Efficient Inference in Probabilistic Computing
M.Eng Propsal

Jeff Wu

Abstract

1 Introduction

Probabilistic programming languages are generalizations of programming languages, in which procedures are replaced with random procedures that induce distributions. Probabilistic computing thus allows for easy description and manipulation of probability distributions, letting one describe classical AI models in compact ways.

A core operation of probabilistic programming languages is inference, which is, in general, a difficult computational problem [?]. However, Markov chain Monte Carlo (MCMC) methods converge quite quickly to the posterior distribution for a large class of inference problems. Much effort has been devoted to studying in which cases this inference is efficient, and how to make it so.

1.1 subsection

2 Language description

2.1 Values and XRPs

A **value** is a valid binding for a variable, in our language. For example, integers, floats, and booleans are valid values, as are procedures — functions which take in arguments and return values.

Our language allows for definition of something more general than random procedures, while retaining the ability to perform inference. It allows for what we call an Exchangeable Random Procedure (XRP).

An XRP is like a random procedure, except that different applications of it are not guaranteed to be independent. An XRP is allowed to maintain an internal state, which can be modified. However, an XRP must maintain the exchangeability condition, which says that applications of the XRP gives an exchangeable sequence of random variables.

The exchangeable sequence

2.2 Expressions and Environments

Expressions are syntactic building blocks which are evaluated into values. An expression may be one of the following:

- Atomic expressions:
  - A value itself.
  - A variable name.
- Non-atomic expressions
  - An application of an expression to a list of argument expressions.
  - A function, consisting of some variable arguments, and a body expression.
  - An if statement, consisting of a branching expression, and two child expressions.
  - An operator statement, consisting of an operator (e.g. $==$, $<$, $+$, $\times$, `AND`, `OR`, `NOT`) and a list of argument expressions.

Expressions are evaluated relative to an **environment**, in which variable names may be bound to values.

Evaluation happens recursively, resulting in return values. When an invalid expression is given (e.g. the first expression in an application does not result in an XRP or procedure, or we call the equality operator with three arguments), an error is thrown.

2.3 Directives

There are four primitive operations, called **directives**, which the language supports. They are:

1. `assume(name, expression)`

   Binds `name`, a string name, to the expression `expression`.

2. `observe(expression, value)` :

   Observes the expression `expression` to have value `value`.

3. `sample(expression)` :

   Evaluates to sample a value for the expression `expression`.

4. `infer()` :

   Runs a single step of the Markov Chain.

2.4 Architecture of inference

We now describe the way in which inference is implemented.

The idea is to run Metropolis-Hastings on the space of all possible executions of the program.

Our proposal density does the following:

1. Pick, at random, one of the random choices made, at any point in the program's execution history. Rerun the entire program, using the same randomness.

2. When we reach the random choice chosen above, resample a value for it. If this causes us to evaluate different branches of the code (or to evaluate procedures called with different arguments), simply run these new branches, and also undo the evaluation of previously-evaluated branches which are no longer evaluated.

3. For all observations, use the outermost noise to force.

Notice that if there is no noise in our observations, this algorithm does not necessarily mix in finite time.

Consider the following example:

Some executions are more likely than others, and so

Here pseudocode for a single iteration:

```
stack = globals.db.random_stack()
(xrp, val, prob, args) = globals.db.get(stack)

old_p = globals.db.prob()
old_to_new_q = - math.log(globals.db.count)

globals.db.save()

globals.db.remove(stack)
new_val = xrp.apply(args)

if val == new_val:
  globals.db.insert(stack, xrp, new_val, args)
  return
globals.db.insert(stack, xrp, new_val, args)

rerun(False)
new_p = globals.db.prob()
new_to_old_q = -math.log(globals.db.count)
old_to_new_q += globals.db.eval_p
new_to_old_q += globals.db.uneval_p
if old_p * old_to_new_q > 0:
  p = random.random()
  if new_p + new_to_old_q - old_p - old_to_new_q < math.log(p):
    globals.db.restore()
globals.db.save()
```

References