Report for MPRI 2-40: Build your own Probabilistic Programming language

Patte, Sébastien Bassiouny, Mohamed

Email: firstname.lastname@ens-paris-saclay.fr

February 2022

1 Introduction

The goal of this project is to provide an OCaml library for probabilistic programming models. Our target is to build a probabilistic programming language (PPL) as a layer on top of OCaml in order to allow the easy usage of some common probabilistic constructs by providing an implementation of some basic sampling methods, inference methods based on MCMC class of methods, as well as allowing conditioning on input.

2 Exercise 1: Inference by Enumeration

The general idea behind an inference technique is exploring the space of executions of random variables. Inference by enumeration in particular is a process where we aim to find the probability distribution of random variable X given observed evidence e and some hidden variable Y. We implement this method by enumerating all possible executions for a given model.

Additionally, our inference model has (soft) conditioning constructs implemented in order to give users the ability to filter the outcome of the execution to a particular desired subset.

In the context of the basic funny_Bernoulli example given in this exercise we explore the tree of executions using the sample and the assume for conditioning. This means that for each "sample" instruction, we choose 1 element of the binomial distribution and we keep the observed value and continue in the same path. It is worth noting that an alternative technique to bypass this limitation in our simple implementation would be by using a continuation model (CPS) that allows to try a value from the support and trace back to a previous point to explore a different execution path.

To test our implementation, We perform several runs and present the mean obtained from the runs for our implementation and compare it to other basic inference methods like importance sampling and rejection sampling (A more visual representation is provided in appendix A)

sample	Enum	Imp	Rej
0	0	0	X
1	0.333333	0.327793	0.327200
2	0.5	0.506216	0.506500
3	0.166667	0.165991	0.166299

An Additional test of our method using a coin example is provided in the appendix.

3 Exercise 2: Metropolis-Hastings (MH)

The library offers a Metropolis-hastings (MH) implementation as well. While simple inference works fine like we saw in the previous example, it may not perform so well in a multidimensional context. Also, naive enumeration takes time proportional to the number of executions, which is very inefficient. Therefore we need other techniques. Here we present an alternative technique known as MCMC techniques.

Markov Chain Monte–Carlo (MCMC) is a method used to obtain information about distributions. Its popularity comes from the fact that it allows one to characterize a distribution without full knowledge of the distribution by randomly sampling values out of the distribution. It relies mainly on having access to a function that computes the probability density for different samples. The power of the Markov chain

property of the MCMC sampling method allows to have a random walk over the distribution which helps speeding up the convergence to the posterior distribution.

In this project we implement an algorithm that uses the MCMC technique known as "Metropolis-Hasting".

The algorithm itself was proposed by Metropolis et al. and generalized by Hastings.

The general MH iterative procedure that our implementation follows is as follows:

1. Given a position X_i sample a proposal position Y from a transition distribution $q(x_i|X_{i-1})$
2. accept this proposal with probability

$$\alpha = \min(1, \frac{p(X_i, Y_i)}{p(X_{i-1}, Y_{i-1})} \frac{q(x_{i-1}|X_i)}{q(x_i|X_{i-1})})$$

Depending on this α, sometimes we keep the old value! The kernel q is the proposal and it affects the acceptance rate and efficiency of the markov chain.

The algorithm converges eventually (sometimes slowly) to a stationary set of samples from the distribution.

3.1 Multi-Sites Metropolis Hastings

At first we provide a simple variation of MH, for multi-sites.

At each "sample", we factor the current execution score with the sampled value. At each assume we factor the score with $-\infty$ if the condition is false (therefore the score is $-\infty$).

After each execution, the current execution score is added in the "scores" array. Then if score.(i) $\geq$ score.(i) or with probability score.(i) / score.(i-1), we accept the current execution (i.e. return its result), else we return the previous execution result (stored in "old_res") and change the current score by the previous one.

3.2 Single-Site Metropolis Hastings

Additionally, we offer an implementation of Single-Site Metropolis Hastings in CPS.

At each iteration, we choose a random "sample" statement from previous execution, re-sample it and continue the model execution with the new value (using the continuation at this "sample" control point).

After this "new" execution we compare the new score with the old one, like in Multi-site version, but this time the values before the chosen "sample" don't change.

Here is the outcome when we run the funny_Bernoulli example of the previous exercise with both Single-site(SS) and multi-site (MS) models (several runs, taking the average):

sample	0	1	2	3
SS	0	0.342484	0.491503	0.166013
MS	0	0.334714	0.501246	0.164038

4 Extension: Hamiltonian Monte-Carlo (HMC)

It is possible to sample from most distributions using MH algorithm. but it remains nothing more than a random walk. There's no logic or a sense of adaptation on how large the steps(/jumps) of the should be based on the current position.

Indeed, sampling high-dimensional distributions with MH becomes very inefficient in practice. A more efficient method is Hamiltonian Monte Carlo (HMC)

The reason behind this is that when dimensionality increases the region of high likelihood is only a small part of the total area, and the majority of the total area is the area of low likelihood. The effect of this is more visible in MH with higher dimensions as the algorithm may end up returning samples that aren't representative of the distribution. So, how do we patch this problem with likelihood in mind? The answer is by using a method inspired from Physics, known as Hamiltonian Monte-Carlo (HMC).

The general algorithm for HMC is not too dissimilar from MH. The main difference from a technical point of view is that it produces proposals a bit far from the current value with high acceptance probability.

The algorithm initially simulates the trajectory of a particle in an energy field, therefore, it has a momentum P, a U: an amount of energy (simply a function with $-\ln$). Low energy pits represent high likelihood regions. The algorithm goes as follows.

1. We start with a position and a momentum, of the form $p(X)\alpha \exp U$

– repeat for a certain number of steps

2. Sample an initial momentum $P_i(0)$ (for this we need the gradient: $-\log p(x)$)

3. Solve the Hamiltonian dynamic (we use leapfrog to reach a proposed state in our implementation)

4. Perform a metropolis hasting update to either s accept or reject with probability

$$\alpha = \min(1, \frac{\exp(X_i, P_i)}{\exp(X_{i-1}, P_{i-1})})$$

Finally we run our implementation on the funny_Bernoulli example and obtain the following outcome:

0	1	2	3
0	0.342484	0.491503	0.166013

5 Final remarks

Each sampling method has its own powers and its own drawbacks. Depending on the context a particular one might be more suitable than another. Our implementations do not use any advanced techniques beyond implementing by following the simple description of these algorithms. For example we don't take into account a burn-in percentage of the total number of samples.

Indeed, the performance of MH and HMC in practice is far from ideal, it is one of those cases where the theory does not match the practice. We noticed its weakness when dealing with low dimensionality. The outcome according to our tests remains acceptable when the number of variables increases.

The project provides several examples that we saw in class. They are all ready to use and can be found under "./examples".

Among these examples there are Funny_bernoulli, HMM, Laplace, coin, and a triangle sampling.

The latter allows to sample three points at random and draw a triangle according to the sampling outcome. The sampling method can be modified at will. More details on how to use this example can be found in the readme of the project.

Overall, our project offers implementations for various sampling methods and various examples are provided. In the appendix there are some graphs that show the outcome of the various methods for 2 examples (funny_Bernouilli and cannabis) . We invite you to try different models and examples.

To conclude, PPLs in general are a powerful tool when used for what they are intended for. They allow for flexible manipulation for probabilistic models like in the examples implemented in this project. Naturally, it goes without saying they are useful to study a statistical behavior and models, they are not the right tool for writing full-fledged software, which is why we choose our implementation to be an external library, an extra layer on top of a normal full-fledged programming language.

References

[1] Noah D Goodman and Andreas Stuhlmüller. *The Design and Implementation of Probabilistic Programming Languages*. `http://dippl.org`. Accessed: 2022-2-16. 2014.

[2] "Handbook of Markov Chain Monte Carlo". In: (2011). DOI: `10.1201/b10905`. URL: `http://dx.doi.org/10.1201/b10905`.

Appendices

A Visual representation of funny Bernoulli with various sampling methods

In this graph we show all sampling methods available in the library, executed in the funny_Bernoulli example.

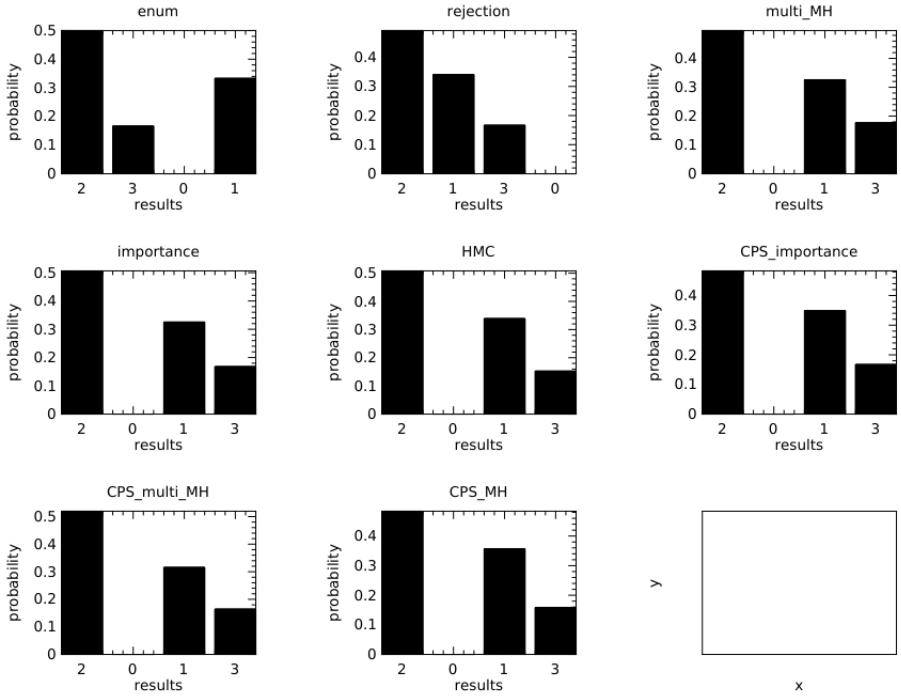

Figure 1: Outcome of Funny_Bernoulli example with various sampling methods

B Cannabis example with basic sampling methods

For further testing of our method, we implement the Cannabis example seen in class.

The model goes as follows:

```
let cannabis prob () =
  let smoke = sample prob (bernoulli ~p:0.6) in
  let coin = sample prob (bernoulli ~p:0.5) in
  assume prob (coin=1 || smoke=1);
  smoke
```

And the outcome of various sampling methods including our Enumeration and our MH algorithms are :

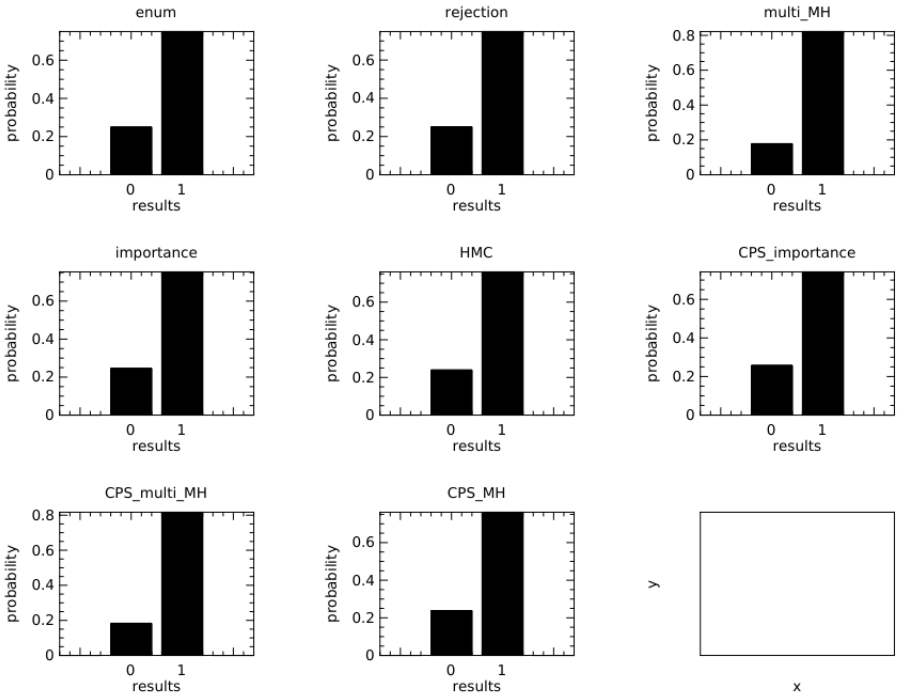

Figure 2: Outcome of cannabis example with various sampling methods