Project report : Probabilistic programming

Emile Trotignon and Simon Coumes

February 17, 2022

1 Discrete inference by enumeration

1.1 Overview

Simon : Draft phase

We build our language in Ocaml with a library offering a recursive structure and then allow the use of syntactic sugar to improve use comfort. For this first approach, we restrain ourselves to discrete distributions with a finite support. All of this is found in the "finite" sublibrary.

1.2 A recursive structure

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

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

With e an expression, b an expression that has type bool, i an expression that has type int, 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 by the time they are encountered in the tree this forms either by normal Ocaml code or by a sample.

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

```
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

Note that the construction found in our models is existential. **Simon :** Let Emile word this.

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 inconvenient in the rest of this project.

As shown in figure 2 this code is not very convenient to use. We will see in subsection 1.4 how we used syntaxic sugar to improve on this.

1.3 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 describe and compute the corresponding score, erasing all runs that fail an assume statement. The result associated to a run is a pair $(value, score)$.

```
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 unwieldly syntax

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 submodel while an assume that evaluate to false returns the empty list (there are no corresponding executions).

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 probabilistic statements in the code.*

1.4 Syntactic sugar

Yada. Show snippets of example code.

2 Metropolis Hasting

2.1 Writing continuous distributions

A continuous distribution is either a finite distribution or a pair (sample, logpdf). We implement only one mode of inference for these distributions, Metropolis Hasting. This is found in the continuous sublibrary.

2.2 General theory

Simon : Re-discuss this first

2.3 We already have continuations

Because of the way we wrote the previous logic, it can be followed loosely here while still allowing easy access to continuation.

3 The tests

Simon : Not now