

cplint on SWISH: Probabilistic Logical Inference with a Web Browser

Marco Alberti^a Elena Bellodi^b Giuseppe Cota^b Fabrizio Riguzzi^a Riccardo Zese^b

^a *Dipartimento di Matematica e Informatica – University of Ferrara
Via Saragat 1, I-44122, Ferrara, Italy*

^b *Dipartimento di Ingegneria – University of Ferrara
Via Saragat 1, I-44122, Ferrara, Italy*

Abstract. `cplint` on SWISH is a web application that allows users to perform reasoning tasks on probabilistic logic programs. Both inference and learning systems can be performed: 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 and particle filtering are used. `cplint` on SWISH is also able to sample goals' arguments and to graph the results. This paper reports on advances and new features of `cplint` on SWISH, including the capability of drawing the binary decision diagrams created during the inference processes.

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

1. Introduction

Probabilistic Programming (PP) [Pfe16] allows users to define complex probabilistic models and perform inference and learning on them. In particular, within the whole set of PP proposals, Probabilistic Logic Programming (PLP) [DRK15] can model complex domains containing many uncertain relationships among their entities.

Many systems have been proposed for reasoning with PLP, most of them freely available. However, in many cases the installation process of these systems is prone to errors and they require that one must follow a non-trivial learning phase before becoming a proficient user. In order to facilitate the use of our PLP systems, we developed `cplint` on SWISH [RBL⁺16], a web application for reasoning on PLP with just a web browser. The application is available at <http://cplint.lamping.unife.it> and Figure 1 shows its interface. Here, users have just to use their browser to post queries and see the results, while all the execution is demanded to a web server.

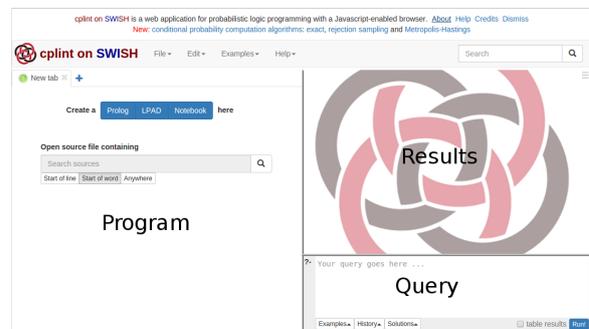


Fig. 1. Interface of `cplint` on SWISH.

`cplint` on SWISH uses the reasoning algorithms included in the `cplint` suite, including exact and approximate inference and parameter and structure learning. This article extends [ACRZ16], where we presented several algorithms for computing conditional probabilities with exact, rejection sampling and Metropolis-Hasting methods. The system also allowed hybrid programs, where some of the random variables are continuous, a feature

that is, to the best of our knowledge, a novelty for web applications.

A similar system is ProbLog2 [FdBR⁺15], which also has an online version¹. The main difference between `cplint` on SWISH and ProbLog2 is that the former currently offers also structure learning, approximate conditional inference through sampling and handling of continuous variables. Moreover, `cplint` on SWISH is based on SWISH² - a web framework for Logic Programming using features and packages of SWI-Prolog and its Pengines library - and utilizes a Prolog-only software stack in the server, whereas ProbLog2 relies on several different technologies, including 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.

In this paper we aim to show that PP based on Logic Programming is flexible and mature enough to develop complex systems equipped with many features and algorithms with relative ease. PLP has moved from a niche language suitable for specific applications to a versatile framework able to offer many features that were previously typical of imperative or functional PP languages only, such as hybrid programs, likelihood weighting and particle filtering.

`cplint` on SWISH not only offers such features but also the possibility of developing programs online and in collaboration. The system contains a wide variety of examples, representing many probabilistic models such as Markov Logic Networks, generative models, Gaussian processes, Gaussian mixtures, Dirichlet processes, Bayesian estimation and Kalman filters. As such, the system shows the soundness of PLP also from a software engineering point of view, opening the way to complex industrial/real world applications.

After introducing the syntax and semantics of PLP in Section 2, we discuss approaches for inference in Section 3. Section 4 presents the predicates the user can call to perform inference in `cplint` on SWISH. Section 5 contains a number of examples that illustrate the new features of `cplint` on SWISH and Section 6 concludes the paper. All the examples presented 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 [Sat95] is one of the most used approaches for representing probabilistic information in Logic Programming and it is at the basis 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. 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 [VVB04].

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

¹<https://dtai.cs.kuleuven.be/problog/>

²<http://swish.swi-prolog.org>

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 [Rig16].

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? `cpaint` 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 [GTK⁺11] and [IRR12]. In [NDLDR16] 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 seman-

tics: 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 [IRR12] 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 variable 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 [NDLDR16] 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 programs without function symbols a successful approach finds explanations for the query q [DKT07], 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 probabil-

ity is polynomial (knowledge compilation [DM02]) yields algorithms able to handle problems of significant size [DKT07,RS11]. One of the most efficient ways of solving the problem makes use of the language of Binary Decision Diagrams (BDDs). They represent a function $f(X)$ taking Boolean values on a set of Boolean variables X by means of a rooted graph that has one level for each variable. Each node is associated with the variable of its level and has 2 children. Given values for all the variables, we can compute the value of the function (in this case the DNF formula) by traversing the graph from the root and returning the value associated with the leaf that is reached: the diagram will have a path to a 1-leaf for each world where the query is true.

Given a BDD, the probability of the corresponding Boolean function can be computed with a dynamic programming algorithm. The probability of a leaf is either 1 or 0 if the leaf is 1 or 0 respectively. The probability $P(n)$ of a node n associated with variable V is $P(n) = P(V) \cdot P(\text{child}_1(n)) + (1 - P(V)) \cdot P(\text{child}_0(n))$ where $\text{child}_1(n)$ ($\text{child}_0(n)$) is the 1-child (0-child) of n . In practice, memorization of intermediate results is used to avoid recomputation at nodes that are shared between multiple paths.

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 [KDD⁺11,Rig13] 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 [VN51], 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 [NR14], 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 [NDLDR16] 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 the probability of the choice/by the density value of the continuous variable.

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 sets of samples can be used to plot the density before and after observing the evidence.

When you have a dynamic model and observations on continuous variables for a number of time points, or your evidence is represented by many atoms, likelihood weighting has numerical stability problems, as samples' weight goes rapidly to 0. In this case, particle filtering can be useful, be-

cause it periodically resamples the individual samples/particles so that their weight is reset to 1.

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, you get a single value for each sample. 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 a query atom given an evidence atom can be asked with the predicate

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

The BDD representation of the explanations for the query atom can be obtained with the predicate

```
bdd_dot_string(:Query:atom,BDD:string,Var:list).
```

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

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

Moreover, you can sample arguments of queries with:

```
mc_sample_arg(:Query:atom,+Samples:int,?Arg:var,
              -Values:list).
```

that returns in `Values` a list of couples V-W where L is the list of values of `Arg` for which `Query` succeeds in a world sampled at random and N is the number of samples returning that list of values. The version

```
mc_sample_arg_first(:Query:atom,+Samples:int,
                   ?Arg:var,-Values:list).
```

also samples arguments of queries but just computes the first answer of the query for each sampled world.

You can ask conditional queries with rejection sampling or with Metropolis-Hastings MCMC, too. In the first case, the available predicate is:

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

In the second case, `mcintyre` follows the algorithm proposed in [NR14] (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 the 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 [NR14] the `lag` is always 1. However, in [NR14], the proof that the acceptance probability (which is $\min\{1, \frac{N_0}{N_1}\}$ 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) 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 samples 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 the arguments of the 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 returns the expected value of the argument `Arg` in `Query` by sampling. It takes `N` samples of `Query` and sums up the value of `Arg` in 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 with continuous values as arguments, you need to use likelihood weighting [NDLDR16] or particle filtering [FC90,KF09] to obtain samples of the 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 had 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:list).

```

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).

In particle filtering, the evidence is a list of atoms. Each sample is weighted by the likelihood of an element of the evidence and constitutes a particle. After weighting, particles are resampled and the next element of the evidence is considered.

The predicate

```

mc_particle_sample_arg(:Query:atom,+Evidence:term,
  +Samples:int,?Arg:var,-Values:list).

```

samples the argument `Arg` of `Query` using particle filtering given that `Evidence` is true. `Evidence` is a list of goals and `Query` can be either a single goal or a list of goals.

When `Query` is a single goal, the predicate returns in `Values` a list of couples `V-W` where `V` is a value of `Arg` for which `Query` succeeds in a particle in the last set of particles, and `W` is the weight of the particle. For each element of `Evidence`, the particles are obtained by sampling `Query` in each current particle and weighting the particle by the likelihood of the evidence element.

When `Query` is a list of goals, `Arg` is a list of variables, one for each query of `Query`; in this case `Arg` and `Query` must have the same length as `Evidence`. `Values` is then a list of the same length as `Evidence` and each of its elements is a list of couples `V-W` where `V` is a value of the corresponding element of `Arg` for which the corresponding element of `Query` succeeds in a particle, and `W` is the weight of the particle. For each element

of **Evidence**, the particles are obtained by sampling the corresponding element of **Query** in each current particle and weighting the particle by the likelihood of the evidence element.

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. Binary Decision Diagrams

Example `epidemic.pl` models the development of an epidemic or a pandemic with an LPAD: if somebody has the flu and the climate is cold, there is the possibility that an epidemic arises with probability 0.6 and the possibility that a pandemic arises with probability 0.3, whereas with 0.1 probability neither an epidemic nor a pandemic arises. We are uncertain whether the climate is cold but we know for sure that David and Robert have the flu.

```
epidemic:0.6; pandemic:0.3 :- flu(_), cold.
cold : 0.7.
flu(david).
flu(robert).
```

In order to calculate the probability that a pandemic arises, you can call the query:

```
?- prob(pandemic,Prob).
```

or

```
?- prob_bar(pandemic,Prob).
```

The latter shows the probabilistic results of the query as a histogram (Fig. 2a).

The corresponding BDD can be obtained with:

```
?- bdd_dot_string(pandemic,BDD,Var).
```

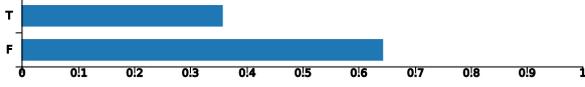
and is represented in Fig. 2b. A solid edge indicates a 1-child, a dashed edge indicates a 0-child and a dotted edge indicates a negated 0-child. Each level of the BDD is associated with a variable of the form X_{IJ} indicated on the left: I indicates the multivalued variable index and J the index of the Boolean variable of I .

5.2. Markov Logic Networks

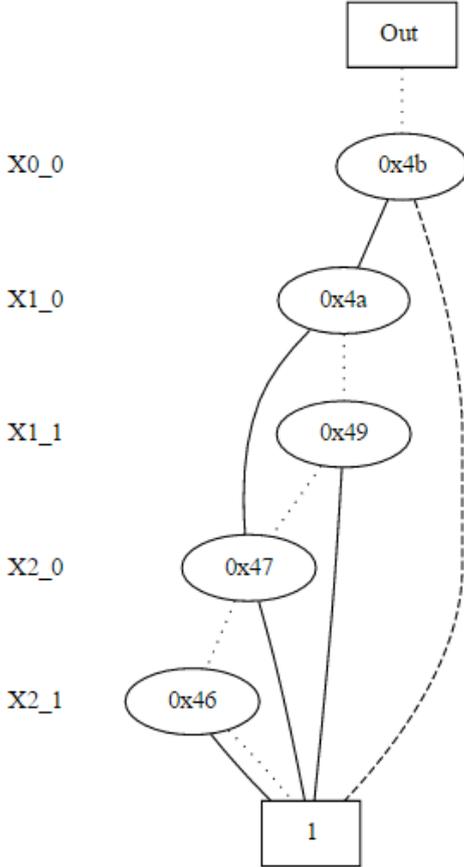
Markov Networks (MNs) and Markov Logic Networks (MLNs)[RD06] can be encoded with Probabilistic Logic Programming. The encoding is based on the observation that a MN factor can be represented with a Bayesian Network with an extra node that is always observed. In order to model MLN formulas with LPADs, we can add an extra atom $clause_i(\mathbf{X})$ for each formula $F_i = w_i C_i$ where w_i is the weight associated with C_i and \mathbf{X} is the vector of variables appearing in C_i . Then, when we ask for the probability of query q given evidence e , we have to ask for the probability of q given $e \wedge ce$, where ce is the conjunction of the groundings of $clause_i(\mathbf{X})$ for all values of i . Then, clause C_i should be transformed into a Disjunctive Normal Form formula $C_{i1} \vee \dots \vee C_{im_i}$, where the disjuncts are mutually exclusive and the LPAD should contain the clauses $clause_i(\mathbf{X}) : e^\alpha / (1 + e^\alpha) \leftarrow C_{ij}$ for all j in $1, \dots, m_i$. Similarly, $\neg C_i$ should be transformed into a disjoint sum $D_{i1} \vee \dots \vee D_{im_i}$ and the LPAD should contain the clauses $clause_i(\mathbf{X}) : 1 / (1 + e^\alpha) \leftarrow D_{il}$ for all l in $1, \dots, m_i$.

Alternatively, if α is negative, e^α will be smaller than 1 and we can use the two probability values e^α and 1 with the clauses $clause_i(\mathbf{X}) : e^\alpha \leftarrow C_{ij} \dots clause_i(\mathbf{X}) \leftarrow D_{il}$. This solution has the advantage that some clauses are certain, reducing the number of random variables. If α is positive in the formula αC , we can consider the equivalent formula $-\alpha \neg C$.

MLN formulas can also be added to a regular probabilistic logic program, their effect being



(a) Histogram of the probabilistic result of query *pandemic* in the *epidemic.pl* example.



(b) Binary Decision Diagram for query *pandemic* in the *epidemic.pl* example.

Fig. 2. Graphical representations for query *pandemic* in *epidemic.pl*.

equivalent to a soft form of evidence, where certain worlds are weighted more than others. This is similar to soft evidence in Figaro [Pfe16].

The transformation above is illustrated by the following example. Here, \cong indicates the truncation function. Given the MLN

```
1.5 Intelligent(x) => GoodMarks(x)
1.1 Friends(x,y) => (Intelligent(x) <=> Intelligent(y))
```

the first formula is translated into the clauses:

```
clause1(X):0.8175 :- \+intelligent(X).
clause1(X):0.1824 :- intelligent(X),
                    \+good_marks(X).
clause1(X):0.8175 :- intelligent(X),good_marks(X).
```

where $0.8175 \cong e^{1.5}/(1+e^{1.5})$ and $0.1824 \cong 1/(1+e^{1.5})$.

The second formula is translated into the clauses

```
clause2(X,Y):0.7502 :- \+friends(X,Y).
clause2(X,Y):0.7502 :- friends(X,Y),
                        intelligent(X),
                        intelligent(Y).
clause2(X,Y):0.7502 :- friends(X,Y),
                        \+intelligent(X),
                        \+intelligent(Y).
clause2(X,Y):0.2497 :- friends(X,Y),
                        intelligent(X),
                        \+intelligent(Y).
clause2(X,Y):0.2497 :- friends(X,Y),
                        \+intelligent(X),
                        intelligent(Y).
```

where $0.7502 \cong e^{1.1}/(1+e^{1.1})$ and $0.2497 \cong 1/(1+e^{1.1})$.

A priori we have a uniform distribution over student intelligence, good marks and friendship:

```
intelligent(_):0.5.
good_marks(_):0.5.
friends(_,_):0.5.
```

and there are two students:

```
student(anna).
student(bob).
```

We have evidence that Anna is friend with Bob and Bob is intelligent. The evidence must also include the truth of all groundings of the *clause_N* predicates:

```
evidence_mln :- clause1(anna),clause1(bob),
                clause2(anna,anna),clause2(anna,bob),
                clause2(bob,anna),clause2(bob,bob).
ev_intelligent_bob_friends_anna_bob :-
    intelligent(bob),friends(anna,bob),
    evidence_mln.
```

If we want to query the probability that Anna gets good marks given the evidence, we can ask:

```
?- prob(good_marks(anna),
        ev_intelligent_bob_friends_anna_bob,P).
```

while the prior probability of Anna getting good marks is given by:

```
?- prob(good_marks(anna),evidence_mln,P).
```

The probability resulting from the first query is higher ($P = 0.733$) than the second query ($P = 0.607$), since it is conditioned to the evidence that Bob is intelligent and Anna is her friend.

In the alternative transformation, the first MLN formula is translated into:

```
clause1(X) :- \+intelligent(X).
clause1(X):0.2231 :- intelligent(X),\+good_marks(X).
clause1(X) :- intelligent(X), good_marks(X).
```

where $0.2231 \cong e^{-1.5}$.

5.3. 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,1(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 $0, \dots, 9$.

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 values is shown in Figure 3. 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 contains a single element.

5.4. Gaussian Process

A Gaussian Process (GP) (example `gpr.pl`) defines a probability distribution over functions. This

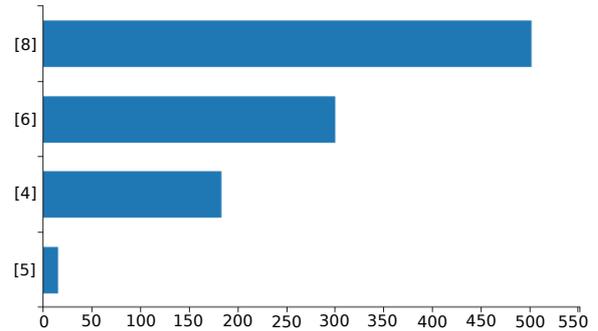


Fig. 3. Distribution of sampled values in the `arithm.pl` example.

distribution has the property that, given N values, their image through a function sampled from the gaussian process follows a multivariate normal with mean 0 and covariance matrix K . A Gaussian Process is defined by a kernel function k that determines K . GPs can be used for regression: the random functions predicts the y value corresponding to a x value given a set X and Y of observed values. When performing GP regression, you choose the kernel and you want to estimate the parameters of the kernel. You can define a prior distribution over the parameters. The following program can sample kernels (and thus functions) and compute the expected value of the predictions for a squared exponential kernel (defined by predicate `sq_exp_p`) with parameters l uniformly distributed in $1, 2, 3$ and σ uniformly distributed in $[-2, 2]$.

The predicate `gp`, given a list of values X and a kernel name, returns in Y the list of values of type $f(x)$ where x belongs to X and f is a function sampled from the Gaussian process. `gp_predict`, given the points described by the lists XT and YT and a kernel, predict the Y values of points with X values in XP and returns them in YP . Prediction is performed by Gaussian process regression.

```
gp(X,Kernel,Y) :-
    compute_cov(X,Kernel,0,C),
    gp(C,Y).

gp(Cov,Y):gaussian(Y,Mean,Cov):-
    length(Cov,N),
    list0(N,Mean).

compute_cov(X,Kernel,Var,C) :-
    length(X,N),
    cov(X,N,Kernel,Var,CT,CND),
    transpose(CND,CNDT),
```

```

matrix_sum(CT,CNDT,C).

cov([],_,_,[],[]).
cov([XH|XT],N,Ker,Var,[KH|KY],[KHND|KYND]) :-
  length(XT,LX),
  N1 is N-LX-1,
  list0(N1,KHO),
  cov_row(XT,XH,Ker,KH1),
  call(Ker,XH,XH,KXH),
  KXH is KXH+Var,
  append([KHO,[KXH],KH1],KH),
  append([KHO,[0],KH1],KHND),
  cov(XT,N,Ker,Var,KY,KYND).

cov_row([],_,_,[]).
cov_row([H|T],XH,Ker,[KH|KT]) :-
  call(Ker,H,XH,KH),
  cov_row(T,XH,Ker,KT).

gp_predict(XP,Kernel,Var,XT,YT,YP) :-
  compute_cov(XT,Kernel,Var,C),
  matrix_inversion(C,C_1),
  transpose([YT],YST),
  matrix_multiply(C_1,YST,C_1T),
  gp_predict_single(XP,Kernel,XT,C_1T,YP).

gp_predict_single([],_,_,[],[]).

gp_predict_single([XH|XT],Kernel,X,C_1T,[YH|YT]) :-
  compute_k(X,XH,Kernel,K),
  matrix_multiply([K],C_1T,[[YH]]),
  gp_predict_single(XT,Kernel,X,C_1T,YP).

compute_k([],_,_,[]).
compute_k([XH|XT],X,Ker,[HK|TK]) :-
  call(Ker,XH,X,HK),
  compute_k(XT,X,Ker,TK).

sq_exp_p(X,XP,K) :-
  sigma(Sigma),
  l(L),
  K is Sigma^2*exp(-(X-XP)^2)/(L^2).

l(L):uniform(L,[1,2,3]).
sigma(Sigma):uniform(Sigma,-2,2).

```

By calling the query `?-draw_fun(sq_exp_p,C)` over the program:

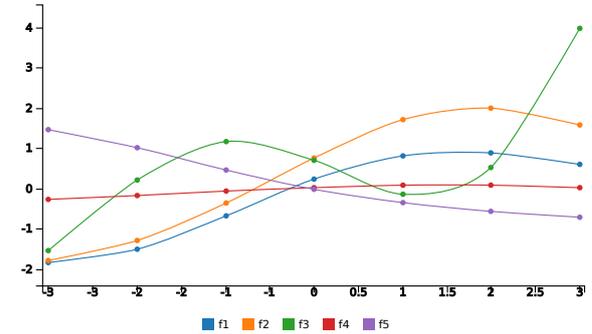
```

draw_fun(Kernel,C) :-
  X=[-3,-2,-1,0,1,2,3],
  draw_fun(X,Kernel,C).

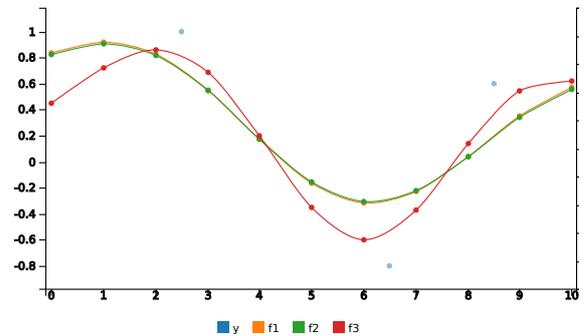
draw_fun(X,Kernel,C) :-
  mc_sample_arg_first(gp(X,Kernel,Y),5,Y,L).

```

we get 5 functions sampled from the Gaussian process with a squared exponential kernel at points $X = [-3, -2, -1, 0, 1, 2, 3]$, shown in Fig. 4a.



(a) Functions sampled from a Gaussian process with a squared exponential kernel in `gpr.pl`.



(b) Functions from a Gaussian process predicting points with $X=[0,\dots,10]$ with a squared exponential kernel in `gpr.pl`.

Fig. 4. Functions sampled from a Gaussian Process in the `gpr.pl` example.

The query `?-draw_fun_pred(sq_exp_p,C)` called over the program:

```

draw_fun_pred(Kernel,C) :-
  numlist(0,10,X),
  XT=[2.5,6.5,8.5],
  YT=[1,-0.8,0.6],
  mc_lw_sample_arg(gp_predict(X,Kernel,
  0.3,XT,YT,Y),gp(XT,Kernel,YT),5,Y,L).

```

draws 3 functions predicting points with X values in $[0,\dots,10]$ given the three couples of points $XT=[2.5,6.5,8.5]$, $YT=[1,-0.8,0.6]$ with a squared exponential kernel. The graph (Fig. 4b) shows as dots the given points.

5.5. 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).

```

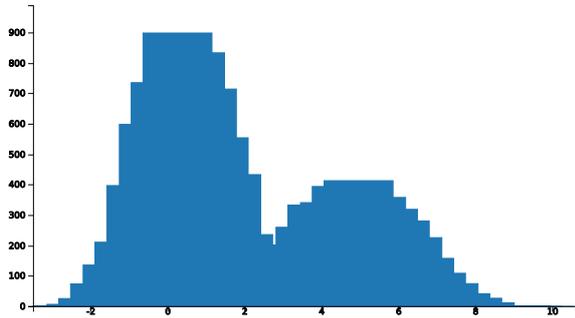


Fig. 5. Density of X of $\text{mix}(X)$ in `gaussian_mixture.pl`.

```
h(X):gaussian(X, 5, 2).
mix(X) :- heads, g(X).
mix(X) :- tails, h(X).
```

The argument X of $\text{mix}(X)$ follows a model mixing two Gaussians, 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 $\text{mix}(X)$, shown in Figure 5.

5.6. Dirichlet Processes

A Dirichlet process (DP) is a probability distribution whose range is itself a set of probability distributions. The DP is specified by a base distribution, which represents the expected value of the process. New samples have a nonzero probability of being equal to already sampled values. The process depends on a parameter α , called concentration parameter: with $\alpha \rightarrow 0$ a single value is sampled, with $\alpha \rightarrow \infty$ the distribution is equal to the base distribution. There are several equivalent views of the Dirichlet process, that are presented in the following.

5.6.1. The stick-breaking process

Example `dirichlet_process.pl` encodes the stick-breaking process view of the DP.

In this example the base distribution is a Gaussian with mean 0 and variance 1. To sample a value, a sample β_1 is taken from the beta distribution $Beta(1, \alpha)$ and a coin with heads probability equal to β_1 is flipped. If the coin lands on heads, a sample from the base distribution is taken and returned. Otherwise, a sample β_2 is taken again

from $Beta(1, \alpha)$ and a coin is flipped. This procedure is repeated until heads is obtained, the index i of β_i being the index of the value to be returned.

The distribution of values is handled by predicates `dp_value/3`, which returns in V the NV th sample from the DP with concentration parameter Alpha , and `dp_n_values/4`, which returns in L a list of $N-NO$ samples from the DP with concentration parameter Alpha .

The distribution of indexes is handled by predicates `dp_stick_index`.

```
dp_n_values(N,N,_Alpha,[]) :- !.

dp_n_values(NO,N,Alpha,[[V]-1|Vs]) :-
    NO<N,
    dp_value(NO,Alpha,V),
    N1 is NO+1,
    dp_n_values(N1,N,Alpha,Vs).

dp_value(NV,Alpha,V) :-
    dp_stick_index(NV,Alpha,I),
    dp_pick_value(I,V).

dp_pick_value(_,V):gaussian(V,0,1).

dp_stick_index(NV,Alpha,I) :-
    dp_stick_index(1,NV,Alpha,I).

dp_stick_index(N,NV,Alpha,V) :-
    stick_proportion(N,Alpha,P),
    choose_prop(N,NV,Alpha,P,V).

choose_prop(N,NV,_Alpha,P,N) :-
    pick_portion(N,NV,P).

choose_prop(N,NV,Alpha,P,V) :-
    neg_pick_portion(N,NV,P),
    N1 is N+1,
    dp_stick_index(N1,NV,Alpha,V).

stick_proportion(_,Alpha,P):
    beta(P,1,Alpha).

pick_portion(_,_,P):P;
neg_pick_portion(_,_,P):1-P.
```

The query

```
?-mc_sample_arg(dp_stick_index(1,10.0,V),200,V,L),
   histogram(L,100,Chart).
```

draws the density of indexes with concentration parameter 10 using 200 samples (Fig 6a).

The query

```
?- mc_sample_arg_first(dp_n_values(0,200,10.0,V),1,
                       V,L),
   L=[Vs-],
   histogram(Vs,100,Chart).
```

draws the density of values with concentration parameter 10 using 200 samples (Fig 6b).

The query

```
?-hist_repeated_indexes(100,40,G).
```

called over the program:

```
hist_repeated_indexes(Samples,NBins,Chart) :-
  repeat_sample(0,Samples,L),
  histogram(L,NBins,Chart).
```

```
repeat_sample(S,S,[]) :- !.
```

```
repeat_sample(S0,S,[[N]-1|LS]) :-
  mc_sample_arg_first(dp_stick_index(1,1,
                                     10.0,V),10,V,L),
  length(L,N),
  S1 is S0+1,
  repeat_sample(S1,S,LS).
```

shows the distribution of unique indexes in 100 samples with concentration parameter 10 (Fig 6c).

5.6.2. The Chinese restaurant process

The Chinese restaurant process is a discrete-time stochastic process, analogous to seating customers at tables in a Chinese restaurant. It results from considering the conditional distribution of one component assignment given all previous ones in a Dirichlet distribution mixture model with K components, and then taking the limit as K goes to infinity.

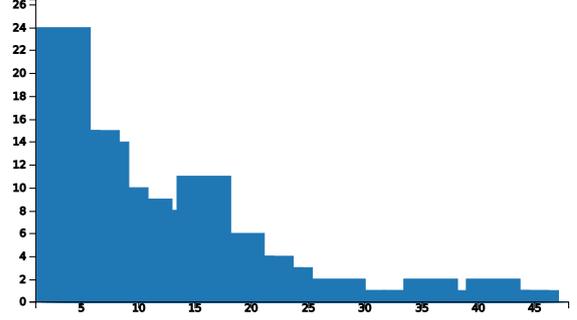
In the example `dp_chinese.pl` the base distribution is a Gaussian with mean 0 and variance 1. X_1 is drawn from the base distribution. For $n > 1$, with probability $\frac{\alpha}{\alpha+n-1}$ X_n is drawn from the base distribution; with probability $\frac{n_x}{\alpha+n-1}$ $X_n = x$, where n_x is the number of previous observations $X_j, j < n$, such that $X_j = x$. Counts are kept by predicate `update_counts/5`.

```
dp_n_values(NO,N,Alpha,[[V]-1|Vs],
            Counts0,Counts) :-
  NO<N,
  dp_value(NO,Alpha,Counts0,V,Counts1),
  N1 is NO+1,
  dp_n_values(N1,N,Alpha,Vs,Counts1,Counts).
```

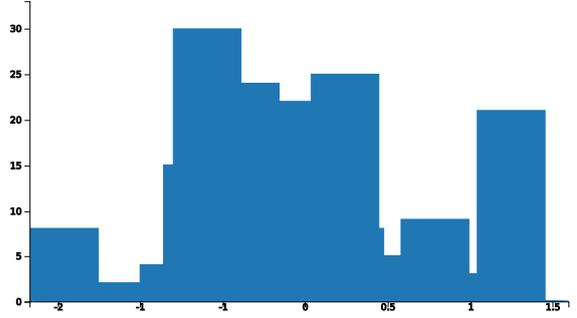
```
dp_value(NV,Alpha,Counts,V,Counts1) :-
  draw_sample(Counts,NV,Alpha,I),
  update_counts(0,I,Alpha,Counts,Counts1),
  dp_pick_value(I,V).
```

```
update_counts(_I0,_I,Alpha,[_C],[1,Alpha]) :- !.
```

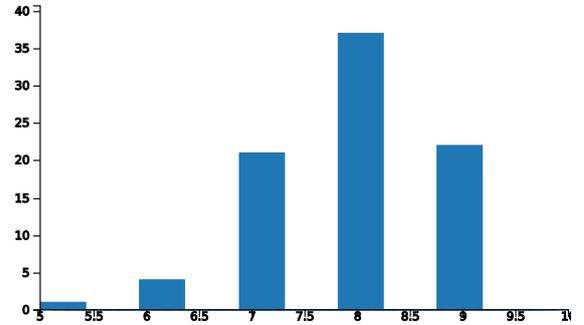
```
update_counts(I,I,_Alpha,[C|Rest],[C1|Rest]) :-
```



(a) Distribution of indexes with concentration parameter 10.



(b) Distribution of values with concentration parameter 10.



(c) Distribution of unique indexes with concentration parameter 10.

Fig. 6. Representation of the distributions in the `dirichlet_process.pl` example.

```
C1 is C+1.
```

```
update_counts(I0,I,Alpha,[C|Rest],[C|Rest1]) :-
  I1 is I0+1,
  update_counts(I1,I,Alpha,Rest,Rest1).
```

```
draw_sample(Counts,NV,Alpha,I) :-
  NS is NV+Alpha,
  maplist(div(NS),Counts,Probs),
```

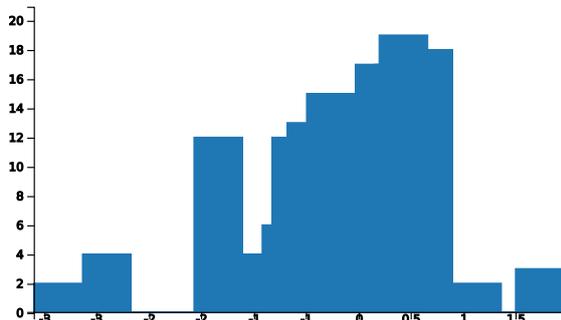


Fig. 7. Distribution of values in the `dp-chinese.pl` example.

```

length(Counts,LC),
numlist(1,LC,Values),
maplist(pair,Values,Probs,Discrete),
take_sample(NV,Discrete,I).

take_sample(_,D,V):discrete(V,D).

dp_pick_value(_,V):gaussian(V,0,1).

div(Den,V,P) :- P is V/Den.

pair(A,B,A:B).

```

The query

```

?- mc_sample_arg_first(dp_n_values(0,200,10.0,V,
                                   [10.0],_),1,V,L),
   L=[Vs-],histogram(Vs,100,Chart).

```

draws the distribution of values with concentration parameter 10 using 200 samples (Fig 7).

5.6.3. Mixture model

A particularly important application of Dirichlet processes is as a prior probability distribution in infinite mixture models. The target is to build a mixture model which does not require us to specify the number of k components from the beginning. In the example `dp_mix.pl` samples are drawn from a mixture of normal distributions whose parameters are defined by means of a Dirichlet process. For each component, the variance is sampled from a gamma distribution and the mean is sampled from a Gaussian with mean 0 and variance 30 times the variance of the component. The program in this case is equivalent to the one encoding the stick-breaking example, except for the `dp_pick_value/3` predicate that is reported in the following.

```

dp_pick_value(I,NV,V) :-
  ivar(I,IV),
  Var is 1.0/IV,
  mean(I,Var,M),
  value(NV,M,Var,V).

ivar(_,IV):gamma(IV,1,0.1).

mean(_,V0,M):gaussian(M,0,V) :- V is V0*30.

value(_,M,V,Val):gaussian(Val,M,V).

```

Given a vector of observations `obs([-1,7,3])`, the queries

```

?- prior(200,100,G).
?- post(200,100,G).

```

called over the program:

```

prior(Samples,NBins,Chart) :-
  mc_sample_arg_first(dp_n_values(0,Samples,10.0,V,
                                   1,V,L),
                    L=[Vs-],
                    histogram(Vs,NBins,Chart)).

post(Samples,NBins,Chart) :-
  obs(0),
  maplist(to_val,0,01),
  length(01,N),
  mc_lw_sample_arg_log(dp_value(0,10.0,T),
                      dp_n_values(0,N,10.0,01),Samples,T,L),
  maplist(keys,L,LW),
  min_list(LW,Min),
  maplist(exp(Min),L,L1),
  density(L1,NBins,-8,15,Chart).

```

```

keys(_-W,W).

```

```

exp(Min,L-W,L-W1) :- W1 is exp(W-Min).

```

```

to_val(V,[V]-1).

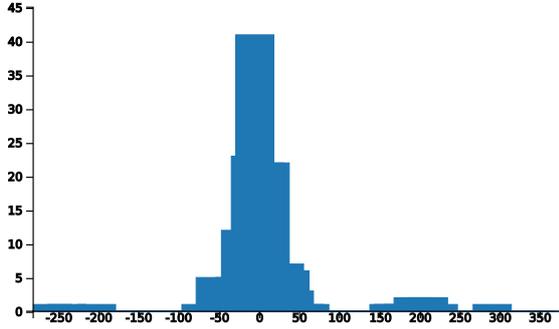
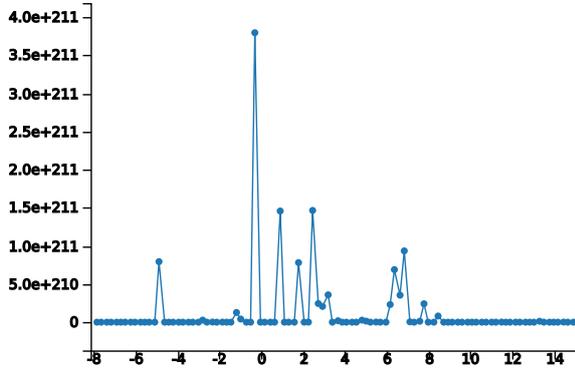
```

draw the prior and the posterior densities respectively using 200 samples (Fig. 8a, 8b).

5.7. Bayesian Estimation

Let us consider a problem proposed on the Anglican [WvdMM14] web site³. 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

³<http://www.robots.ox.ac.uk/~fwood/anglican/examples/viewer/?worksheet=gaussian-posteriors>

(a) Prior density in the `dp_mix.pl` example.(b) Posterior density in the `dp_mix.pl` example.Fig. 8. Representation of the distributions in the `dp_mix.pl` example.

measurements (e.g. at different times), indexed by an integer.

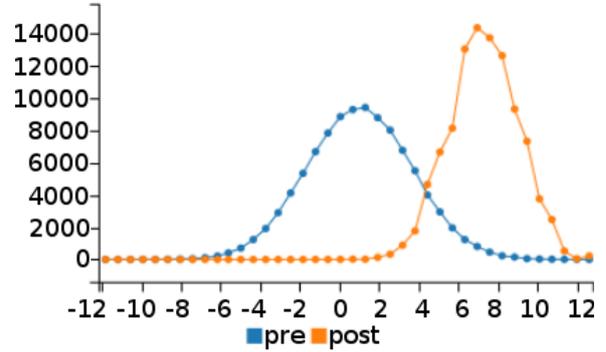
This problem, handled in `gauss_mean_est.pl`, can be modeled with:

```
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) change with respect to the case of no observations? This example shows that the parameters of the distribution atoms can be taken from the probabilistic atoms (`gaussian(X,M,2.0)`) and `value(_,M,X)` respectively). The query

```
?- 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 100,000 samples of the argument `X` of `value(0,X)` before and after the observation of

Fig. 9. Prior and posterior densities in `gauss_mean_est.pl`.

`value(1,9)`, `value(2,8)` and draws the prior and posterior densities of the samples using a line chart. Figure 9 shows the resulting graph where the posterior is clearly peaked at around 7.

5.8. Kalman Filter

The example `kalman_filter.pl` (adapted from [NR14]) encodes a Kalman filter, i.e., a Hidden Markov model with a real value as state and a real value as output.

```
kf(N,0,T) :- init(S), kf_part(0,N,S,0,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).
```

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 so that the same clauses can be used both to sample and to evaluate the weight of the sample on the basis of the 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 (fil-

tering problem)? Likelihood weighting can be 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 $\{V:=\text{NextS}+X\}$ is used to compute V from X and NextS . When weighting, the constraint is used to compute X from V and NextS . The above query can be expressed with

```
?- mc_sample_arg(kf(1,_01,Y),10000,Y,L0),
   mc_lw_sample_arg(kf(1,_02,T),kf(1,[2.5],_T),
                   10000,T,L),
   densities(L0,L,40,Chart).
```

that returns the graph of Figure 10a, from which it is evident that the posterior distribution is peaked around 2.5.

Given a Kalman filter with four observations, the value of the state at the same time points can be sampled by running particle filtering:

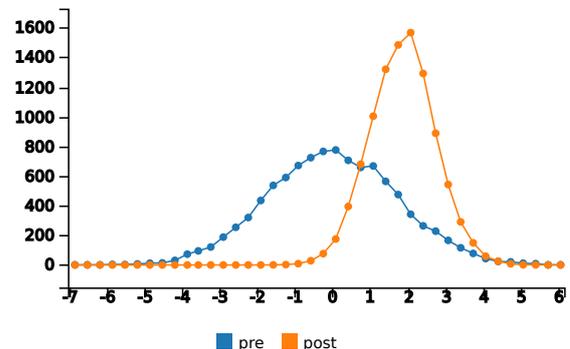
```
?-[01,02,03,04]=[-0.133, -1.183, -3.212, -4.586],
  mc_particle_sample_arg([kf_fin(1,T1),kf_fin(2,T2),
                        kf_fin(3,T3),kf_fin(4,T4)],
                        [kf_o(1,01),kf_o(2,02),
                        kf_o(3,03),kf_o(4,04)],100,[T1,T2,T3,T4],
                        [F1,F2,F3,F4]).
```

The list of samples is returned in $[F1,F2,F3,F4]$, with each element being the sample for a time point.

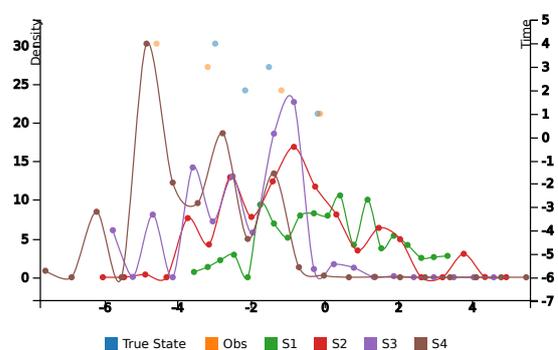
Given the states from which the observations were obtained, Figure 10b shows a graph with the distributions of the state variable at time 1, 2, 3 and 4 (S1, S2, S3, S4, density on the left Y axis) and with the points for the observations and the states with respect to time (time on the right Y axis).

5.9. Stochastic Logic Programs

Stochastic logic programs (SLPs) [Mug00] 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



(a) Prior and posterior densities in `kalman.pl`.



(b) Example of particle filtering in `kalman.pl`.

Fig. 10. Representation of the distributions in the `kalman_filter.pl` example.

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

```
s_as(N):0.2;s_bs(N):0.2;s_a(N):0.3;s_b(N):0.3.
s([a|R],NO):- s_as(NO), N1 is NO+1, s(R,N1).
s([b|R],NO):- s_bs(NO), N1 is NO+1, s(R,N1).
s([a],NO):- s_a(NO).   s([b],NO):- s_b(NO).
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 with 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 11.

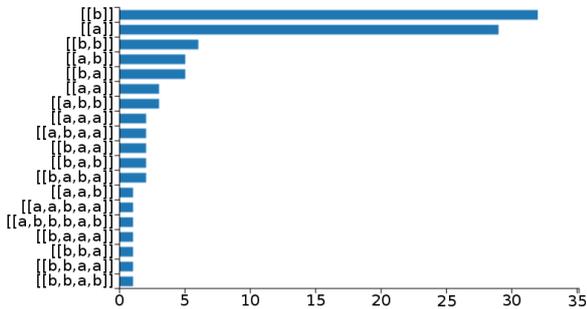


Fig. 11. Samples of sentences of the language defined in `slp.pcfg.pl`.

6. Conclusions

`cplint` on SWISH is a web application for PLP that offers many features, including some that used to be present only in other PP paradigms, such as functional or imperative PP. For example, the possibility of handling hybrid programs, containing both discrete and continuous variables, is relatively novel in PLP and `cplint` on SWISH is the first online system offering it.

`cplint` on SWISH allows users to perform reasoning tasks by using just a web browser, without requiring the installation of a PLP system on their machine, an usually complex 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 in more detail the connection with PP using functional/imperative languages and exploit the techniques developed in that field. We are currently working on supporting a probabilistic extension [ALRZ16] of hybrid logic knowledge bases [MR10], for combining probabilistic logic programming with probabilistic description logics. Moreover, we plan to add the support to tractable languages and causal probability theory. Finally, we are working on the implementation of a new feature which will allow users to exploit also the R language for improving statistical computing functionalities.

A complete online tutorial [RC16] is available at <http://ds.ing.unife.it/~gcota/plptutorial/>.

Acknowledgements

This work was supported by the “GNCS-INDAM”.

References

- [ACRZ16] Marco Alberti, Giuseppe Cota, Fabrizio Riguzzi, and Riccardo Zese. Probabilistic logical inference on the web. In Giovanni Adorni, Stefano Cagnoni, Marco Gori, and Marco Maratea, editors, *AI*IA 2016 Advances in Artificial Intelligence*, volume 10037 of *Lecture Notes in Artificial Intelligence*. Springer, 2016.
- [ALRZ16] Marco Alberti, Evelina Lamma, Fabrizio Riguzzi, and Riccardo Zese. Probabilistic hybrid knowledge bases under the distribution semantics. In Giovanni Adorni, Stefano Cagnoni, Marco Gori, and Marco Maratea, editors, *AI*IA 2016 Advances in Artificial Intelligence*, volume 10037 of *Lecture Notes in Artificial Intelligence*. Springer, 2016.
- [DKT07] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. ProbLog: A probabilistic Prolog and its application in link discovery. In *IJCAI 2007*, volume 7, pages 2462–2467, Palo Alto, California USA, 2007. AAAI Press.
- [DM02] Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *J. Artif. Intell. Res.*, 17:229–264, 2002.
- [DRK15] Luc De Raedt and Angelika Kimmig. Probabilistic (logic) programming concepts. *Mach. Learn.*, 100(1):5–47, 2015.
- [FC90] Robert M Fung and Kuo-Chu Chang. Weighing and integrating evidence for stochastic simulation in bayesian networks. In *Fifth Annual Conference on Uncertainty in Artificial Intelligence*, pages 209–220. North-Holland Publishing Co., 1990.
- [FdBR⁺15] Daan Fierens, Guy Van den Broeck, Joris Renkens, Dimitar Sht. Shterionov, Bernd Gutmann, Ingo Thon, Gerda Janssens, and Luc De Raedt. Inference and learning in probabilistic logic programs using weighted boolean formulas. *Theor. Pract. Log. Prog.*, 15(3):358–401, 2015.
- [GTK⁺11] Bernd Gutmann, Ingo Thon, Angelika Kimmig, Maurice Bruynooghe, and Luc De Raedt. The magic of logical inference in probabilistic programming. *Theor. Pract. Log. Prog.*, 11(4-5):663–680, 2011.
- [IRR12] Muhammad Asiful Islam, CR Ramakrishnan, and IV Ramakrishnan. Inference in probabilistic logic programs with continuous random variables. *Theor. Pract. Log. Prog.*, 12:505–523, 2012.
- [KDD⁺11] Angelika Kimmig, Bart Demoen, Luc De Raedt, Vitor Santos Costa, and Ricardo Rocha. On the implementation of the probabilistic logic programming language ProbLog. *Theor. Pract. Log. Prog.*, 11(2-3):235–262, 2011.
- [KF09] Daphne Koller and Nir Friedman. *Probabilistic Graphical Models: Principles and Techniques*. Adaptive computation and machine

- learning. MIT Press, Cambridge, MA, 2009.
- [MR10] Boris Motik and Riccardo Rosati. Reconciling description logics and rules. *J. ACM*, 57(5):30:1–30:62, June 2010.
- [Mug00] Stephen Muggleton. Learning stochastic logic programs. *Electron. Trans. Artif. Intell.*, 4(B):141–153, 2000.
- [NDLDR16] Davide Nitti, Tinne De Laet, and Luc De Raedt. Probabilistic logic programming for hybrid relational domains. *Mach. Learn.*, 103(3):407–449, 2016.
- [NR14] Arun Nampally and CR Ramakrishnan. Adaptive MCMC-based inference in probabilistic logic programs. *arXiv preprint arXiv:1403.6036*, 2014.
- [Pfe16] Avi Pfeffer. *Practical Probabilistic Programming*. Manning Publications, 2016.
- [RBL⁺16] Fabrizio Riguzzi, Elena Bellodi, Evelina Lamma, Riccardo Zese, and Giuseppe Cota. Probabilistic logic programming on the web. *Software Pract. and Exper.*, 46(10):1381–1396, October 2016.
- [RC16] Fabrizio Riguzzi and Giuseppe Cota. Probabilistic logic programming tutorial. *The Association for Logic Programming Newsletter*, 29(1):1–1, March/April 2016.
- [RD06] Matthew Richardson and Pedro Domingos. Markov logic networks. *Mach. Learn.*, 62(1-2):107–136, 2006.
- [Rig13] Fabrizio Riguzzi. MCINTYRE: A Monte Carlo system for probabilistic logic programming. *Fund. Inform.*, 124(4):521–541, 2013.
- [Rig16] Fabrizio Riguzzi. The distribution semantics for normal programs with function symbols. *Int. J. Approx. Reason.*, 77:1 – 19, 2016.
- [RS11] Fabrizio Riguzzi and Terrance Swift. The PITA system: Tabling and answer subsumption for reasoning under uncertainty. *Theor. Pract. Log. Prog.*, 11(4–5):433–449, 2011.
- [Sat95] Taisuke Sato. A statistical learning method for logic programs with distribution semantics. In Leon Sterling, editor, *ICLP-95*, pages 715–729, Cambridge, Massachusetts, 1995. MIT Press.
- [VN51] John Von Neumann. Various techniques used in connection with random digits. *Nat. Bureau Stand. Appl. Math. Ser.*, 12:36–38, 1951.
- [VVB04] Joost Vennekens, Sofie Verbaeten, and Maurice Bruynooghe. Logic Programs With Annotated Disjunctions. In Bart Demoen and Vladimir Lifschitz, editors, *Logic Programming: 20th International Conference, ICLP 2004, Saint-Malo, France, September 6-10, 2004. Proceedings*, volume 3132 of *LNCS*, pages 431–445, Berlin Heidelberg, Germany, 2004. Springer Berlin Heidelberg.
- [WvdMM14] Frank Wood, Jan Willem van de Meent, and Vikash Mansinghka. A new approach to probabilistic programming inference. In *Proceedings of the 17th International conference on Artificial Intelligence and Statistics*, pages 1024–1032, 2014.