Project report : Probabilistic programming

Emile Trotignon and Simon Coumes

February 18, 2022

1 Introduction

For this project we have chosen to implement a probabilistic inference framework as a library in Ocaml. This report takes most of the base knowledge about what we do in this project for granted. We try with reasonable success not to repeat too much of your own lessons back to you. Hopefuly nothing was made unclear as a result.

A few words about vocabulary. In this document we consider separately the code that is actually written and executed and the "corresponding probabilistic program". The later is the program whose set of executions define the distributions we build. In generality we can consider that the user writes code that describes a probabilistic program. This is done through the syntax we define. An execution of said program is an assignment of a value to each variable that is "sampled" in the probabilistic program. This defines a distribution on all possible outputs of the program (which the user can skew with statements such as $assume$ and $factor$). Our goal is to compute or approximate this distribution.

2 Discrete inference by enumeration

2.1 Overview

We build our language in Ocaml with a library offering a recursive structure and then allow the use of syntactic sugar to improve user comfort. In this section, we restrain ourselves to discrete distributions with a finite support and present our implementation of an inference method by exhaustive enumeration. The use of an exhaustive enumeration of all possible executions is made possible by the fact that all supports are finite. All of this is found in the "finite" sub-library.

2.2 A recursive structure

Our approach is centered on the introduction of a "model" type, which we inductively defined as follows :

$$Model_1 ::= return\ e \mid (assume\ b),\ Model_2 \mid a = sample(d)\ in\ Model_2 \mid (factor\ i),\ Model_2 \qquad (1)$$

With e an expression, b an expression that has type bool, i an expression that has type float, d an expression that has the type of a distribution, and a a variable name. We require that every expression be well formed with regard to free variables. ie all of their free variables are bound either by normal Ocaml code or by a sample by the time they are encountered in the tree this forms.

When we move to code, this gives the AST (Abstract Syntax Tree) from figure 1.

You may notice the `Sample` constructor defines an existantial type, in `á`. This is because two models that represent a distribution with values of the same type should have the same type, even if they sample distributions with values of different types.

Perhaps more interesting, we can see how this can be used to define an example model (see figure 2). Please note that the bindings happening in samples in equation 1 are replace by functions here, which will be called during evaluation. This approach will present multiples advantages and inconvenients in the rest of this project.

As shown in figure 2 this code is not very convenient to use. We will see in subsection 2.4 how we used syntactic sugar to improve on this.

```
      type 'a t =
  | Return : 'a -> 'a t
  | Assume : (bool * 'a t) -> 'a t
  | Factor : (Prob.t * 'a t) -> 'a t
  | Sample : ('a Dist.t * ('a -> 'b t)) -> 'b t
```

Figure 1: Type definition for finite distributions

```
let funny_bernoulli_ugly =
    Model.(
      Sample
        ( sample (Dist.bernoulli_int ~p:0.5)
        , fun a ->
            Sample
              ( sample (Dist.bernoulli_int ~p:0.5)
              , fun b ->
                  Sample
                    ( sample (Dist.bernoulli_int ~p:0.5)
                    , fun (c : int) ->
                        Assume (a = 1 || b = 1, Return (a + b + c))))))
```

Figure 2: A first code example with an unwieldy syntax

2.3 Recursive evaluation

We now want to perform a process of inference by exhaustive enumeration. This means we want to look at each possible run of the probabilistic program our code describes and compute the corresponding score, avoiding all runs that fail an assume statement. The result associated to a run is a pair $(value, score)$.

The recursive structure presented above allows this process to proceed recursively. For each *sample* statement, we can enumerate all possible outputs (each element of the support of the distribution we sample). For each possibility we bind the value to the corresponding variable and make a recursive call. Each call gives us a list of results. We then output the concatenation of all those lists.

An *assume* that evaluates to *true* passes on the list of result we get by evaluating its corresponding sub-model while an *assume* that evaluate to false returns the empty list (there are no corresponding executions). [1]

Likewise, $factor$ statements pass on their values while changing the scores and *return* statement give a value together with the default score.

Remark 1 *For simple cases, this can be considered equivalent to looking at all possible results for each element in the cartesian product of the sets of possible results of all sample statements in the code.*

2.4 Syntactic sugar

We introduced a let operator that allows one to sample distribution in a nicer way.

3 Metropolis Hasting

We now discuss the second method of inference we implemented, Metropolis Hasting. This corresponds to the "continuous" sub-library of our implementation.

Like for the previous inference method we will not recall the general idea of the method and will instead only describe the points of our specific approach we consider the most relevant.

[1] Remark that in the case of an *assume* that returns false we do not explore the rest of the execution.

```
Model.(
  let* a = Dist.bernoulli_int ~p:0.5 in
  let* b = Dist.bernoulli_int ~p:0.5 in
  let* c = Dist.bernoulli_int ~p:0.5 in
  assume (a = 1 || b = 1) (return (a + b + c)))
```

Figure 3: A reformulation of the previous example with a better syntax

3.1 Writing continuous distributions

As we move to our second method of inference, Metropolis Hasting, we also start to include continuous distributions. While a finite support distribution is simple enough (a list of pairs of values and probabilities), we need a slightly subtler representation for continuous distributions. For our purposes, they are a pair of two functions. A function *sample* allows to draw a value and a function *logpdf* which is the log of the probability density function of the distribution.

From now on, when we speak of a distribution, we mean either a finite distribution or a continuous one. Note that these can be mixed in the same code. Concretely, this means we implemented a variant type in Ocaml with a constructor for each case.

3.2 A first execution

The user has access to the same syntax as we presented in subsection 2.4. At first, we run the probabilistic program that this defines once. For each probabilistic statement we encounter, we iteratively compute the "current score" of the run (more on how this is done in subsection 3.4). If the statement is a *sample* we also store both the current continuation and the current score for later use.

When the execution is over this leaves us with a trace of the execution (all the pairs of a continuation and a score that we computed) and with a final value and score.

Remark 2 *Obtaining continuations is easy for us because the underlying structure of our code is still akin to the recursive definition we presented for the previous inference method. Hence, the "continuations" associated to a sample were already present in our approach to begin with.*

3.3 Subsequent executions

We now consider a hashtable that associates to each value we previously encountered a single score. We choose at random a continuation we stored and restart execution at this point, taking the score we stored with this continuation as the current score. All probabilistic statements following the corresponding point in the probabilistic program are encountered anew (hence all random values after this point are re-sampled). We also create a new list of pairs of computations and scores, which differs from the previous one after the continuation with which we started this new execution. This creates a new trace.

We compare the score of this new trace with the previous one and choose which one to keep. The formula we use is that we keep the new trace with probability $\frac{W_{new}}{W_{old}}$. With W_{new} (resp W_{old}) the score of the new (resp old) trace.

If we kept the new trace we add the corresponding pair $(value, score)$ to the hashtable [2] . We then iterate this step with the trace we kept (which can be equal to the old one).

3.4 Choices

The explanation above does not say :

- How to iteratively compute the score of a trace.

[2]Please remark that a single hashtable can contain multiple pairs with the same keys and that all the pairs can still be enumerated. We do not "lose" anything in case of duplicates.

- How to compute the final distribution using our hashtable.

The answer to the first question is that we multiply [3] the scores given by each probabilistic operator we encounter. In the case of the $factor$ statement the associated score is straightforward. Both the $observe$ and $assume$ statements can be seen as specific cases of factor.

This leaves us the *sample* statement, which gives as a score the probability associated by the sampled distribution to the value that was drawn.

In most cases the answer to the second question is easy. If the hashtable contains no two entries with the same value we can output the finite distribution that associates to each value the corresponding score.

However when there are duplicates we offer two options, neither of which give us exactly what we would like in all cases. The first is to add the scores of all pairs that have the same value. The second is to compute the average of the scores. Both option are left available to the user in our implementation.

4 The tests

We have implemented various tests and examples from the subject and class. All these examples are available in the bin directory and can each be executed with the command "dune exec bin/testname.exe", with testname the name of the relevant test.

In no particular order :

- "coin" implements the example seen in class where we guess the bias of a coin based on a series of tosses (assuming an uniform prior).
- "laplace" implements the example seen in class with the same name.
- Likewise for "funny_bernouilli".

[3] All our explanations ignore the fact that we instead store the log of the score to avoid rounding errors.