Lecture 23: Probabilistic Programming

Steven Holtzen
s.holtzen@northeastern.edu

CS4400/5400 Fall 2024

1 What are Probabilistic Programs?

- Probabilistic programming languages (PPLs) are programming languages whose semantics are probability distributions
- Their defining feature is the ability to introduce and manipulate *first-class probabilistic uncertainty*
- As an example, consider the following syntax:

```
x ← flip 0.5;
y ← flip 0.5;
return x || y
```

- This program evaluates to a probability distribution that maps `#t` to 0.25. and `#f` to 0.75. The process of computing the probability that a probabilistic program returns a particular value is called **probabilistic inference**.
- The `flip 0.5` syntax denotes a random Boolean variable that is true with probability $1/2$ and false with probability $1/2$. In general, `flip p` will be true with probability p and false with probability $1 - p$.
- We use monadic style for the syntax of probabilistic programs, similar to what we saw in Lecture 19. The syntax `x ← flip 0.5` *binds* the outcome of the coin flip to the variable `x`. Note that `x` is a Boolean.
- The syntax `return x || y` *lifts* the pure computation `x || y` to a probability distribution on Booleans.
- Why do we care about probabilistic programs?
  - Reasoning about the probability that something happens is broadly useful (failure probabilities, game-playing, forecasting, insurance, stock markets, fraud detection, etc.). Probabilistic programming languages can automate this process.
  - Reasoning about randomized algorithms and inherent uncertainty for program verification.

2 Implementing a Simple PPL

- Here is our abstract syntax:

```
;;; pure expressions
(struct eif (guard thn els) #:transparent)
(struct eand (e1 e2) #:transparent)
(struct eor (e1 e2) #:transparent)
(struct enot (e) #:transparent)
(struct ebool (v) #:transparent)
(struct eident (id) #:transparent)

;;; probabilistic expressions
(struct ebind (id e1 e2) #:transparent)
(struct ereturn (e) #:transparent)
(struct eflip (p) #:transparent)
```

- Interpreting pure expressions is standard, which is the main reason why we used a monadic style to define our language:

```
;;; interp-pure : env -> expr -> bool
(define (interp-pure env e)
  (match e
    [(ebool b) b]
    [(eif guard thn els)
     (if (interp-pure env guard)
         (interp-pure env thn)
         (interp-pure env els))]
    [(eand e1 e2)
     (and (interp-pure env e1) (interp-pure env e2))]
    [(eor e1 e2)
     (or (interp-pure env e1) (interp-pure env e2))]
    [(enot e)
     (not (interp-pure env e))]
    [(eident id) (hash-ref env id)]))
```

- Implementing the impure probabilistic component of the semantics is a bit more involved, so let's do it in stages.

3 Implementing the Probabilistic Semantics

- One way of understanding the semantics of PPLs is in terms of **possible worlds**. A possible world is an assignment to each `flip` expression: for example, the possible worlds for the example on the first page are:

$$\underbrace{x = \#t, y = \#t}_{\omega_1} \qquad \underbrace{x = \#t, y = \#f}_{\omega_2} \qquad \underbrace{x = \#f, y = \#t}_{\omega_3} \qquad \underbrace{x = \#f, y = \#f}_{\omega_4}$$

- The probability of each possible world is given by the product of the parameters for each `flip`, and is denoted $\Pr(\omega)$. For example, $\Pr(\omega_1) = 0.5 \times 0.5 = 0.25$.
- We are interested in the semantics for the whole program: in particular, we want to know the probability that at least one of `x` and `y` are true. To compute this, we can sum probability of the possible worlds where this is the case. This is $\Pr(\omega_1) + \Pr(\omega_2) + \Pr(\omega 3) = 0.75$.
- Now, let's design an interpreter for our language that follows these semantics.
- The interpreter for probabilistic terms will produce probability distributions, which have the following datatype and interface:

```
(struct distribution (probs) #:transparent)

(define (make-dist trueprob falseprob)
  (distribution (hash #t trueprob #f falseprob)))

(define (true-prob dist)
  (hash-ref (distribution-probs dist) #t))

(define (false-prob dist)
  (hash-ref (distribution-probs dist) #f))
```

- Now we can fill in the skeleton of our interpreter:

```
;;; interp-dist : env -> expr -> prob
;;; returns the probability that an expression evaluates to true
(define (interp-dist-h env e)
  (match e
    [(ereturn e)
     (define res (interp-pure env e))
     (if res
         (make-dist 1 0)
         (make-dist 0 1))]
    [(eflip p) (make-dist p (- 1 p))]
    [(ebind id e1 e2)
     ...]))
```

- The semantics of `ereturn` and `eflip` are relatively straightforward: `ereturn e` assigns a probability of 1 to whichever value `e` evaluates to, and `flip p` assigns a probability `p` to the true outcome and `1-p` to the false outcome.
- Bind is quite tricky, and is where most of the work in our interpreter happens. It helps to look at a couple of small examples of how we want bind to work.
- Consider the following tiny program:

```
x ← flip 0.3;
return x
```

- Intuitively, how should we evaluate this program? Intuitively, it has the following steps:
  - Compute the probability p_t and p_f that `flip 0.3` evaluates to `#t` and `#f` respectively
  - Substitute in the two possible values for `x` into `return x`, and evaluate the semantics
  - Average these two semantics together weighted by p_t and p_f
- We can summarize this using a judgment:

$$\frac{\texttt{flip 0.3} \Downarrow \{\#t \mapsto 0.3, \#f \mapsto 0.7\} \quad \texttt{return \#t} \Downarrow \{\#t \mapsto 1, \#f \mapsto 0\} \qquad \texttt{return \#f} \Downarrow \{\#t \mapsto 0, \#f \mapsto 1\}}{\texttt{x} \leftarrow \texttt{flip 0.3; return x} \Downarrow 0.3 \times \{\#t \mapsto 1, \#f \mapsto 0\} + 0.7 \times \{\#t \mapsto 0, \#f \mapsto 1\}}$$

- Now we can implement these semantics. We first define some auxiliary functions for manipulating probability distributions:

```
;;; scale a distribution by a constant p
(define (scale-dist dist p)
  (distribution
   (hash #t (* (true-prob dist) p)
         #f (* (false-prob dist) p))))

;;; add two distributions point-wise
(define (add-dist dist1 dist2)
  (distribution
   (hash #t (+ (true-prob dist1) (true-prob dist2))
         #f (+ (false-prob dist1) (false-prob dist2)))))
```

- Now, using these auxiliary functions, we can give the semantics for bind:

```
(define (interp-dist-h env e)
  (match e
    ...
    [(ebind id e1 e2)
     ; first, evaluate e1 to a distribution
     (define e1dist (interp-dist-h env e1))
     ; next, evalate e2 for id = #t and id = #f
     (define e2true (interp-dist-h (hash-set env id #t) e2))
     (define e2false (interp-dist-h (hash-set env id #f) e2))
     ; now, construct the new distribution
     (add-dist
      (scale-dist e2true (true-prob e1dist))
      (scale-dist e2false (false-prob e1dist)))]))
```

4 Programming in a Probabilistic Programming Language

- The code we provided has a small parser for our tiny probabilistic programming language, we we can use for writing some simple example programs
- We can test our implementation:

```
> (interp-dist
   (parse-prob '(x <- (flip 0.4)
                 (y <- (flip 0.6)
                 (return (and x y))))))
(distribution '#hash((#f . 0.76) (#t . 0.24)))
```

- Let's try to program something a bit more interesting. Consider the following scenario that models relationships between symptoms and diseases:
  - 2% of people have a cold.
  - 1% of people have the flu.
  - If you have the flu, then there is a 10% chance you have a fever; if you have a cold and no flu, then there is a 2% chance you have a fever; otherwise, there is a 0.1% chance you have a fever.

  Then, what is the chance that an average person has a fever?

- We can model this scenario using a probabilistic program:

```
> (define disease-model
    (parse-prob '(flu <- (flip 0.01)
              (cold <- (flip 0.02)
              (feverIfFlu <- (flip 0.1)
              (feverIfCold <- (flip 0.02)
              (feverHealthy <- (flip 0.001)
              (return (if flu feverIfFlu
                         (if cold feverIfCold
                             feverHealthy))))))))))
> (interp-dist disease-model)
(distribution '#hash((#f . 0.9976337999999999) (#t . 0.0023662)))
```

5 Observation and Bayesian Conditioning

- It is often useful when reasoning about probabilities to be able to update your beliefs in light of new information: to ask *what is the probability of this given that*?
- For example, we might want to be able to perform **medical diagnosis**: i.e., to compute the probability that a patient has a particular disease *given that* the patient has some collection of observed symptoms.
- We can support this style of reasoning in our probabilistic programs with the addition of a probabilistic term `observe e1 e2`, which denotes observing that the outcome `e1` holds and then executing `e2`.
- As a simple example we can consider a simple extension of the coin flipping scenario where we observe that at least one coin is true, and ask for the probability that one of the coins is true:

  ```
  x ← flip 0.5;
  y ← flip 0.5;
  observe x || y;
  return x
  ```

  Intuitively, the probability that `x` is true should *increase*, since the observation gives us additional information about the state of `x` and `y` (i.e., that at least one of them must be true)

- In terms of possible worlds, what is happening with `observe` (recall the possible worlds from earlier)? Intuitively, it does two things:
  - It *eliminates* the worlds that violate the observation (i.e., it eliminates ω_4 by setting its probability to 0)
  - It *renormalizes* the remaining worlds so that their total probability mass is still 1

  The end result of this process is that $\Pr(\omega_1) = \Pr(\omega_2) = \Pr(\omega_3) = 0.25/0.75 = 1/3$ and $\Pr(\omega_4) = 0$. Then, this program should evaluate to a probability distribution that assigns `#t` the probability 2/3. This matches our intuition that the probability that `x` is true should increase upon observation.

6 Implementing Conditioning

- We will handle the two stages of observation separately. First, we will update our `interp-dist-h` function to output *unnormalized probability distributions* that do not necessarily sum to 1:

```
;;; interp-dist : env -> expr -> prob
;;; returns the probability that an expression evaluates to true
(define (interp-dist-h env e)
  (match e
    [(ereturn e)
     (define res (interp-pure env e))
     (if res
         (make-dist 1 0)
         (make-dist 0 1))]
    [(eflip p) (make-dist p (- 1 p))]
    [(ebind id e1 e2)
     ; first, evaluate e1 to a distribution
     (define e1dist (interp-dist-h env e1))
     ; next, evalate e2 for id = #t and id = #f
     (define e2true (interp-dist-h (hash-set env id #t) e2))
     (define e2false (interp-dist-h (hash-set env id #f) e2))
     ; now, construct the new distribution
     (add-dist
      (scale-dist e2true (true-prob e1dist))
      (scale-dist e2false (false-prob e1dist)))]
    [(eobserve e1 e2)
     (if (interp-pure env e1)
         (interp-dist-h env e2)
         (make-dist 0 0))]))
```

- Notice how the semantics of `eobserve` is relatively simple: if the guard of the observation is false, then it outputs the 0 distribution (assigns zero to both the `#t` and `#f` outcome).

- Then, to compute the semantics of the program, we must renormalize:

```
(define (normalize dist)
  (define c (+ (true-prob dist) (false-prob dist)))
  (make-dist (/ (true-prob dist) c)
             (/ (false-prob dist) c)))

(define (interp-dist e)
  (normalize (interp-dist-h (hash) e)))
```

- Now we can again test our program:

```
> (interp-dist
  (parse-prob '(x <- (flip 1/2)
                (y <- (flip 1/2)
                   (observe (or x y)
                            (return x))))))
(distribution '#hash((#f . 1/3) (#t . 2/3)))
```

7 Programming with Observations: Medical Diagnosis

- Returning to our medical diagnosis example, now we can ask: *what is the probability that a patient has a flu given that they have a fever*?

```
> (define disease-model
    (parse-prob '(flu <- (flip 0.01)
             (cold <- (flip 0.02)
             (feverIfFlu <- (flip 0.1)
             (feverIfCold <- (flip 0.02)
             (feverHealthy <- (flip 0.001)
             (fever <- (return (if flu feverIfFlu
                         (if cold feverIfCold
                             feverHealthy)))
             (observe fever
             (return flu)))))))))
> (interp-dist disease-model)
  (distribution '#hash((#f . 0.5773814554982672) (#t . 0.4226185445017327)))
```

8 Hardness of Inference

- Clearly, our approach to inference will not scale very well. For example, the following program will time out if you try to perform inference on it:

```
(define hard-model
    (parse-prob '(x1 <- (flip 0.5)
                  (x2 <- (flip 0.5)
                  (x2 <- (flip 0.5)
                  (x2 <- (flip 0.5)
                  (x2 <- (flip 0.5)
                  (x2 <- (flip 0.5)
                  (x2 <- (flip 0.5)
                  (x2 <- (flip 0.5)
                  (x2 <- (flip 0.5)
                  (x2 <- (flip 0.5)
                  (x2 <- (flip 0.5)
                  (x2 <- (flip 0.5)
                  (x2 <- (flip 0.5)
                  (x2 <- (flip 0.5)
                  (x2 <- (flip 0.5)
                  (x2 <- (flip 0.5)
                  (x2 <- (flip 0.5)
                  (x2 <- (flip 0.5)
                  (x2 <- (flip 0.5)
                  (x2 <- (flip 0.5)
                  (x2 <- (flip 0.5)
                  (x2 <- (flip 0.5)
                  (x2 <- (flip 0.5)
                  (x2 <- (flip 0.5)
                  (x2 <- (flip 0.5)
                  (x2 <- (flip 0.5)
                  (x2 <- (flip 0.5)
                  (return x2))))))))))))))))))))))))))))))
```

- A key research direction in probabilistic programming languages involves designing scalable inference

9 Examples of Probabilistic Programming Languages in the Wild

- Probabilistic programming languages are becoming increasingly widely deployed and are beginning to have some impact
- There are a number of probabilistic programming languages that have been developed and deployed in industry for various applications
- Stan: `https://mc-stan.org/`
- Pyro (originally developed by Uber): `https://pyro.ai/`
- PyMC3: `https://www.pymc.io/projects/docs/en/stable/learn.html`
- Edward/Tensorflow Probability (Google): `https://www.tensorflow.org/probability`

10 Roulette

- Here is a preview of some work that our group is currently doing in the space of designing scalable probabilistic programming languages
- Our group has made a new probabilistic programming language based on Racket called "Roulette" that scales quite well on some large programs, while being expressive enough to support almost all of Racket.[1]
- Roulette's syntax looks just like Racket, but with the addition of `flip` and `observe`:

```
#lang roulette/example/discrete
> (flip 0.5)
#hash((#t . 0.5) (#f . 0.5))
; roulette can support large computations! this completes instantly:
> (and (flip 0.5)
       (flip 0.5)
       (flip 0.5)
       (flip 0.5)
       (flip 0.5)
       (flip 0.5)
       (flip 0.5)
       (flip 0.5)
       (flip 0.5)
       (flip 0.5)
       (flip 0.5)
       (flip 0.5)
       (flip 0.5)
       (flip 0.5)
       (flip 0.5)
       (flip 0.5)
       (flip 0.5)
       (flip 0.5)
       (flip 0.5)
       (flip 0.5)
       (flip 0.5)
       (flip 0.5)
       (flip 0.5)
       (flip 0.5)
       (flip 0.5)
       (flip 0.5)
       (flip 0.5)
       (flip 0.5)
       (flip 0.5))
#hash((#t . 1.862645149230957e-9) (#f . 0.9999999981373549))
> (define coin1 (flip 0.5))
> (define coin2 (flip 0.5))
> (observe (or coin1 coin2))
> coin1
#hash((#t . 0.6666666666666666) (#f . 0.3333333333333333))
; roulette supports functions
> (define (add-sometimes x y)
> (if (flip 0.5)
      (+ x y)
      x))
```

[1]This work isn't published yet, and was just submitted to a conference 3 weeks ago! The project is led by Cameron Moy, with help from Jack Czenszak, John Li, and Brianna Marshall. If you would like to see a preprint, email me.

```
> (add-sometimes 10 20)
#hash((10 . 0.5) (30 . 0.5))
```

11 Probability with Effects

```
#lang racket

(require effect-racket)

(effect flip (prob))

(define (process r prob b)
  (match r
    [#t prob]
    [#f 0]
    [r (if b (* r prob) (* (- 1 prob) r))]))

(define (prob-service)
  (handler
   [(flip prob)
    (define r1 (continue #t))
    (define r2 (continue #f))
    (define p1 (process r1 prob #t))
    (define p2 (process r2 prob #f))
    (+ p1 p2)]))

(with ((prob-service))
      (define a (flip 1/3))
      (define b (flip 1/2))
      (and a b))
```