Probabilistic Programming

Figure credit: Frank Wood
Why Probabilistic Programming?
Simplify Machine Learning…

Start

Identify And Formalize Problem, Gather Data

Existing Model Sufficient?

Search For Useable Implementation

Performs Well Statistically?

Scale

Perform Well Computationally?

End

Design Model + Read Papers, Do Math

Exists And Can Use?

Test

Derive Updates And Code Inference Algorithm

Feasible?

Deploy

Simple Model?

Implement Using High Level Modeling Tool

Tool Supports Required Features?

Choose Approximate Inference Algorithm

LEGEND Color indicates the skills that are required to traverse the edge.

- Non-specialist
- PhD-level machine learning or statistics
- PhD-level machine learning or computer science

Slide credits: Frank Wood
**To This**

1. **Start**
   - Identify And Formalize Problem, Gather Data

2. **Design Model = Write Probabilistic Program**
   - Design Model = Write Probabilistic Program

3. **Search For Useable Implementation**
   - Search For Useable Implementation

4. **Existing Model Sufficient?**
   - Existing Model Sufficient?

5. **Performs Well?**
   - Perform Well?
     - N
     - Y

6. **Scale**
   - Scale

7. **Derive Updates And Code Inference Algorithm**
   - Derive Updates And Code Inference Algorithm

8. **Perform Well Computationally?**
   - Perform Well Computationally?

9. **Deploy**
   - Deploy

10. **End**
    - End

**Legend**
- Color indicates the skills that are required to traverse the edge.

- Non-specialist

---

*Slide credits: Frank Wood*
Latent Dirichlet Allocation is formally written as:

\[
\begin{align*}
\text{independently from Mult(} & \text{ Each of generative model for the } \\
& \text{ for convenience only and the integer mapping will contain no semantic information. The translated to the set of integers denoted as } \\
& \text{Here, we will denote by } \{ & \text{a mixture of topics. Each topic is defined as a distribution over the words in the vocabulary. } \\
& \mathrm{Dir} & \text{ are drawn independently from Dir} \\
& \mathrm{Mult(} & \mathrm{Mult(} \\
& \{ & \text{denote the number of words in document } \\
& & \text{topics } d \} \{ & \text{words in document } \\
& & \text{d} \} & \{ & \text{and the scalar positive parameters } \\
& & \mathrm{Mult(} & \text{hyperparameters } \\
& & \text{Under the uniform deletion model, the number } \\
& & \text{to indicate the } \{ & \text{The model is parameterized by the vector valued parameters } \\
& & \{ & \text{topics } d \} & \{ & \text{documents can be thought of as sequentially drawing a topic } \\
& & \{ & \text{Discrete } \\
& & \} & \{ & \text{through the use of a static dictionary. This is } \\
& & \} & \{ & \text{We will assume that the words have been } \\
& & \} & \{ & \text{(Pitt and Walker, 2005).} \\
& & \} & \{ & \text{Combining the stationary Pitman-Yor and cluster locations models, we can summarize the full } \\
& & \} & \{ & \text{It is possible to define a slightly modified version of our model that is consistent under marginal- } \\
& & \} & \{ & \text{Inference Engine(s)} \\
& & \} & \{ & \text{ARON ET AL} \\
& & \} & \{ & \text{Slide credits: Frank Wood} \\
\end{align*}
\]
What is Probabilistic Programming?
Operative Definition

“Probabilistic programs are usual functional or imperative programs with two added constructs:

(1) the ability to draw values at random from distributions, and

(2) the ability to condition values of variables in a program via observations.”

Gordon et al, 2014
Probabilistic Programs: Defining Sampling Processes

```javascript
// create a gaussian distribution:
var g = Gaussian({mu: 0, sigma: 1})

// sample from it:
print( sample(g) )

// can also use the sampling helper (note lower-case name):
print( gaussian(0,1) )

// and build more complex processes!
var foo = function(){return gaussian(0,1)*gaussian(0,1)}
foo()
```
Probabilistic Programs:
Defining Sampling Processes

```javascript
// create a gaussian distribution:
var g = Gaussian({mu: 0, sigma: 1})

// sample from it:
print( sample(g) )

// can also use the sampling helper (note lower-case name):
print( gaussian(0,1) )

// and build more complex processes!
var foo = function(){return gaussian(0,1)*gaussian(0,1)}
foo()
```
Probabilistic Programs: Defining Sampling Processes

// create a gaussian distribution:
var g = Gaussian({mu: 0, sigma: 1})

// sample from it:
print( sample(g) )

// can also use the sampling helper (note lower-case name):
print( gaussian(0,1) )

// and build more complex processes!
var foo = function(){return gaussian(0,1)*gaussian(0,1)}
foo()
Probabilistic Programs:
Defining Sampling Processes

```javascript
// create a gaussian distribution:
var g = Gaussian({mu: 0, sigma: 1})

// sample from it:
print( sample(g) )

// can also use the sampling helper (note lower-case name):
print( gaussian(0,1) )

// and build more complex processes!
var foo = function(){return gaussian(0,1)*gaussian(0,1)}
foo()
```

(Distribution objects)
(Distributions support sample)
(Easy to build complex distributions)
Probabilistic Programs: Defining Sampling Processes

//create a gaussian distribution:
var g = Gaussian({mu: 0, sigma: 1})

//sample from it:
print(sample(g))

//can also use the sampling helper (note lower-case name):
print(gaussian(0,1))

//and build more complex processes!
var foo = function(){return gaussian(0,1)*gaussian(0,1)}
foo()

run

0.4844819841420675
1.4341692553095442
0.08057731257784836
Probabilistic Programs:
Defining Sampling Processes

The generative model is now defined by a sampling process:
A sampling process implicitly defines a distribution over output values...
Another PPL construct makes this distribution explicit: Infer
Probabilistic Programs:
`Infer` Construct: Convert Implicit Distribution to Explicit Object

```javascript
// a complex function, that specifies a complex sampling process:
var foo = function(){gaussian(0,1)*gaussian(0,1)}

// make the marginal distributions on return values explicit:
var d = Infer({method: 'forward', samples: 10000}, foo)

// now we can use d as we would any other distribution:
print( sample(d) )
viz(d)
```
Probabilistic Programs:
`Infer` Construct: Convert Implicit Distribution to Explicit Object

```javascript
// a complex function, that specifies a complex sampling process:
var foo = function(){gaussian(0,1)*gaussian(0,1)}

// make the marginal distributions on return values explicit:
var d = Infer({method: 'forward', samples: 10000}, foo)

// now we can use d as we would any other distribution:
print( sample(d) )
viz(d)
```

(Implicitly Defined Distribution)

(Infer by Forward Sampling)
Probabilistic Programs:
`Infer` Construct: Convert Implicit Distribution to Explicit Object

---

// a complex function, that specifies a complex sampling process:
var foo = function(){gaussian(0,1)*gaussian(0,1)}

// make the marginal distributions on return values explicit:
var d = Infer({method: 'forward', samples: 10000}, foo)

// now we can use d as we would any other distribution:
print( sample(d) )
viz(d)
Probabilistic Programs:
`Infer` Construct: Convert Implicit Distribution to Explicit Object

// a complex function, that specifies a complex sampling process:
var foo = function() {gaussian(0,1) * gaussian(0,1)}

// make the marginal distributions on return values explicit:
var d = Infer({method: 'forward', samples: 10000}, foo)

// now we can use d as we would any other distribution:
print(sample(d))
viz(d)
Need one more language feature: “mem”
`Random but persistent`: random on first call, cached for subsequent calls

Why needed:

```javascript
var eyeColor = function (person) {
    return uniformDraw(['blue', 'green', 'brown']);
};
print([eyeColor('bob'), eyeColor('alice'), eyeColor('bob')]);
print([eyeColor('bob'), eyeColor('alice'), eyeColor('bob')]);
```
Need one more language feature: “mem”
`Random but persistent`: random on first call,
cached for subsequent calls
Why needed:

```javascript
var eyeColor = function (person) {
    return uniformDraw(['blue', 'green', 'brown']);
};
print([eyeColor('bob'), eyeColor('alice'), eyeColor('bob')]);
print([eyeColor('bob'), eyeColor('alice'), eyeColor('bob')]);
```

Call once
Call twice
Need one more language feature: “mem”

`Random but persistent`: random on first call, cached for subsequent calls

Why needed:

```javascript
var eyeColor = function (person) {
    return uniformDraw(['blue', 'green', 'brown']);
};
print([eyeColor('bob'), eyeColor('alice'), eyeColor('bob')]);
print([eyeColor('bob'), eyeColor('alice'), eyeColor('bob')]);
```

Call once
Call twice

Bob’s eye color shouldn’t change...
Need one more language feature: `mem`

`Random but persistent`: random on first call, cached for subsequent calls

Why needed:

```javascript
var eyeColor = mem(function (person) {
    return uniformDraw(['blue', 'green', 'brown']);
});
print([eyeColor('bob'), eyeColor('alice'), eyeColor('bob')]);
print([eyeColor('bob'), eyeColor('alice'), eyeColor('bob')]);
```

Call once

Call twice

Fixed: value is memoized after first run
Aside:
Dirichlet Process as Probabilistic Program
Recall: Dirichlet as Stick-Breaking Process

\[
\{\beta_k\}_{k=1}^{\infty}, \quad \beta_k' \sim \text{Beta}(1, \alpha)
\]

\[
Pr\{k\} = \beta_k = \prod_{i=1}^{k-1} (1 - \beta_i') \cdot \beta_k'
\]

As generative model:

- Walk down the natural numbers
- Flip a biased coin at each number: \(\text{Ber}(\beta_i')\)
- If FALSE, continue to next number. If TRUE, return the number
As probabilistic program

```
var pickStick = function(sticks, J) {
  return flip(sticks(J)) ? J : pickStick(sticks, J+1);
};

var makeSticks = function(alpha) {
  var sticks = mem(function(index) {return beta(1, alpha)});
  return function() {
    return pickStick(sticks,1)
  }
};

var mySticks = makeSticks(1);

viz(repeat(1000, mySticks))
```
As probabilistic program

```javascript
var pickStick = function(sticks, J) {
    return flip(sticks(J)) ? J : pickStick(sticks, J+1);
};

var makeSticks = function(alpha) {
    var sticks = mem(function(index) {return beta(1, alpha)});
    return function() {
        return pickStick(sticks,1)
    }
}
var mySticks = makeSticks(1);

viz(repeat(1000, mySticks))
```
Universal Inference for Probabilistic Programming Languages

(define (ibp-stick-breaking-process concentration base-measure)
  (let ((sticks (ren (lambda j (random-beta 1.0 concentration))))
       (atoms (ren (lambda j (base-measure))))
       (lambda ()
         (let loop (((j 1) (dualstick (sticks 1))))
           (append (if (flip dualstick) ;; with prob. dualstick
                       (atoms j) ;; add feature j
                       '()) ;; otherwise, next stick
             (loop (+ j 1) (∗ dualstick (sticks (+ j 1))))) ))))
So far…

- Build complicated probabilistic models with PPLs
- Using `sample` statements: Specify prior generative proc.
- Using `factor` statements: Specify data likelihood
- A prob. program represents posterior over possible execution “traces”

How to develop generic inference algorithms?
What is a “Trace”? 

- Sequence of $M$ sample statements
  \[
  \{f_j, \theta_j\}_{j=1}^{M}
  \]

- Sequence of $M$ sampled values
  \[
  \{x_j\}_{j=1}^{M}
  \]

- Sequence of $N$ factor statements
  \[
  \{g_i, \phi_i, Y_i\}_{i=1}^{N}
  \]
Inference over traces

- Trace probability:

\[ \gamma(x) \triangleq p(x, y) = \prod_{i=1}^{N} g_{i}(y_{i}|\phi_{i}) \prod_{j=1}^{M} f_{j}(x_{j}|\theta_{j}) \]

- Posterior over traces:

\[ \pi(x) \triangleq p(x|y) = \frac{\gamma(x)}{Z} \quad \text{with} \quad Z = p(y) = \int \gamma(x) dx \]

- What we care about:

\[ \mathbb{E}_{\pi(x)} [f(x)] \]
Sampling based inference over “traces”

Inference over trace space: exploration–exploitation task

- **Explore** possible execution paths
  - As a side-effect, compute “goodness” of a trace

- **Exploit** good (more probable) traces
  - Return projection of the posterior over traces
Inference over execution traces

```javascript
var my_model = function(){
    var x_1 = sample(Categorical({vs: [0, 1, 2], ps: [.33, .33, .33]}));
    if (x_1 == 1){
        var x_2 = poisson(x_1 + 7);
       // factor(Gaussian({mu:x_2,sigma:0.0001}).score(1)) // y ~ N(x_1,0.0001) ; obs y =
    }
    return x_1;
}
var dist = Infer({method: 'MCMC', samples:1000000, burn: 10000}, my_model)
viz(dist)
```

\[
x_2 | x_1 = 0 \sim \text{Poisson}(7)
\]

\[
x_2 | x_1 = 1 \sim \text{Poisson}(9)
\]

\[
x_1 \sim \text{Categorical}(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})
\]

\[
x_1 = 0
\]

\[
x_1 = 1
\]

\[
x_1 = 2
\]

\[
x_2 | x_1 = 2 \sim \text{Poisson}(9)
\]
Inference over execution traces

```javascript
var my_model = function()
{
  var x_1 = sample(Categorical({vs: [0, 1, 2], ps: [.33, .33, .33]}));
  if (x_1 !== 1){
    var x_2 = poisson(x_1 + 7);
    factor(Gaussian({mu:x_2,sigma:0.0001}).score(1)) // y ~ N(x_2,0.0001) ; obs y = 1
  }
  return x_1;
}
var dist = Infer({method: 'MCMC', samples:1000000, burn: 10000}, my_model)
viz(dist)
```
Importance sampling

• Run $K$ independent copies of the program simulating from the prior

$$q(x^k) = \prod_{j=1}^{M_k} f_j(x_j^k | \theta_j^k)$$

• Calculate importance weights as follows:

$$w(x^k) = \frac{\gamma(x^k)}{q(x^k)} = \prod_{i=1}^{N_k} g_i^k(y_i^k | \phi_i^k)$$

$$W^k = \frac{w(x^k)}{\sum_{\ell=1}^{K} w(x^\ell)}$$

• Approximate expectation by Monte Carlo integration

$$\mathbb{E}_{\pi(x)} [f(x)] \approx \sum_{k=1}^{K} W^k f(x^k)$$
Importance sampling

\[ f(x^K), w^K \]

\[ f(x^1), w^1 \]

\[ f(x^2), w^2 \]

\[ \vdots \]

\[ \vdots \]
Single-Site *Metropolis–Hastings*

Want samples from  
$$\pi(x) \triangleq p(x|y) = \frac{\gamma(x)}{Z}$$

- Pick a proposal distribution  
  $$q(x'|x)$$  
  that generates a new trace given current trace

- Use Metropolis–Hastings acceptance

$$\alpha = \min \left( 1, \frac{\pi(x')q(x|x')}{\pi(x)q(x'|x)} \right)$$
Single-Site Metropolis–Hastings

\[ f(x^1) \]

\[ f(x^2)f(x^1) \quad \alpha_1 \]

\[ f(x^2) \quad \alpha_2 \]

\[ \vdots \]

\[ f(x^K) \quad \alpha_K \]
Single-Site Metropolis–Hastings

\[ q(x' | x^s) = \frac{1}{M^s} \kappa(x'_l | x^s_l) \prod_{j=l+1}^{M'} f'_j(x'_j | \theta'_j) \]

- \(M^s\) = Number of random elements in old trace
- \(\kappa(x'_l | x^s_l)\) = Proposal distribution for the \(l\)th random element

Can set \(\kappa(x'_m | x_m) = f_m(x'_m | \theta_m), \theta_m = \theta'_m\)
What did we cover?

WebPPL

CHURCH

VENTURE

(\lambda (P \lambda))

ANGLICAN

MONAD-BAYES
What did we miss?
That’s all folks!