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()
```

3 Test Cases

In order to ensure the language was working properly, a suite of test cases was developed.

One of the primary techniques used for testing was "following the prior", using the function `follow_prior(variable, niters, burnin)`. To follow the prior, we draw `niters` samples from the prior, and run each of them `burnin` steps in the random walk. We then view the resulting distribution on `variable`.

Here are some of the more illustrative and interesting test cases.

3.1 Testing a tricky coin

Consider the following scenario: There is a coin that is fair with probability 50%. The rest of the time, its weight is drawn uniformly from the interval [0, 1]. The coin is flipped, and we observe it to be heads `nheads` times. We then infer the posterior probability that the coin was fair.

```
noise_level = .001
nheads = 1

assume('weight', beta(1, 1))
assume('tricky-coin', function([], bernoulli('weight')))
assume('fair-coin', function([], bernoulli(0.5)))
assume('is-fair', bernoulli(0.5))
assume('coin', ifelse('is-fair', 'fair-coin', 'tricky-coin'))

for i in xrange(nheads):
  observe(bernoulli_noise(apply('coin'), noise_level), True)

follow_prior('is-fair', 10000, 1000)
```

We should be able to predict the results of running this program for different values of `nheads`. First we compute the probability that the coin comes up heads n times, given that it is tricky. The probability is simply $\beta(1, n+1) = \frac{1}{n+1}$. Thus the probability the coin is fair (not tricky), given it came up heads n times, is, by Baye's law

$$\frac{\frac{1}{2} \cdot \frac{1}{2^n}}{\frac{1}{2} \cdot \frac{1}{n+1} + \frac{1}{2} \cdot \frac{1}{2^n}} = \frac{n+1}{2^n + n + 1}.$$

Here are our results, when running `follow_prior('is-fair', 10000, 1000)`, for the percentage of times the coin was fair:

`nheads`	Predicted	Actual
0	0.5	0.4969
1	0.5	0.4964
2	0.429	0.436
3	0.333	0.331
4		0.
5		0.

3.2 Bayes nets

We first test that inference works, by considering a number of test cases. A classic inference problem which is well understood, is inference in Bayesian networks. Bayesian networks are incredibly easy to describe in our language. Here are some simple examples of inference giving the correct answer in a Bayesian network.

3.2.1 Sprinkler net

Let's start with a simple example, to get familiar with our language. Here's a definition of a bayesian network with just two nodes.

```
>>> assume('cloudy', bernoulli(0.5))
>>> assume('sprinkler', ifelse('cloudy', bernoulli(0.1), bernoulli(0.5)))
```

We then observe that the sprinkler is on. Worlds in which the sprinkler is on are weighted as 100 times more likely than ones in which the sprinkler is off.

```
>>> noise_level = .01
>>> sprinkler_ob = observe(bernoulli_noise('sprinkler', noise_level), True)
```

Now, let's try inferring the weather. We follow the prior 10000 times, going 50 steps each time.

```
>>> follow_prior('cloudy', 10000, 50)}
{False: 0.82010000000000005, True: 0.1799}
```

This is close to the value of $\frac{5}{6} = 0.8\overline{333}$ False, which is what we'd get if there was noise in our observation. However, there is still a number of worlds in which the sprinkler was actually on. In those worlds, the weather was 50/50. Thus we should've expect the answer to be on the order of 0.01 lower.

Now, let's suppose we didn't observe the sprinkler being on, after all.

```
>>> forget(sprinkler_ob)
```

So it should be the case that it's cloudy exactly half the time.

```
>>> follow_prior('cloudy', 10000, 50)}
{False: 0.50429999999999997, True: 0.49569999999999997}
```

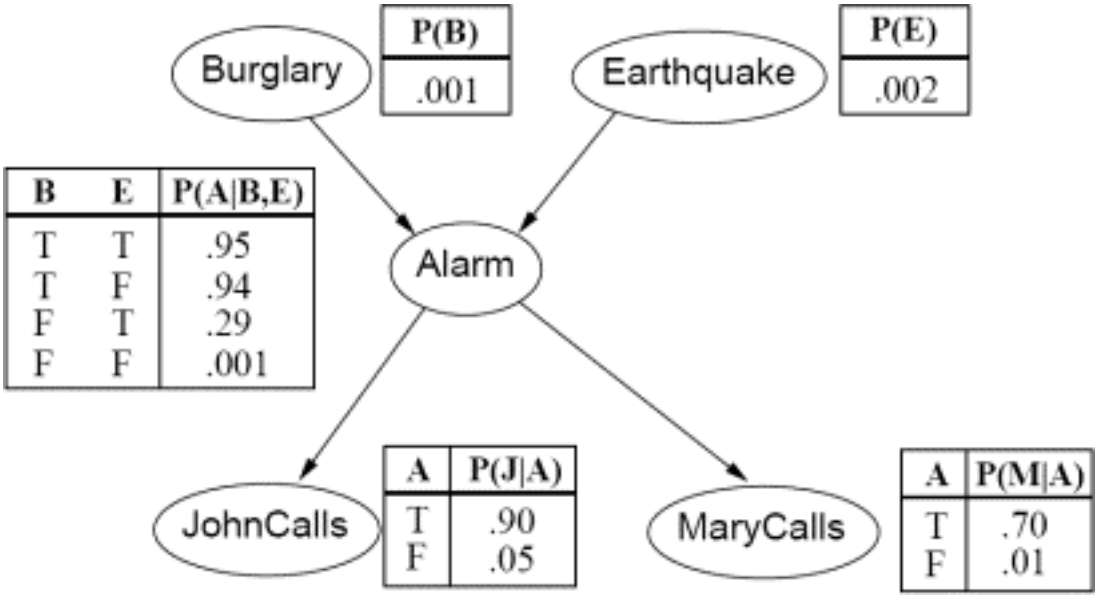
Now, we re-observe the sprinkler to be on. This time, we are much more sure.

```
>>> noise_level = .001
>>> sprinkler_ob = observe(bernoulli_noise('sprinkler', noise_level), True)
>>> follow_prior('cloudy', 10000, 50)}
{False: 0.83999999999999997, True: 0.16}
```

We see that our answer is even closer to the value of $\frac{5}{6} = 0.8\overline{333}$ False, now.

3.2.2 Alarm net

Here's a more complicated Bayesian network, which is given as an example in the classic AI textbook Artificial Intelligence (A Modern Approach).



Let's define this Bayesian network.

```
>>> assume('burglary', bernoulli(0.001))
>>> assume('earthquake', bernoulli(0.002))
>>> assume('alarm', ifelse('burglary', ifelse('earthquake', bernoulli(0.95), bernoulli(0.94)), \
...                                    ifelse('earthquake', bernoulli(0.29), bernoulli(0.001))))
>>> assume('johnCalls', ifelse('alarm',  bernoulli(0.9), bernoulli(0.05)))
>>> assume('maryCalls', ifelse('alarm',  bernoulli(0.7), bernoulli(0.01)))
```

Let's try a couple inference problems.

```
>>> follow_prior('alarm', 1000, 100) # should give 0.002516 True
{False: 0.99739999999999995, True: 0.0025999999999999999}
>>>
>>> noise_level = .001
>>> mary_ob = observe(bernoulli_noise('maryCalls', noise_level), True)
>>> follow_prior('johnCalls', 1000, 100) # should give 0.177577 True
{False: 0.91620000000000001, True: 0.083799999999999999}
>>>
>>> forget(mary_ob)
>>> burglary_ob = observe(bernoulli_noise(negation('burglary'), noise_level), True)
>>> follow_prior('johnCalls', 1000, 100) # should give 0.051343 True
{False: 0.94710000000000005, True: 0.052900000000000003}
```

The first and third inferences were on the mark, but the second one was off by a factor of two! The explanation is that Mary calls very rarely calls. Indeed, she only calls about 0.01 of the time, since the alarm almost never sounds. This is rare enough that our observation that she called is

reasonably likely to be wrong. In worlds where she doesn't call, John also tends not to call, thus accounting for the large dip.

To verify that this is the explanation, we can alter the probabilities so that Mary calling is more likely. Here is an example of this being done.

```
>>> reset()
>>>
>>> assume('burglary', bernoulli(0.1))
>>> assume('earthquake', bernoulli(0.2))
>>> assume('alarm', ifelse('burglary', ifelse('earthquake', bernoulli(0.95), bernoulli(0.94)), \
...                                    ifelse('earthquake', bernoulli(0.29), bernoulli(0.10))))
>>> assume('johnCalls', ifelse('alarm',  bernoulli(0.9), bernoulli(0.5)))
>>> assume('maryCalls', ifelse('alarm',  bernoulli(0.7), bernoulli(0.1)))
>>>
>>> follow_prior('alarm', 1000, 100) # should give 0.218400 True
{False: 0.77959999999999996, True: 0.22040000000000001}
>>>
>>> noise_level = .001
>>> mary_ob = observe(bernoulli_noise('maryCalls', noise_level), True)
>>> follow_prior('johnCalls', 1000, 100) # should give 0.764681 True
{False: 0.2432, True: 0.75680000000000003}
>>>
>>> forget(mary_ob)
>>> burglary_ob = observe(bernoulli_noise(negation('burglary'), noise_level), True)
>>> follow_prior('johnCalls', 1000, 100) # should give 0.561333 True
{False: 0.44290000000000002, True: 0.55710000000000004}
```

Success!

3.3 Testing xor

Consider the following program, where we simply draw two booleans `a` and `b`, and observe that their xor is `True`:

```
>>> p, q = 0.6, 0.4
>>> noise_level = .01
>>>
>>> assume('a', bernoulli(p))
>>> assume('b', bernoulli(q))
>>> assume('c', (var('a') & ~var('b')) |(~var('a') & var('b')))
>>>
>>> follow_prior('a', 10000, 100) # should be 0.60 True
{False: 0.3977, True: 0.60229999999999995}
>>>
>>> xor_ob = observe(bernoulli_noise('c', noise_level), True)
>>> follow_prior('a', 10000, 100) # should be 0.69 True
{False: 0.3342, True: 0.66579999999999995}
```

This second inference is significantly off, and it's not the case that our observation is particularly unlikely. Here, the burn-in was not enough. Notice that in order to mix, the program must walk over states which contradict the observed values.

Suppose we are in the state {a:True, b:True}. Then, the random walk is highly discouraged from entering either of the two adjacent states {a:True, b:False} and {a:False, b:True}, since worlds in which a⊕b is False are weighted against, by a factor of 100.

Let's estimate the amount of burn-in it should take for this random walk to mix. Suppose we are in a state where a⊕b is True. About half the time, we will attempt to enter one of the two adjacent states (the other half of the time, we will keep the same value for that flip). Entering this state will occur with probability roughly $\frac{1}{100}$. Once we are in such a state, we are given an opportunity to enter either of the two a⊕b being True states, so we have successfully mixed.

Thus we can roughly model this as having a $\frac{1}{200}$ chance of mixing properly, for each iteration. Thus if we have a burn-in of $200 \cdot x$, there is roughly a $1 - \frac{1}{e^x}$ chance of mixing. Here are empirical results of burnin against follow_prior('a', 10000, burnin) results:

burnin	follow_prior('a', 10000, burnin) percentage True
100	0.66579999999999995
200	0.67869999999999997
300	0.68200000000000005
400	0.68489999999999995
500	0.68759999999999999

These results are slightly better than our prediction, but confirm the overall phenomenon. Because of the noise, we can't expect it to ever converge to exactly 0.69. Instead, we may hope for it to converge to something like $0.01 \cdot 0.60 + 0.99 \cdot 0.69 = 0.6891$.

3.4 Testing a decaying atom

3.5 Testing mem

Here, we test that the memoization procedure works.
Let's first write the fibonacci function, in the naive way:

```
>>> fibonacci_expr = function('x', ifelse(var('x')<=1, 1, \
...                   apply('fibonacci', var('x')-1) + apply('fibonacci', var('x')-2)))
>>> assume('fibonacci', fibonacci_expr)
```

Of course, evaluating fibonacci in this manner is an exponential time operation:

```
>>> t = time(); sample(apply('fibonacci', 20)); time() - t
10946
1.76103687286
```

Mem is an XRP which, when applied to (possibly probabilistic) functions, returns a version of the function which is memoized. That is, function calls are remembered, so that if a function is called with the same arguments, it does not need to recompute. Here is an example of mem being used.

```
>>> assume('bad_mem_fibonacci', mem('fibonacci'))
>>>
>>> t = time(); sample(apply('bad_mem_fibonacci', 20)); time() - t
10946
```

```
1.90201187134
>>> t = time(); sample(apply('bad_mem_fibonacci', 20)); time() - t
10946
0.00019097328186
```

Notice that the second call to this mem'd fibonacci is much faster, since mem remembers the value. However, the first call is just as slow as before. Since Fibonacci is recursive, we really want to memoize all the recursive subcalls as well. The canonical introduction to dynamic programming shows how Fibonacci can be computed in linear time this way. We can write the program easily:

```
>>> mem_fibonacci_expr = function('x', ifelse(var('x')<=1, 1, \
...                     apply('mem_fibonacci', var('x')-1) + apply('mem_fibonacci', var('x')-2)))
>>> assume('mem_fibonacci', mem(mem_fibonacci_expr))
>>>
>>> t = time(); sample(apply('mem_fibonacci', 20)); time() - t
10946
0.0271570682526
>>> t = time(); sample(apply('mem_fibonacci', 20)); time() - t
10946
0.000293016433716
```

3.6 Testing DPMem

Let's now implement the Dirichlet process.

```
>>> sticks_expr = mem(function('j', beta(1, 'concentration2')))
>>> atoms_expr = mem(function('j', apply('basemeasure2')))
>>> loop_body_expr = function('j', ifelse(bernoulli(apply('sticks', 'j')), apply('atoms', 'j'), \
...                   apply(apply('loophelper', ['concentration2', 'basemeasure2']), var('j')+1)))
>>> loop_expr = apply(function(['sticks', 'atoms'], loop_body_expr), [sticks_expr , atoms_expr])
>>> assume('loophelper', function(['concentration2', 'basemeasure2'], loop_expr))
>>> assume( 'DP', function(['concentration', 'basemeasure'], \
...                         apply(apply('loophelper', ['concentration', 'basemeasure']), 1)))
```

We can now use the Dirichlet process to create something we call DPmem. DPmem is a generalization of mem which sometimes returns memoized values, and sometimes resamples new values.

```
"""DEFINITION OF DPMEM"""
restaurants_expr = mem(function('args', apply('DP', ['alpha', function([], apply('proc', 'args'))])))
assume('DPmem', function(['alpha', 'proc'], function('args', apply(restaurants_expr, 'args'))))


"""TESTING DPMEM"""

concentration = 1 # when close to 0, just mem.  when close to infinity, sample
assume('DPmemflip', apply('DPmem', [concentration, function(['x'], bernoulli(0.5))]))
print [sample(apply('DPmemflip', 5)) for i in xrange(10)]

print "\n TESTING GAUSSIAN MIXTURE MODEL\n"
```

```
assume('concentration', gaussian(1, 0.2)) # use vague-gamma?
assume('expected-mean', gaussian(0, 1))
assume('expected-variance', gaussian(0, 1))
assume('gen-cluster-mean', gaussian(0, 1))
assume('get-datapoint', mem( function(['id'], gaussian('gen-cluster-mean', 1.0))))
assume('outer-noise', gaussian(1, 0.2)) # use vague-gamma?

observe(gaussian(apply('get-datapoint', 0), 'outer-noise'), 1.3)
observe(gaussian(apply('get-datapoint', 0), 'outer-noise'), 1.2)

t = time()
print format(get_pdf(follow_prior('expected-mean', 100, 30), -4, 4, .5), '%0.2f')
print 'time taken', time() - t

#concentration = 1
#uniform_base_measure = uniform_no_args_XRP(2)
#print [sample(apply('DP', [concentration, uniform_base_measure])) for i in xrange(10)]
#expr = beta_bernoulli_1()
```

DPmem can be used, for example, to do mixture modeling. The idea is that we should sometimes

References