Implementing a stochastic programming language

Josh Burdick

May 3, 2002

Graphical models unify several probabilistic inference algorithms that were developed separately. Stochastic programs are a further attempt to solve several probabilistic inference problems at the same time.

I implemented a stochastic programming language based on the language described by Koller, McAllester, and Pfeffer. I first tried to implement this in Java, using `JLex` and `javacup`, which are Java analogs of `lex` and `yacc`, to deal with parsing.

It was clear that performance was going to be a problem. One method for speeding up an interpreted program is partial evaluation. Jørgensen has used a Scheme partial evaluator (Similix) to compile a lazy variant of Scheme. Therefore, I switched to implementing in Scheme.

1 Implementation in Java

The language mostly follows Koller *et al.*

One major difference is that we omit the constructors from the language. Instead, there are named constants, and an equality test for those constants. By convention, constants start with capital letters. The expression `x = y` evaluates to `True` if `x` and `y` reduce to the same constant, and `False` otherwise.

From this, records can be coded up as "functions which expect a constant". The record

```
{ A = True; B = False; }
```

can be coded up as

```
\label.
  if (label = A) True
    if (label = B) False
    Unknown_label
```

Datatypes can also be coded up, although doing so starts to get syntactically clumsy.

Another difference is that functions (lambda abstractions) are written as

```
\x. \y. x
```

1.1 Example: the alarm network

This is the version of the standard "alarm network" that we used in homework 7, problem 1.

```
let if = \c. \t. \f. { True = t; False = f; } c;

let not = \a. if a False True;
let and = \a. \b. if a b False;
let or = \a. \b. if a True b;
let implies = \a. \b. if a b True;

let earthquake = flip(0.002);
let burglary = flip(0.001);
let alarm = if earthquake
(if burglary flip(0.95) flip(0.29))
(if burglary flip(0.94) flip(0.001));
let john = if alarm flip(0.90) flip(0.05);
let mary = if alarm flip(0.70) flip(0.01);

if alarm
(if earthquake Alarm_and_earthquake Alarm_and_not_earthquake)
Not_alarm
```

This outputs

```
[-7.450209178206304 : Alarm_and_earthquake
, -6.247584905392295 : Alarm_and_not_earthquake
, -0.0025196434779503063 : Not_alarm
]
```

from which we can calculate that P(Earthquake|Alarm) = 0.23.

1.2 Bugs

Network simplification isn't implemented correctly.

2 Implementation in Scheme

The language attempts to follow both Koller *et al* and Scheme as much as possible. It only includes a tiny fraction of R4RS Scheme.

Stochastic programs are represented as quoted Scheme lists. This removes the need to implement a parser, which is a significant simplification.

Applications, `lambda` abstractions and variables are as in Scheme.

2.1 `flip`

`(flip` p`)`
evaluates to `#t` with probability p, and evaluates to `#f` with probability $1-p$.

2.2 `let`

The `let` statement allows mutually recursive bindings, like Scheme's `letrec` statement.

3 Bugs

A bug that I haven't fixed yet, is that the Scheme version doesn't correctly keep correlations from shared random variables. For example, consider running the following query with the alarm network from class (which is included below.)

```
(if alarm (if earthquake "a-and-e" "a-and-not-e") "not-a")
```

From the relative values of `"a-and-e"` and `"a-and-not-e"`, we can compute P(Earthquake|Alarm), which should be 0.23. However, currently the program "forgets" the value of `earthquake`, and incorrectly returns a tiny probability.

A minimal example illustrating the problem is

```
'((a . (flip 0.5))
  (b . (iff a a)))
```

where we assume `iff` has been defined to return `#t` if its (Boolean) arguments are equal. `b` should always be `#t`.

As noted in Koller *et al*, we need to make sure that the value of a given variable is stored in the environment, and not recomputed. This is analogous to the "fully sharing" nature of Haskell, in which delayed thunks are only evaluated once.

This worked in the Java version because `let x = flip(0.5); b` was translated to a redex `(\x. b) flip(0.5)` in the usual way. Then, since leftmost-innermost reduction was implemented correctly, the `flip` statement was only evaluated just before it was substituted in, and that result (either `#t` or `#f`) was used everywhere that variable occurred.

Also, network simplification isn't implemented correctly. A sketch of the helper functions needed is in `seenby`.

3.1 Example

Here is the same alarm network example, written in the syntax that the Scheme stochastic interpreter expects.

```
(define alarm_network
  '((earthquake . (flip 0.002))
    (burglar . (flip 0.001))
    (alarm . (if earthquake
       (if burglar
   (flip 0.95)
   (flip 0.29))
       (if burglar
   (flip 0.94)
   (flip 0.001))))
    (john . (if alarm (flip 0.90) (flip 0.05)))
    (mary . (if alarm (flip 0.70) (flip 0.01)))))
```

Some queries work, but as noted above, there's a bug which causes correlations due to shared variables to be lost.

4 Future work

Both versions of this need work. I think the Scheme version, though buggier, is simpler, and in the long run, would be easier to maintain, although the Java version would be better if nice input syntax, and display of results, were priorities.

The interpreter written in Scheme was intended to be used with the Similix partial evaluator. I haven't yet tried using this, though.

This might have been easier to implement in a lazy functional language such as Haskell. However, Scheme compilers are simpler, and there's a wider variety of them to choose from. Also, Scheme seems relatively well suited for metaprogramming; Similix seems to cover most of Scheme, but there only seem to be partial evaluators for small subsets of Haskell.

The current interpreter only computes the distribution over the outputs of a given program. A useful extension would be to use the expectation-maximization algorithm to increase the likelihood of the program producing some output(s).

4.1 Approximation

Since exact Bayesian inference is often intractable, methods for approximating the distribution of a program would be useful.

One possibility would be, when merging distributions, to only keep the k most likely possibilities. This is analogous to the difference between the forward-backward algorithm, which keeps track of all possible paths through an HMM, and the Viterbi algorithm, which only keeps track of the most likely path. Of course, it wouldn't help much in networks in which most variables were Boolean (preserving the top two possibilities would then preserve the entire distribution.)

Another way of finding the distribution of the output of the program is to simply run it a million times, and compute the distribution of what it returns. A compromise between this random sampling, and computing the full distribution, would be to switch between them at each coin flip. For instance, when evaluating a `flip` statement, one might pick true or false according to its probability half the time, and compute the distribution in each case the other half of the time. Doing this would reduce the branching from `flip` statements, but would greatly reduce the accuracy of the output distribution. It's not clear that this would be a useful compromise.

5 References

H. Abelson, G. J. Sussman and J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1984.

A. Bondorf. *Similix 5.0 Manual.* DIKU, University of Copenhagen, Denmark, 1993.

JavaCup. Available on the WWW at
`http://www.cs.princeton.edu/~appel/modern/java/CUP/`

JLex. Available on the WWW at
`http://www.cs.princeton.edu/~appel/modern/java/JLex/`

J. Jørgensen. Generating a Compiler for a Lazy Language by Partial Evaluation. In *ACM SIGACT-SIGPLAN*, 1992.

D. Koller, D. McAllester, and A. Pfeffer. Effective Bayesian Inference for Stochastic Programs. In *AAAI*, 1997.

C. D. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing.* MIT Press, 1999.