Probabilistic Programming

Daniel M. Roy

Department of Statistical Sciences
Department of Computer Science
University of Toronto

Workshop on Uncertainty in Computation
2016 Program on Logical Structures in Computation
Simons Institute for the Theory of Computing
1. Simple story:
   Probabilistic programming automates Bayesian inference

2. Real story:
   It’s complicated
Probabilistic programming

1. Represent probability distributions by *formulas programs that generate samples*.

2. Build *generic algorithms for probabilistic conditioning* using probabilistic programs as representations.

Bayesian statistics

1. Express statistical assumptions via *probability distributions*.

   \[
   \Pr(\text{parameters, data}) = \Pr(\text{parameters}) \Pr(\text{data | parameters})
   \]

   \[
   \text{joint} \quad \text{prior} \quad \text{model/likelihood}
   \]

2. Statistical inference from data \(\rightarrow\) parameters via *conditioning*.

   \[
   \Pr(\text{parameters, data}), \quad x \xrightarrow{\text{conditioning}} \Pr(\text{parameters | data} = x)
   \]

   \[
   \text{posterior}
   \]
Example: simple probabilistic Python program

```python
1  def binomial(n, p):
2      return sum([bernoulli(p) for i in range(n)])
```

- returns a random integer in \{0, \ldots, n\}.
- defines a family of distributions on \{0, \ldots, n\}, in particular, the Binomial family.
- represents a statistical model of
  - the number of successes among
  - \(n\) independent and identical experiments
Example: simple probabilistic Python program

```python
def binomial(n, p):
    return sum([bernoulli(p) for i in range(n)])

def randomized_trial():
    p_control = uniform(0, 1)  # prior
    p_treatment = uniform(0, 1)  # prior
    return (binomial(100, p_control),
            binomial(10, p_treatment))
```

represents a **Bayesian model** of a randomized trial.

\[
\text{simulation} \quad \rightarrow \quad (71, 9)
\]

\[
\text{inference} \quad \leftarrow \quad (71, 9)
\]
The stochastic inference problem

**INPUT:** guesser and checker probabilistic programs.

**OUTPUT:** a sample from the same distribution as the program

```python
accept = False
while (not accept):
    guess = guesser()
    accept = checker(guess)
return guess
```

This computation captures **Bayesian statistical inference**.

“prior” distribution $\leftrightarrow$ distribution of `guesser()`

“likelihood($g$)” $\leftrightarrow$ $\Pr(\text{checker}(g) \text{ is True})$

“posterior” distribution $\leftrightarrow$ distribution of return value
Example: inferring bias of a coin

```python
accept = False
while (not accept):
    guess = guesser()
    accept = checker(guess)
return guess
```

Given \(x_1, \ldots, x_n \in \{0, 1\}\),
report probability of \(x_{n+1} = 1\)? E.g., 0, 0, 1, 0, 0

```python
def guesser():
    p = uniform()
    return p
def checker(p):
    return [0,0,1,0,0] == bernoulli(p,5)
```
Example: inferring bias of a coin

```python
accept = False
while (not accept):
    guess = guesser()
    accept = checker(guess)
return guess
```

Given $x_1, \ldots, x_n \in \{0, 1\}$,
report probability of $x_{n+1} = 1$? E.g., 0, 0, 1, 0, 0

```python
def guesser():
    p = uniform()
return p
def checker(p):
    return [0,0,1,0,0] == bernoulli(p,5)
```
Example: inferring bias of a coin

```python
def guesser():
    p = uniform()
    return p

def checker(p):
    return [0,0,1,0,0] == bernoulli(p,5)
```

```python
accept = False
while (not accept):
    guess = guesser()
    accept = checker(guess)
return guess
```

Given $x_1, \ldots, x_n \in \{0, 1\}$,
report probability of $x_{n+1} = 1$? E.g., 0, 0, 1, 0, 0
Example: inferring objects from an image

```python
accept = False
while (not accept):
    guess = guesser()
    accept = checker(guess)
return guess
```

How many objects in this image?
Example: inferring objects from an image

```python
accept = False
while (not accept):
    guess = guesser()
    accept = checker(guess)
return guess
```

How many objects in this image?
Example: inferring objects from an image

```python
accept = False
while (not accept):
    guess = guesser()
    accept = checker(guess)
return guess
```

How many objects in this image?
Example: inferring objects from an image

```python
accept = False
while (not accept):
    guess = guesser()
    accept = checker(guess)
return guess
```

How many objects in this image?

```python
def guesser():
    k = geometric()
    blocks = [ randomblock() for _ in range(k) ]
    colors = [ randomcolor() for _ in range(k) ]
    return (k,blocks,colors)
def checker(k,blocks,colors):
    return rasterize(blocks,colors) ==
```
Example: inferring objects from an image

```python
accept = False
while (not accept):
    guess = guesser()
    accept = checker(guess)
return guess
```

How many objects in this image?

```python
def guesser():
    k = geometric()
    blocks = [randomblock() for _ in range(k)]
    colors = [randomcolor() for _ in range(k)]
    return (k, blocks, colors)

def checker(k, blocks, colors):
    return rasterize(blocks, colors) ==
```

```python
7/50
```
Example: inferring objects from an image

```python
accept = False
while (not accept):
    guess = guesser()
    accept = checker(guess)
return guess
```

How many objects in this image?

```python
def guesser():
    k = geometric()
    blocks = [ randomblock() for _ in range(k) ]
    colors = [ randomcolor() for _ in range(k) ]
    return (k,blocks,colors)

def checker(k,blocks,colors):
    return rasterize(blocks,colors) ==
```
Example: inferring objects from an image

```python
accept = False
while (not accept):
    guess = guesser()
    accept = checker(guess)
return guess
```

How many objects in this image?

```python
def guesser():
    k = geometric()
    blocks = [ randomblock() for _ in range(k) ]
    colors = [ randomcolor() for _ in range(k) ]
    return (k,blocks,colors)

def checker(k,blocks,colors):
    return rasterize(blocks,colors) ==
```
Example: inferring objects from an image

accept = False
while (not accept):
    guess = guesser()
    accept = checker(guess)
return guess

How many objects in this image?

def guesser():
    k = geometric()
    blocks = [ randomblock() for _ in range(k) ]
    colors = [ randomcolor() for _ in range(k) ]
    return (k,blocks,colors)

def checker(k,blocks,colors):
    return rasterize(blocks,colors) ==
Fantasy example: extracting 3D structure from images

```python
accept = False
while (not accept):
    guess = guesser()
    accept = checker(guess)
return guess
```
Fantasy example: extracting 3D structure from images

```python
accept = False
while (not accept):
    guess = guesser()
    accept = checker(guess)
return guess
```

inference
Example from Mansinghka’s group at MIT
def geometric(p):
    if bernoulli(p) == 1: return 0
    else: return 1 + geometric(p)

represents the Geometric distribution with mean $1/p - 1$.

Note: no bound on running time! Only halts with probability 1!

A sampler that halts with probability one is called a.e. computable.

A sampler that always halts is called computable.

**Theorem.** The set of distributions with a.e. computable samplers is a strict superset of those with computable samplers.
def condition(guesser, checker):
    # guesser: Unit -> S
    # predicate : S -> Boolean
    accept = False
    while (not accept)
        guess = guesser()
        accept = checker(guess)
    return guess

represents the higher order operation of **conditioning**. When **checker** is deterministic, then

\[(P, 1_A) \mapsto P(\cdot | A) = \frac{P(\cdot \cap A)}{P(A)}\].

**Halts with probability 1 provided** \(P(A) > 0\).
condition as an algorithm

**Key point:** condition is not a serious proposal for an *algorithm*, but it denotes the operation we care about in Bayesian analysis.

How efficient is condition? Let \texttt{model()} represent a distribution \( P \) and \texttt{pred} represent an indicator function \( 1_A \).

**Proposition.** In expectation, \texttt{condition(model,pred)} takes \( \frac{1}{P(A)} \) times as long to run as \texttt{pred(model())}.

**Corollary.** If \texttt{pred(model())} is efficient and \( P(A) \) not too small, then \texttt{condition(model,pred)} is efficient.
An efficient “version” of condition

State-of-the-art CHURCH engines work by MCMC, performing a random walk over the possible executions of \texttt{model()}. These engines are complex, but we can ignore polynomial factors to get a much simpler algorithm:

```python
# assume: r_model(random()) \equiv model()
# : perturb(random()) \equiv random()
# (and ergodic w.r.t. random() on every meas. set)
def mcmc_condition(r_model, predicate, n):
    entropy = random()
    for _ in range(n):
        new_entropy = perturb(entropy)
        if predicate(r_model(new_entropy)) == True:
            entropy = new_entropy
    return r_model(new_entropy)
```

approximates condition to arbitrary accuracy as \( n \to \infty \).
Perturb and traces

The key to this working reasonably well is perturb and to understand perturb we’ve got to understand “traces”.

A traces is a tree data structure the captures the random choices encountered and the path taken by the interpreter while evaluating the probabilistic program.

Traces have two key parts.

1. Random primitives.
2. Applications/control-flow.

The trace is determined by the values of the random primitives. Changes to these primitives can modify the control flow.

Goal of perturb is to efficiently take small steps in the space of traces.
Perturb and traces

The key to this working reasonably well is \texttt{perturb} and to understand perturb we’ve got to understand “traces”.

A traces is a tree data structure the captures the random choices encountered and the path taken by the interpreter while evaluating the probabilistic program.

Traces have two key parts.

1. Random primitives.
2. Applications/control-flow.

The trace is determined by the values of the random primitives. Changes to these primitives can modify the control flow.

Goal of \texttt{perturb} is to efficiently take small steps in the space of traces.
“Universal” MCMC inference for probabilistic programs
def geometric(p):
    if bernoulli(p) == 1:
        return 1
    else:
        return 1 + geometric(p)

def aliased_geometric(p):
    g = geometric(p)
    return 1 if g < 3 else 0
MIT-Church [GMR+08]
MIT-Church [GMR+08]
\[
\left(1 - \frac{1}{2} \tilde{p} \quad \frac{1}{2} \tilde{p}\right)^k \rightarrow \left(\frac{p}{p + \tilde{p}}, \frac{\tilde{p}}{p + \tilde{p}}\right) \quad \text{as } k \rightarrow \infty
\]
The real story:
Probabilistic programming automates is game changing but messy
What is probabilistic programming?

Simple story from Gordon et al. (2014). Probabilistic Programming. ICSE.

The goal of probabilistic programming is to enable probabilistic modeling and machine learning to be accessible to the working programmer, who has sufficient domain expertise, but perhaps not enough expertise in probability theory or machine learning. We wish to hide the details of inference inside the compiler and run-time, and enable the programmer to express models using her domain expertise and dramatically increase the number of programmers who can benefit from probabilistic modeling.

This is essentially the perspective the subfield took in 2008 at the 1st NIPS Workshop on Probabilistic Programming. It has turned out to be possible for simple models with fixed structure, but constant work is required on the part of system designers to keep cracks from appearing in this facade as users then push beyond these simple models.

Many systems now diverge from presenting this facade. E.g., in Venture, probabilistic programs are interactions with an approximate inference engine. Users control aspects of inference, which is emphasized to be approximate, but convergent.
Q: Can we automate Bayesian reasoning?

\[ \Pr(X, Y), \ x \longrightarrow \Pr(Y|X = x) \]

A: No, but almost.

The halting distribution (Ackerman, Freer, and R., 2011)

\[ Y \sim \text{Uniform}[0, 1] \]
\[ N := \lfloor -\log_2 Y \rfloor \]
\[ X|Y \sim \text{HaltingTime}(M_N) \]

\[
\frac{\Pr(N = n \mid X = x)}{\Pr(N = \ast \mid X = x)} \cdot 2^{n-\ast}
\]
\[ \in \begin{cases} 
\{2/3, 4/3\} & M_n \text{ halts;} \\
\{1\} & \text{otherwise.}
\end{cases} \]

**Theorem (Ackerman, Freer, and R., 2011).**
The halting problem is computable from \( \Pr(Y \mid X = x) \).
Q: What about efficient inference?

\[ \text{Pr}(X,Y), \ x \xrightarrow{} \text{Pr}(Y|X = x) \]

A: No, of course, but...

\begin{verbatim}
def hash_of_random_string(n):
    str = random_binary_string(n)
    return cryptographic_hash(str)
\end{verbatim}

Bayes nets, Tree Width, ...

Conditional independence is importance!
Conditional independence and traces

Conditional independence looks flat in a trace.

1. bad shape: deep
   no conditional independence

2. good shape: shallow
   lots of conditional independence

**Key idea:** There are multiple ways to implement the same distribution. Choose one that keeps the trace shallow.
Measuring the reliability of MCMC is hard

Standard diagnostics are heuristic and can only hint that something is wrong, rather than guarantee that answers are reliable.

(a) Posterior distribution
(b) Predictive distribution
(c) Clusters
(d) Log-likelihood
(e) Complete Log-likelihood
(f) Autocorrelation (LogLik)

Table 1: Time spend in inference in seconds

<table>
<thead>
<tr>
<th></th>
<th>CRP</th>
<th>SB</th>
<th>PT</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>2910.23</td>
<td>10274.97</td>
<td>18357.70</td>
</tr>
<tr>
<td>metropolis</td>
<td>7089.60</td>
<td>21405.44</td>
<td>40053.26</td>
</tr>
</tbody>
</table>
Bidirectional Monte Carlo [GAG16]

New tools based on BDMC (joint work with Grosse and Ancha, NIPS 2016, see also Cusumano–Mansinghka arXiv:1606.00068) provide first rigorous approach to diagnosing chains on synthetic data.

**REVERSE AIS:** $p_t \rightarrow p_1$ yields *upper* bound $Z^+$ on marginal likelihood

**FORWARD AIS:** $p_1 \rightarrow p_t$ yields *lower* bound $Z^-$ on marginal likelihood

**Theorem [GAR16,CM16].** Difference $Z^+ - Z^-$ bounds MCMC approximation error in expectation.
Aspects of probabilistic programming

- Statistical models are *models*
  - Box: “All models are wrong, some are useful.”
  - Claim: We have a license to approximate
    (Big difference with logic programming)
  - Stronger claim: exact inference not possible in general
  - Stronger still: accuracy of approximate inference usually unknown, but good empirical performance on task suffices

- “Right” semantics for probabilistic program depends on inference
  - Inference is an *extremely* complex transformation:
    probabilistic program $\leftrightarrow$ inference program
  - Distributional semantics: too coarse to capture this interaction
  - Most general languages demand a lot of users.
    - In contrast, Stan.

- Do we know how to write good probabilistic programs?
- Efficient approximations rule
  - Conditioning is hard
  - Whether an approximation is reasonable is often problem specific
Conclusions

- Probabilistic programming for machine learning and statistics is a grand challenge for programming languages
- Underlying computational problem is much harder than execution
- Extreme approximation is licensed
- Current users must understand how to interact with underlying inference