion on the syntax and expressive power of *Dlps* can be found in Angelopoulos and Cussens (2017).

Unlike *Slps* the *Dlp* predicate *umember/2* succinctly captures the probabilistic intuition of uniform member selection that works on any input List. At the bottom part of Figure 2 we show the two plots generated by *Pepl* and *Bims* for the two membership programs of Figure 3 as generated by the following queries:

?− *lib*(*mlu*).
?− *sload_pe*(*member3*).
?− *seed_pe*.
?− *mlu_sample*(*scall*(*member3*(*X*,[*a*,*b*,*c*,*d*,*e*,*f*,*g*,*h*)), 1000, *X*, *Freqs*),
mlu_frequency_plot(*Freqs*, [*interface*(*barplot*)]).
Freqs = [*a*−345,*b*−225,*c*−151,*d*−86,*e*−73,*f*−54,*fail*−27,*g*−24,*h*−15].

?− *dlp_load*(*umember*).
?− *dlp_seed*.
?− *mlu_sample*(*dlp_sample*(*umember*([*a*,*b*,*c*,*d*,*e*,*f*,*g*,*h*],*X*)), 1000, *X*, *Freqs*),
mlu_frequency_plot(*Freqs*, [*interface*(*barplot*),*outputs*(*pdf*)),*las* = 2]).
Freqs = [*a*−130, *b*−122, *c*−133, *d*−126, *e*−120, *f*−105, *g*−145, *h*−119].

The bottom part of Figure 2 clearly shows that the *Slp* draws from a very skewed distribution far away from the intentional uniform draw.

4 Conclusions

We have enhanced two probabilistic logic programming packs with facilities to do high level sampling and probabilistic inference. We demonstrated stochastic aspects emerging from sampling across *Slps* and *Dlps*. The ability to perform such tasks are both beneficial both to the libraries but also enable researchers in probabilistic logic programming to become more familiar and easier to experiment with *Slps* and *Dlps*. There have been a number of different Probabilistic Logic Programming formalisms over the years and our work contributes to this ever enriching body of work (Riguzzi, 2023).

For future work we plan to add additional sampling for *Dlps* that account for failure similarly to the facilites for *Slps*. The reason why this is integral to *Pepl* is because it is central to the failure adjustment part of the FAM algorightm.

The software described here is available on github [1] [2] and as easily installed archived packs for the *SWI-Prolog* server [3]. This is also the case for the auxiliary packs we briefly used here to produce the sampling plots (packs mlu, b_real and Real).

[1] `https://github.com/nicos-angelopoulos/bims`
[2] `https://github.com/nicos-angelopoulos/pepl`
[3] `http://eu.swi-prolog.org/pack/list`

Bibliography

Nicos Angelopoulos. Notes on the implementation of FAM. In *3rd Probabilistic Logic Programming Workshop (collocated with ILP 2016)*, volume 1661, Imperial College, London, September 2016. CEUR. URL `http://ceur-ws.org/Vol-1661/`.

Nicos Angelopoulos and James Cussens. Markov chain Monte Carlo using tree-based priors on model structure. In Jack Breese and Daphne Koller, editors, *Uncertainty in Artificial Intelligence: Proceedings of the Seventeenth Conference (UAI-2001)*, pages 16–23, Seattle, USA, August 2001. Morgan Kaufmann. URL `https://stoics.org.uk/~nicos/pbs/uai01.ps.gz`.
CORE2014 Rank: A* "Artificial Intelligence and Image Processing".

Nicos Angelopoulos and James Cussens. Exploiting informative priors for Bayesian classification and regression trees. In *19th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 641–646, Edinburgh, UK, August 2005. URL `ftp://ftp.cs.york.ac.uk/pub/aig/Papers/James.cussens/ijcai05.pdf`.

Nicos Angelopoulos and James Cussens. Exploiting independence for branch operations in Bayesian learning of C&RTs. In Luc De Raedt, Thomas Dietterich, Lise Getoor, and Stephen H. Muggleton, editors, *Probabilistic, Logical and Relational Learning - Towards a Synthesis*, number 05051 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2006. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany. URL `http://drops.dagstuhl.de/opus/volltexte/2006/415`.

Nicos Angelopoulos and James Cussens. Bayesian learning of Bayesian networks with informative priors. *Journal of Annals of Mathematics and Artificial Intelligence*, 54(1-3):53–98, November 2008. URL `http://link.springer.com/article/10.1007/s10472-009-9133-x`.
SJR 2015: 0.593, SJR "Artificial Intelligence" quartile: Q2.

Nicos Angelopoulos and James Cussens. Distributional logic programming for Bayesian knowledge representation. *International Journal of Approximate Reasoning*, 80:52–66, January 2017. doi: http://dx.doi.org/10.1016/j.ijar.2016.08.004.

Nicos Angelopoulos, Andreas Hadjiprocopis, and Malcolm D. Walkinshaw. Bayesian ligand discovery from high dimensional descriptor data. *ACS Journal of Chemical Information and Modeling*, 49(6):1547–1557, 6 2009.

Nicos Angelopoulos, Samer Abdallah, and Georgios Giamas. Advances in integrative statistics for logic programming. *International Journal of Approximate Reasoning*, 78:103–115, November 2016. doi: http://dx.doi.org/10.1016/j.ijar.2016.06.008.

James Cussens. Stochastic logic programs: Sampling, inference and applications. In *Proceedings of the 16th Conference in Uncertainty in Artificial Intelligence (UAI)*, pages 115–122, Stanford University, Stanford, CA, USA, 2000. URL `ftp://ftp.cs.york.ac.uk/pub/ML_GROUP/Papers/uai00.ps.gz`.

Stephen H. Muggleton. Stochastic logic programs. In *Advances in Inductive Logic Programming*, pages 254–264. IOS Press, 1996. URL ftp://ftp.cs.york.ac.uk/pub/ML_GROUP/Papers/slp.ps.gz.

Fabrizio Riguzzi. *Foundations of Probabilistic Logic Programming: Languages, Semantics, Inference and Learning*. River Publishers Series in Software Engineering. River Publishers, 2nd edition, 2023. ISBN 9788770220651. URL https://books.google.co.uk/books?id=zk-MDwAAQBAJ.

Taisuke Sato and Yoshitaka Kameya. Prism: A symbolic-statistical modeling language. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1330–1335, 1997. URL http://sato-www.cs.titech.ac.jp/reference/IJCAI97.ps.gz.

Taisuke Sato, Yoshitaka Kameya, and Neng-Fa Zhou. Generative modeling with failure in prism. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 847–852, 2005.