Probabilistic Logical Inference On the Web

Marco Alberti[1], Giuseppe Cota[2], Fabrizio Riguzzi[1], and Riccardo Zese[2]

[1] Dipartimento di Matematica e Informatica – University of Ferrara
Via Saragat 1, I-44122, Ferrara, Italy
[2] Dipartimento di Ingegneria – University of Ferrara
Via Saragat 1, I-44122, Ferrara, Italy
{marco.alberti,giuseppe.cota,fabrizio.riguzzi,riccardo.zese}@unife.it

Abstract. `cplint` on SWISH is a web application for probabilistic logic programming. It allows users to perform inference and learning using just a web browser, with the computation performed on the server. In this paper we report on recent advances in the system, namely the inclusion of algorithms for computing conditional probabilities with exact, rejection sampling and Metropolis-Hasting methods. Moreover, the system now allows hybrid programs, i.e., programs where some of the random variables are continuous. To perform inference on such programs likelihood weighting is used that makes it possible to also have evidence on continuous variables. `cplint` on SWISH offers also the possibility of sampling arguments of goals, a kind of inference rarely considered but useful especially when the arguments are continuous variables. Finally, `cplint` on SWISH offers the possibility of graphing the results, for example by drawing the distribution of the sampled continuous arguments of goals.

Keywords: Probabilistic Logic Programming, Probabilistic Logical Inference, Hybrid program

1 Introduction

Probabilistic Programming (PP) [11] has recently emerged as a useful tool for building complex probabilistic models and for performing inference and learning on them. Probabilistic Logic Programming (PLP) [3] is PP based on Logic Programming that allows to model domains characterized by complex and uncertain relationships among domain entities.

Many systems have been proposed for reasoning with PLP. Even if they are freely available for download, using them usually requires a complex installation process and a steep learning curve. In order to mitigate these problems, we developed `cplint` on SWISH [14], a web application for reasoning on PLP with just a web browser: the algorithms run on a server and the users can post queries and see the results in their browser. The application is available at `http://cplint.lamping.unife.it` and Figure 1 shows its interface.

`cplint` on SWISH uses the reasoning algorithms included in the `cplint` suite, including exact and approximate inference and parameter and structure learning.
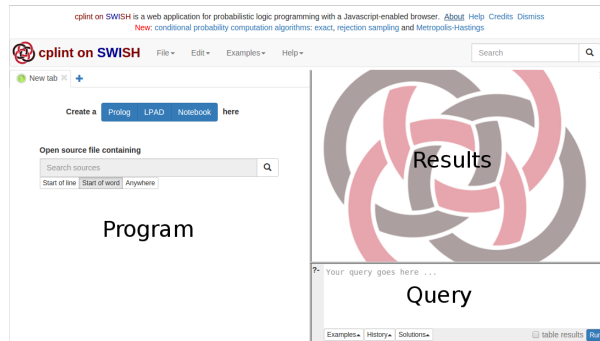
Fig. 1: Interface of `cplint` on SWISH.

In this paper we report on new advances implemented in the system. In particular, we included algorithms for computing conditional probabilities with exact, rejection sampling and Metropolis-Hasting methods. The system now also allows hybrid programs, where some of the random variables are continuous. Likelihood weighting is exploited in order to perform inference on hybrid programs and to collect evidences on continuous variables. When using such variables, the availability of techniques for sampling arguments of goals is extremely useful though it is infrequently considered. `cplint` on SWISH offers these features together with the possibility of graphing the results, for example by drawing the distribution of the sampled continuous arguments of goals. To the best of our knowledge, `cplint` on SWISH is the only online system able to cope with these problems.

A similar system is ProbLog2 [4], which has also an online version[3]. The main difference between `cplint` on SWISH and ProbLog2 is that the former currently also offers structure learning, approximate conditional inference through sampling and handling of continuous variables. Moreover, `cplint` on SWISH is based on SWISH[4], a web framework for Logic Programming using features and packages of SWI-Prolog and its Pengines library and uses a Prolog-only software stack, whereas ProbLog2 relies on several different technologies, including JavaScript, Python 3 and the DSHARP compiler. In particular, it writes intermediate files to disk in order to call external programs such as DSHARP, while we work in main memory only.

All the examples in the paper named as `<name>.pl` can be accessed online at `http://cplint.lamping.unife.it/example/inference/<name>.pl`.

2 Syntax and Semantics

The distribution semantics [17] is one of the most used approaches for representing probabilistic information in Logic Programming and it is at the basis

---

[3] `https://dtai.cs.kuleuven.be/problog/`

[4] `http://swish.swi-prolog.org`

of many languages, such as Independent Choice Logic, PRISM, Logic Programs with Annotated Disjunctions (LPADs) and ProbLog.

We consider first the discrete version of probabilistic logic programming languages. In this version, each atom is a Boolean random variable that can assume values true or false. The facts and rules of the program specify the dependences among the truth value of atoms and the main inference task is to compute the probability that a ground query is true, often conditioned on the truth of another ground goal, the evidence. All the languages following the distribution semantics allow the specification of alternatives either for facts and/or for clauses. We present here the syntax of LPADs because it is the most general [18].

An LPAD is a finite set of annotated disjunctive clauses of the form h_{i1} : $\Pi_{i1}; \ldots; h_{in_i}$: Π_{in_i} :- $b_{i1}, \ldots, b_{im_i}$. where $b_{i1}, \ldots, b_{im_i}$ are literals, $h_{i1}, \ldots h_{in_i}$ are atoms and $\Pi_{i1}, \ldots, \Pi_{in_i}$ are real numbers in the interval $[0, 1]$. This clause can be interpreted as "if $b_{i1}, \ldots, b_{im_i}$ is true, then h_{i1} is true with probability Π_{i1} or $\ldots$ or h_{in_i} is true with probability Π_{in_i}."

Given an LPAD P, the grounding $ground(P)$ is obtained by replacing variables with terms from the Herbrand universe in all possible ways. If P does not contain function symbols and P is finite, $ground(P)$ is finite as well.

$ground(P)$ is still an LPAD from which, by selecting a head atom for each ground clause, we can obtain a normal logic program, called "world", to which we can assign a probability by multiplying the probabilities of all the head atoms chosen. In this way we get a probability distribution over worlds from which we can define a probability distribution over the truth values of a ground atom: the probability of an atom q being true is the sum of the probabilities of the worlds where q is true, that can be checked because the worlds are normal programs that we assume have a two-valued well-founded model.

This semantics can be given also a sampling interpretation: the probability of a query q is the fraction of worlds, sampled from the distribution over worlds, where q is true. To sample from the distribution over worlds, you simply randomly select a head atom for each clause according to the probabilistic annotations. Note that you don't even need to sample a complete world: if the samples you have taken ensure the truth value of q is determined, you don't need to sample more clauses.

To compute the conditional probability $P(q|e)$ of a query q given evidence e, you can use the definition of conditional probability, $P(q|e) = P(q, e)/P(e)$, and compute first the probability of q, e (the sum of probabilities of worlds where both q and e are true) and the probability of e and then divide the two.

If the program P contains function symbols, a more complex definition of the semantics is necessary, because $ground(P)$ is infinite, a world would be obtained by making an infinite number of choices and so its probability, the product of infinite numbers all smaller than one, would be 0. In this case you have to work with sets of worlds and use Kolmogorov's definition of probability space [13].

Up to now we have considered only discrete random variables and discrete probability distributions. How can we consider continuous random variables and probability density functions, for example real variables following a Gaussian

distribution? `cplint` allows the specification of density functions over arguments of atoms in the head of rules. For example, in

```
g(X,Y): gaussian(Y,0,1):- object(X).
```

`X` takes terms while `Y` takes real numbers as values. The clause states that, for each `X` such that `object(X)` is true, the values of `Y` such that `g(X,Y)` is true follow a Gaussian distribution with mean 0 and variance 1. You can think of an atom such as `g(a,Y)` as an encoding of a continuous random variable associated to term `g(a)`. A semantics to such programs was given independently in [5] and [6]. In [10] the semantics of these programs, called Hybrid Probabilistic Logic Programs (HPLP), is defined by means of a stochastic generalization STp of the Tp operator that applies to continuous variables the sampling interpretation of the distribution semantics: STp is applied to interpretations that contain ground atoms (as in standard logic programming) and terms of the form $t = v$ where t is a term indicating a continuous random variable and v is a real number. If the body of a clause is true in an interpretation I, $STp(I)$ will contain a sample from the head.

In [6] a probability space for N continuous random variables is defined by considering the Borel σ-algebra over $\mathbb{R}^N$ and a Lebesgue measure on this set as the probability measure. The probability space is lifted to cover the entire program using the least model semantics of constraint logic programs.

If an atom encodes a continuous random variable (such as `g(X,Y)` above), asking the probability that a ground instantiation, such as `g(a,0.3)`, is true is not meaningful, as the probability that a continuous random variables takes a specific value is always 0. In this case you are more interested in computing the distribution of `Y` of a goal `g(a,Y)`, possibly after having observed some evidence. If the evidence is on an atom defining another continuous random variable, the definition of conditional probability cannot be applied, as the probability of the evidence would be 0 and so the fraction would be undefined. This problem is resolved in [10] by providing a definition using limits.

3   Inference

Computing all the worlds is impractical because their number is exponential in the number of ground probabilistic clauses. Alternative approaches have been considered that can be grouped in exact and approximate ones.

For exact inference from discrete program without function symbols a successful approach finds explanations for the query q [2], where an explanation is a set of clause choices that are sufficient for entailing the query. Once all explanations for the query are found, they are encoded as a Boolean formula in DNF (with a propositional variable per choice and a conjunction per explanation) and the problem is reduced to that of computing the probability that a propositional formula is true. This problem is difficult (#P complexity) but converting the DNF into a language from which the computation of the probability is polynomial (knowledge compilation [1]) yields algorithm able to handle problems of significant size [2,16].

For approximate inference one of the most used approach consists in Monte Carlo sampling, following the sampling interpretation of the semantics given above. Monte Carlo backward reasoning has been implemented in [7,12] and found to give good performance in terms of quality of the solutions and of running time. Monte Carlo sampling is attractive for the simplicity of its implementation and because you can improve the estimate as more time is available. Moreover, Monte Carlo can be used also for programs with function symbols, in which goals may have infinite explanations and exact inference may loop. In sampling, infinite explanations have probability 0, so the computation of each sample eventually terminates.

Monte Carlo inference provides also smart algorithms for computing conditional probabilities: rejection sampling or Metropolis-Hastings Markov Chain Monte Carlo (MCMC). In rejection sampling [19], you first query the evidence and, if the query is successful, query the goal in the same sample, otherwise the sample is discarded. In Metropolis-Hastings MCMC [9], a Markov chain is built by taking an initial sample and by generating successor samples.

The initial sample is built by randomly sampling choices so that the evidence is true. A successor sample is obtained by deleting a fixed number of sampled probabilistic choices. Then the evidence is queried by taking a sample starting with the undeleted choices. If the query succeeds, the goal is queried by taking a sample. The sample is accepted with a probability of $\min\{1, \frac{N_0}{N_1}\}$ where N_0 is the number of choices sampled in the previous sample and N_1 is the number of choices sampled in the current sample. Then the number of successes of the query is increased by 1 if the query succeeded in the last accepted sample. The final probability is given by the number of successes over the number of samples.

When you have evidence on ground atoms that have continuous values as arguments, you can still use Monte Carlo sampling. You cannot use rejection sampling or Metropolis-Hastings, as the probability of the evidence is 0, but you can use likelihood weighting [10] to obtain samples of continuous arguments of a goal. For each sample to be taken, likelihood weighting samples the query and then assigns a weight to the sample on the basis of evidence. The weight is computed by deriving the evidence backward in the same sample of the query starting with a weight of one: each time a choice should be taken or a continuous variable sampled, if the choice/variable has already been taken, the current weight is multiplied by probability of the choice/by the density value of the continuous value.

If likelihood weighting is used to find the posterior density of a continuous random variable, you obtain a set of samples for the variables with each sample associated with a weight that can be interpreted as a relative frequency. The set of samples without the weight, instead, can be interpreted as the prior density of the variable. These two set of samples can be used to plot the density before and after observing the evidence.

You can sample arguments of queries also for discrete goals: in this case you get a discrete distribution over the values of one or more arguments of a goal. If the query predicate is determinate in each world, i.e., given values for input

arguments there is a single value for output arguments that make the query true, for each sample you get a single value. Moreover, if clauses sharing an atom in the head are mutually exclusive, i.e., in each world the body of at most one clause is true, then the query defines a probability distribution over output arguments. In this way we can simulate those languages such as PRISM and Stochastic Logic Programs that define probability distributions over arguments rather than probability distributions over truth values of ground atoms.

4 Inference with `cplint`

`cplint` on SWISH uses two modules for performing inference, `pita` for exact inference by knowledge compilation and `mcintyre` for approximate inference by sampling. In this section we discuss the algorithms and predicates provided by these two modules.

The unconditional probability of an atom can be asked using `pita` with the predicate

```
prob(:Query:atom,-Probability:float).
```

The conditional probability of an atom query given another atom evidence can be asked with the predicate

```
prob(:Query:atom,:Evidence:atom,-Probability:float).
```

With `mcintyre`, you can estimate the probability of a goal by taking a given number of sample using the predicate

```
mc_sample(:Query:atom,+Samples:int,-Probability:float).
```

You can ask conditional queries with rejection sampling or with Metropolis-Hastings MCMC too.

```
mc_rejection_sample(:Query:atom,:Evidence:atom,+Samples:int,
  -Successes:int,-Failures:int,-Probability:float).
```

In Metropolis-Hastings MCMC, `mcintyre` follows the algorithm proposed in [9] (the non adaptive version). The initial sample is built with a backtracking meta-interpreter that starts with the goal and randomizes the order in which clauses are selected during the search so that the initial sample is unbiased. Then the goal is queried using regular `mcintyre`.

A successor sample is obtained by deleting a number of sampled probabilistic choices given by parameter `Lag`. Then the evidence is queried using regular `mcintyre` starting with the undeleted choices. If the query succeeds, the goal is queried using regular `mcintyre`. The sample is accepted with the probability indicated in Section 3. In [9] the lag is always 1 but the proof in [9] that the above acceptance probability yields a valid Metropolis-Hastings algorithm holds also when forgetting more than one sampled choice, so the lag is user defined in `cplint`.

You can take a given number of sample with Metropolis-Hastings MCMC using

```
mc_mh_sample(:Query:atom,:Evidence:atom,+Samples:int,+Lag:int,
  -Successes:int,-Failures:int,-Probability:float).
```

Moreover, you can sample arguments of queries with rejection sampling and Metropolis-Hastings MCMC using

```
mc_rejection_sample_arg(:Query:atom,:Evidence:atom,+Samples:int,
  ?Arg:var,-Values:list).
mc_mh_sample_arg(:Query:atom,:Evidence:atom,+Samples:int,
  +Lag:int,?Arg:var,-Values:list).
```

Finally, you can compute expectations with

```
mc_expectation(:Query:atom,+N:int,?Arg:var,-Exp:float).
```

that computes the expected value of `Arg` in `Query` by sampling. It takes `N` samples of `Query` and sums up the value of `Arg` for each sample. The overall sum is divided by `N` to give `Exp`.

To compute conditional expectations, use

```
mc_mh_expectation(:Query:atom,:Evidence:atom,+N:int,
  +Lag:int,?Arg:var,-Exp:float).
```

For visualizing the results of sampling arguments you can use

```
mc_sample_arg_bar(:Query:atom,+Samples:int,?Arg:var,-Chart:dict).
mc_rejection_sample_arg_bar(:Query:atom,:Evidence:atom,
  +Samples:int,?Arg:var,-Chart:dict).
mc_mh_sample_arg_bar(:Query:atom,:Evidence:atom,+Samples:int,
  +Lag:int,?Arg:var,-Chart:dict).
```

that return in `Chart` a bar chart with a bar for each possible sampled value whose size is the number of samples returning that value.

When you have continuous random variables, you may be interested in sampling arguments of goals representing continuous random variables. In this way you can build a probability density of the sampled argument. When you do not have evidence or you have evidence on atoms not depending on continuous random variables, you can use the above predicates for sampling arguments.

When you have evidence on ground atoms that have continuous values as arguments, you need to use likelihood weighting [10] to obtain samples of continuous arguments of a goal.

For each sample to be taken, likelihood weighting uses a meta-interpreter to find a sample where the goal is true, randomizing the choice of clauses when more than one resolves with the goal in order to obtain an unbiased sample. This meta-interpreter is similar to the one used to generate the first sample in Metropolis-Hastings.

Then a different meta-interpreter is used to evaluate the weight of the sample. This meta-interpreter starts with the evidence as the query and a weight of 1. Each time the meta-interpreter encounters a probabilistic choice over a

continuous variable, it first checks whether a value has already been sampled. If so, it computes the probability density of the sampled value and multiplies the weight by it. If the value has not been sampled, it takes a sample and records it, leaving the weight unchanged. In this way, each sample in the result has a weight that is 1 for the prior distribution and that may be different from the posterior distribution, reflecting the influence of evidence.

The predicate

```
mc_lw_sample_arg(:Query:atom,:Evidence:atom,+N:int,?Arg:var,
  -ValList).
```

returns in `ValList` a list of couples `V-W` where `V` is a value of `Arg` for which `Query` succeeds and `W` is the weight computed by likelihood weighting according to `Evidence` (a conjunction of atoms is allowed here).

You can use the samples to draw the probability density function of the argument. The predicate

```
histogram(+List:list,+NBins:int,-Chart:dict).
```

draws a histogram of the samples in `List` dividing the domain in `NBins` bins. `List` must be a list of couples of the form `[V]-W` where `V` is a sampled value and `W` is its weight. This is the format of the list of samples returned by argument sampling predicates except `mc_lw_sample_arg/5` that returns a list of couples `V-W`. In this case you can use

```
densities(+PriorList:list,+PostList:list,+NBins:int,-Chart:dict).
```

that draws a line chart of the density of two sets of samples, usually prior and post observations. The samples from the prior are in `PriorList` as couples `[V]-W`, while the samples from the posterior are in `PostList` as couples `V-W` where `V` is a value and `W` its weight. The lines are drawn dividing the domain in `NBins` bins.

5   Examples

5.1   Generative Model

Program `arithm.pl` encodes a model for generating random functions:

```
eval(X,Y) :- random_fn(X,0,F), Y is F.
op(+):0.5;op(-):0.5.
random_fn(X,L,F) :- comb(L), random_fn(X,l(L),F1),
  random_fn(X,r(L),F2), op(Op), F=..[Op,F1,F2].
random_fn(X,L,F) :- \+ comb(L), base_random_fn(X,L,F).
comb(_):0.3.
base_random_fn(X,L,X) :- identity(L).
base_random_fn(_,L,C) :- \+ identity(L), random_const(L,C).
identity(_):0.5.
random_const(_,C):discrete(C,[0:0.1,1:0.1,2:0.1,3:0.1,4:0.1,
  5:0.1,6:0.1,7:0.1,8:0.1,9:0.1]).
```

A random function is either an operator ('+' or '-') applied to two random functions or a base random function. A base random function is either an identity or a constant drawn uniformly from the integers $0, \ldots, 9$.

You may be interested in the distribution of the output values of the random function with input 2 given that the function outputs 3 for input 1. You can get this distribution with

```
?- mc_mh_sample_arg_bar(eval(2,Y),eval(1,3),1000,1,Y,V).
```

that samples 1000 values for `Y` in `eval(2,Y)` and returns them in `V`. A bar graph of the frequencies of the sampled value is shown in Figure 2a. Since each world of the program is determinate, in each world there is a single value of `Y` in `eval(2,Y)` and the list of sampled values contain a single element.

5.2   Gaussian Mixture Model

Example `gaussian_mixture.pl` defines a mixture of two Gaussians:

```
heads:0.6;tails:0.4.
g(X): gaussian(X,0, 1).
h(X): gaussian(X,5, 2).
mix(X) :- heads, g(X).
mix(X) :- tails, h(X).
```

The argument `X` of `mix(X)` follows a model mixing two Gaussian, one with mean 0 and variance 1 with probability 0.6 and one with mean 5 and variance 2 with probability 0.4. The query

```
?- mc_sample_arg(mix(X),10000,X,L0), histogram(L0,40,Chart).
```

draws the density of the random variable `X` of `mix(X)`, shown in Figure 2b.

5.3   Bayesian Estimation

Let us consider a problem proposed on the Anglican [20] web site[5]. We are trying to estimate the true value of a Gaussian distributed random variable, given some observed data. The variance is known (its value is 2) and we suppose that the mean has itself a Gaussian distribution with mean 1 and variance 5. We take different measurement (e.g. at different times), indexed with an integer.

This problem can be modeled with (`gauss_mean_est.pl`)

```
value(I,X) :-  mean(M),value(I,M,X).
mean(M): gaussian(M,1.0, 5.0).
value(_,M,X): gaussian(X,M, 2.0).
```

Given that we observe 9 and 8 at indexes 1 and 2, how does the distribution of the random variable (value at index 0) changes with respect to the case of no observations? This example shows that the parameters of the distribution atoms can be taken from the probabilistic atom. The query

---

[5] http://www.robots.ox.ac.uk/~fwood/anglican/examples/viewer/?worksheet=
gaussian-posteriors

```
?- mc_sample_arg(value(0,Y),100000,Y,L0),
   mc_lw_sample_arg(value(0,X),(value(1,9),value(2,8)),1000,X,L),
   densities(L0,L,40,Chart).
```

takes 100000 samples of argument `X` of `value(0,X)` before and after observing
`value(1,9),value(2,8)` and draws the prior and posterior densities of the sam-
ples using a line chart. Figure 2c shows the resulting graph where the posterior
is clearly peaked at around 9.

5.4 Kalman Filter

Example `kalman_filter.pl` (adapted from [9])

```
kf(N,O, T) :- init(S), kf_part(0, N, S,O,T).
kf_part(I, N, S,[V|RO], T) :- I < N, NextI is I+1, trans(S,I,NextS),
  emit(NextS,I,V), kf_part(NextI, N, NextS,RO, T).
kf_part(N, N, S, [],S).
trans(S,I,NextS) :- {NextS =:= E + S}, trans_err(I,E).
emit(NextS,I,V) :- {V =:= NextS+X}, obs_err(I,X).
init(S):gaussian(S,0,1).
trans_err(_,E):gaussian(E,0,2).
obs_err(_,E):gaussian(E,0,1).
```

encodes a Kalman filter, i.e., a Hidden Markov model with a real value as state
and a real value as output. The next state is given by the current state plus
Gaussian noise (with mean 0 and variance 2 in this example) and the output is
given by the current state plus Gaussian noise (with mean 0 and variance 1 in
this example). A Kalman filter can be considered as modeling a random walk of
a single continuous state variable with noisy observations.

Continuous random variables are involved in arithmetic expressions (in the
predicates `trans/3` and `emit/3`). It is often convenient, as in this case, to use
CLP(R) constraints because in this way the same clauses can be used both to
sample and to evaluate the weight the sample on the basis of evidence, otherwise
different clauses have to be written.

Given that at time 0 the value 2.5 was observed, what is the distribution of
the state at time 1 (filtering problem)? Likelihood weighting is used to condition
the distribution on evidence on a continuous random variable (evidence with
probability 0). CLP(R) constraints allow both sampling and weighting samples
with the same program: when sampling, the constraint `{V=:=NextS+X}` is used
to compute `V` from `X` and `NextS`. When weighting, `V` is known and the constraint
is used to compute `X` from `V` and `NextS`, which is then given the density value at
`X` as weight. The above query can be expressed with

```
?- mc_sample_arg(kf(1,_O1,Y),10000,Y,L0),
   mc_lw_sample_arg(kf(1,_O2,T),kf(1,[2.5],_T),10000,T,L),
   densities(L0,L,40,Chart).
```

that returns the graph of Figure 2d, from which it is evident that the posterior
distribution is peaked around 2.5.

5.5 Stochastic Logic Programs

Stochastic logic programs (SLPs) [8] are a probabilistic formalism where each clause is annotated with a probability. The probabilities of all clauses with the same head predicate sum to one and define a mutually exclusive choice on how to continue a proof. Furthermore, repeated choices are independent, i.e., no stochastic memorization is done. SLPs are used most commonly for defining a distribution over the values of arguments of a query. SLPs are a direct generalization of probabilistic context-free grammars and are particularly suitable for representing them. For example, the grammar

```
0.2:S->aS    0.2:S->bS    0.3:S->a    0.3:S->b
```

can be represented with the SLP

```
0.2::s([a|R]):- s(R).  0.2::s([b|R]):- s(R).
0.3::s([a]).           0.3::s([b]).
```

This SLP can be encoded in `cplint` as (`slp_pcfg.pl`):

```
s_as(N):0.2;s_bs(N):0.2;s_a(N):0.3;s_b(N):0.3.
s([a|R],N0):- s_as(N0), N1 is N0+1, s(R,N1).
s([b|R],N0):- s_bs(N0), N1 is N0+1, s(R,N1).
s([a],N0):- s_a(N0).    s([b],N0):- s_b(N0).
s(L):-s(L,0).
```

where we have added an argument to `s/1` for passing a counter to ensure that different calls to `s/2` are associated to independent random variables.

By using the argument sampling features of `cplint` we can simulate the behavior of SLPs. For example the query

```
?- mc_sample_arg_bar(s(S),100,S,L).
```

samples 100 sentences from the language and draws the bar chart of Figure 2e.


6 Conclusions

PLP has now become flexible enough to encode and solve problems usually tackled only with other Probabilistic Programming paradigms, such as functional or imperative PP. `cplint` on SWISH allows the user to exploit these new features of PLP without the need of a complex installation process. In this way we hope to reach out to a wider audience and increase the user base of PLP. In the future we plan to explore more in detail the connection with probabilistic programming using functional/imperative languages and exploit the techniques developed in that field.

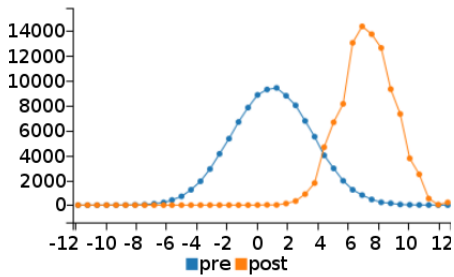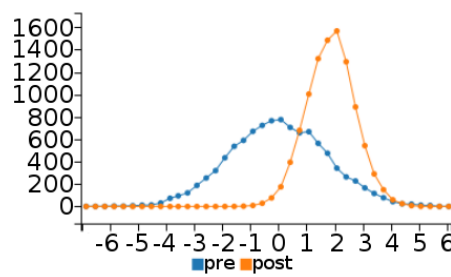A complete online tutorial [15] is available at `http://ds.ing.unife.it/~gcota/plptutorial/`.
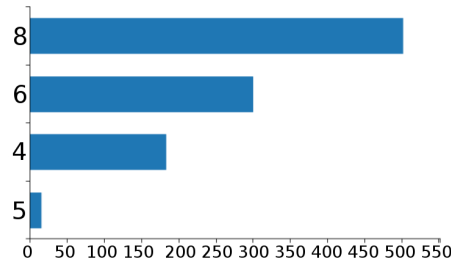
(a) Distribution of sampled values the arithm.pl example.
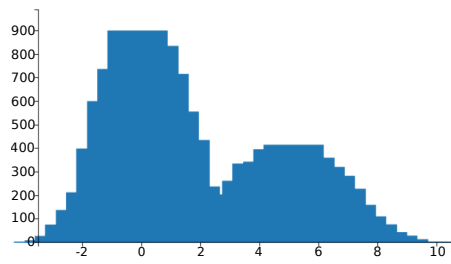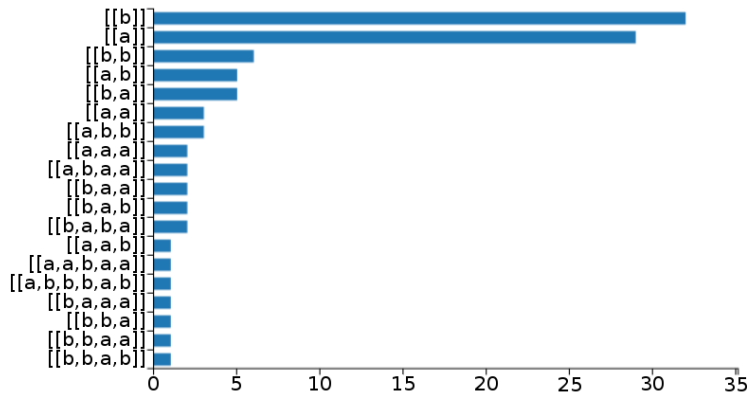

(b) Density of X of mix(X) in gaussian_mixture.pl.

(c) Prior and posterior densities in gauss_mean_est.pl.

(d) Prior and posterior densities in kalman.pl.

(e) Samples of sentences of the language defined in slp_pcfg.pl.

Fig. 2: Graphs of examples

References

1. Darwiche, A., Marquis, P.: A knowledge compilation map. J. Artif. Intell. Res. 17, 229–264 (2002)
2. De Raedt, L., Kimmig, A., Toivonen, H.: ProbLog: A probabilistic Prolog and its application in link discovery. In: IJCAI 2007. vol. 7, pp. 2462–2467. AAAI Press, Palo Alto, California USA (2007)
3. De Raedt, L., Kimmig, A.: Probabilistic (logic) programming concepts. Mach. Learn. 100(1), 5–47 (2015)
4. Fierens, D., den Broeck, G.V., Renkens, J., Shterionov, D.S., Gutmann, B., Thon, I., Janssens, G., De Raedt, L.: Inference and learning in probabilistic logic programs using weighted boolean formulas. Theor. Pract. Log. Prog. 15(3), 358–401 (2015)
5. Gutmann, B., Thon, I., Kimmig, A., Bruynooghe, M., Raedt, L.D.: The magic of logical inference in probabilistic programming. Theor. Pract. Log. Prog. 11(4-5), 663–680 (2011)
6. Islam, M.A., Ramakrishnan, C., Ramakrishnan, I.: Inference in probabilistic logic programs with continuous random variables. Theor. Pract. Log. Prog. 12, 505–523 (2012)
7. Kimmig, A., Demoen, B., De Raedt, L., Costa, V.S., Rocha, R.: On the implementation of the probabilistic logic programming language ProbLog. Theor. Pract. Log. Prog. 11(2-3), 235–262 (2011)
8. Muggleton, S.: Learning stochastic logic programs. Electron. Trans. Artif. Intell. 4(B), 141–153 (2000)
9. Nampally, A., Ramakrishnan, C.: Adaptive MCMC-based inference in probabilistic logic programs. arXiv preprint arXiv:1403.6036 (2014), http://arxiv.org/pdf/1403.6036.pdf
10. Nitti, D., De Laet, T., De Raedt, L.: Probabilistic logic programming for hybrid relational domains. Mach. Learn. 103(3), 407–449 (2016)
11. Pfeffer, A.: Practical Probabilistic Programming. Manning Publications (2016)
12. Riguzzi, F.: MCINTYRE: A Monte Carlo system for probabilistic logic programming. Fund. Inform. 124(4), 521–541 (2013)
13. Riguzzi, F.: The distribution semantics for normal programs with function symbols. International Journal of Approximate Reasoning 77, 1 – 19 (October 2016)
14. Riguzzi, F., Bellodi, E., Lamma, E., Zese, R., Cota, G.: Probabilistic logic programming on the web. Software Pract. and Exper. (2015)
15. Riguzzi, F., Cota, G.: Probabilistic logic programming tutorial. The Association for Logic Programming Newsletter 29(1) (March/April 2016), http://www.cs.nmsu.edu/ALP/2016/03/probabilistic-logic-programming-tutorial/
16. Riguzzi, F., Swift, T.: The PITA system: Tabling and answer subsumption for reasoning under uncertainty. Theor. Pract. Log. Prog. 11(4–5), 433–449 (2011)
17. Sato, T.: A statistical learning method for logic programs with distribution semantics. In: Sterling, L. (ed.) ICLP-95. pp. 715–729. MIT Press, Cambridge, Massachusetts (1995)
18. Vennekens, J., Verbaeten, S., Bruynooghe, M.: Logic programs with annotated disjunctions. In: 20th International Conference on Logic Programming. LNCS, vol. 3131, pp. 195–209. Springer, Berlin Heidelberg, Germany (2004)
19. Von Neumann, J.: Various techniques used in connection with random digits. Nat. Bureau Stand. Appl. Math. Ser. 12, 36–38 (1951)
20. Wood, F., van de Meent, J.W., Mansinghka, V.: A new approach to probabilistic programming inference. In: Proceedings of the 17th International conference on Artificial Intelligence and Statistics. pp. 1024–1032 (2014)