Introduction to Probabilistic Programming Language (with Church as an example)

Presenter:
Enrique Rosales, Xing Zeng
Knowledge

How can we infer knowledge from observations?
Ron’s box

Bob has a box with two buttons and a light.

He presses both buttons, and the light comes on.
Ron’s box

How does the box work?
Ron’s box

Red alone

Green alone
Ron’s box

Red or Green

Red and Green
Ron’s box

None
Different ideas of cognitive science

Thought is useful in an uncertain world
Different ideas of cognitive science

Thought is useful in an uncertain world

Noise
Different ideas of cognitive science

Thought is useful in an uncertain world

Lack of information
Different ideas of cognitive science

Thought is useful in an uncertain world

Noise

and/or

Lack of information
Different ideas of cognitive science

Why did you think it was a photograph?
Different ideas of cognitive science

Thought is productive
“Infinite use of finite means”
Different ideas of cognitive science

Thought is productive
“Infinite use of finite means”

“A big green bear who loves chocolate”
Different ideas of cognitive science

Thought is productive
“Infinite use of finite means”

“A big green bear who loves chocolate”
Different ideas of cognitive science

Thought is productive
“Infinite use of finite means”

Thought can be a composition of several concepts
Different ideas of cognitive science

We have created concepts like:

Mass
Compiler
Friendship...

Thought is productive
“Infinite use of finite means”

Thought can be a composition of several concepts
Different ideas of cognitive science

Thought is useful in an uncertain world

Thought is productive “Infinite use of finite means”

Probabilistic Inference

Compositional representation
Different ideas of cognitive science

Thought is useful in an uncertain world

Thought is productive “Infinite use of finite means”

Probabilistic Inference

Compositional representation

Generative models
Probabilistic language of thought hypothesis

- Mental representations are compositional
  - Their meaning is probabilistic
  - They encode generative knowledge

Hence, they support thinking and learning by probabilistic inference
Ron’s box

![Probability chart]

- Red only: 0.2
- Green only: 0.2
- Red or Green: 0.3
- Red and Green: 0.4
- None: 0
Ron’s box

Beliefs:

Desires:

Actions:

Decision
Generative model

Generative models represent knowledge about the causal structure of the world – simplified, “working models” of a domain. These models may then be used to answer many different questions, by conditional inference.
Generative model

Generative knowledge is often causal knowledge that describes how events or states of the world are related to each other. As an example of how causal knowledge can be encoded, consider a simplified medical scenario:

- Probability of having lung-cancer 0.01
- Probability of having cold 0.2

Which happen if I cough?

We can use programs to describe knowledge, in this case, uncertain knowledge.
Invented by Alonzo Church in 1936 as a way of formalizing the notion of an effectively computable function.

Notation: (+ x y)

\( \lambda \) makes functions, define binds values to symbols:

```scheme
(define double
  (\( x \) (+ x x)))

(double 3) => 6
```
(define repeat
(\(f\)(\(x\))(f (f x))))

((repeat double) 3 ) => 12

(define 2nd-derivative ( repeat derivative ))
- How can we use these ideas to describe probabilities?

- \( \Psi \lambda \)-Calculus: a stochastic variant

- We introduce a random choice operator that reduces to it’s first or second subexpression.
- The usual evaluation rules now result in \textit{sampled} values. This induces \textit{distributions}.

- This calculus, plus primitive operators and data types, gives the probabilistic programming language Church.
Probabilistic Programming

Main Idea: Describe and Inference Generative Models in a systematic and automatic way without rewriting individual algorithm.

Main Tools: use $Ψλ$-Calculus to describe Generative Models first. Then compile the describe models into more “primitive” languages.
Church

A Universal Probabilistic Programming Language.

Named after Alonzo Church.

Based on Scheme, one of the main dialect of Lisp.

Developed in 2008 at MIT BCS/CSAIL.

Roots of a lot of more “modern” PPL, includes: Anglican, Probabilistic-C, WebPPL.
(define a ( flip 0.3 ))
(define b ( flip 0.3 ))
(define c ( flip 0.3 ))
(+ a b c )
Church

\[
\begin{align*}
&(\text{define } a ( \text{ flip } 0.3 ) ) \Rightarrow 1 \\
&(\text{define } b ( \text{ flip } 0.3 ) ) \Rightarrow 0 \\
&(\text{define } c ( \text{ flip } 0.3 ) ) \Rightarrow 1 \\
&(+ a b c ) \Rightarrow 2
\end{align*}
\]
Church

\[
\text{(define } a \ ( \text{flip } 0.3 ) \ ) \implies 1 \ 0 \\
\text{(define } b \ ( \text{flip } 0.3 ) \ ) \implies 0 \ 0 \\
\text{(define } c \ ( \text{flip } 0.3 ) \ ) \implies 1 \ 0 \\
(+ \ a \ b \ c ) \implies 2 \ 0
\]
Church

(define a ( flip 0.3 )) => 1 0 0
(define b ( flip 0.3 )) => 0 0 0
(define c ( flip 0.3 )) => 1 0 1
(+ a b c ) => 2 0 1
Church

(define a ( flip 0.3 )) => 1 0 0
(define b ( flip 0.3 )) => 0 0 0
(define c ( flip 0.3 )) => 1 0 1
(+ a b c ) => 2 0 1...
Church

(define a ( flip 0.3 ) ) => 1 0 0
(define b ( flip 0.3 ) ) => 0 0 0
(define c ( flip 0.3 ) ) => 1 0 1
(+ a b c ) => 2 0 1...

probability / frequency

0 1 2 3
Church

(define \texttt{a} ( \texttt{flip 0.3} )) \Rightarrow 1 0 0
(define \texttt{b} ( \texttt{flip 0.3} )) \Rightarrow 0 0 0
(define \texttt{c} ( \texttt{flip 0.3} )) \Rightarrow 1 0 1
(+ \texttt{a} \texttt{b} \texttt{c} ) \Rightarrow 2 0 1...

Sampling semantics

Distribution semantics
flip is a procedure which returns a sample from a fair coin. That is, it’s a sampler or simulator.

Or

flip is itself a characterization of the distribution over true and false.
Church

When we think about probabilistic programs we will often move back and forth between these two views, emphasizing either the sampling perspective or the distributional perspective.

Although for our the proof part of our presentation, we will mostly focusing on sampling perspective.
(Core) Church Syntax

\[ v ::= c \mid x \mid (\text{lambda} \ (x_1 \ x_2 \ \ldots) \ e) \]

\[ e ::= v \mid (\text{if} \ e_1 \ e_2 \ e_3) \mid (\text{define} \ x \ e) \mid (\text{quote} \ e) \]

\( v: \text{Value} \)

\( x: \text{Variable} \)

\( e: \text{Church expression} \)

\( c: \text{Primitive Constant} \)
An Naive Multi-stepping of Church Expression

Unlike TFL but like IMP

Church had a similar concept of “Store”

\(<e, \sigma> \rightarrow^* <e', \sigma'>\)

Although a better word for that is actually “environment”
An Naive Multi-stepping of Church Expression

Let $\mu(e, \sigma)$ be the conditional distribution of expression $e$ under environment $\sigma$

$<e, \sigma> \rightarrow^* <v, \sigma>$ where $v$ is a sample from $\mu(e, \sigma)$

$<c, \sigma> \rightarrow^* <c, \sigma>$

$<x, \sigma> \rightarrow^* <\sigma[x], \sigma>$

$<(\text{if } e_1 e_2 e_3), \sigma> \rightarrow^* <e_2, \sigma>$ if $<e_1, \sigma> \rightarrow^* <\text{True}, \sigma_2>$

$<(\text{if } e_1 e_2 e_3), \sigma> \rightarrow^* <e_3, \sigma>$ if $<e_1, \sigma> \rightarrow^* <\text{False}, \sigma_2>$
An Naive Multi-stepping of Church Expression

If $\langle e, \sigma \rangle \rightarrow^{*} \langle v, \sigma \rangle$ then

$\langle\text{define } x \ e\rangle, \sigma \rightarrow^{*} \langle(), \sigma[x\rightarrow v]\rangle$

If $\langle e_1, \sigma \rangle \rightarrow^{*} \langle p, \sigma_2 \rangle = \langle\text{lambda } (x_1 \ x_2 \ldots) \ e\rangle, \sigma_2\rangle$

and $\langle e_2, \sigma \rangle \rightarrow^{*} \langle v_1, \sigma \rangle$, $\langle e_3, \sigma \rangle \rightarrow^{*} \langle v_2, \sigma \rangle$, ... then

$\langle e_1 \ e_2 \ e_3\ldots\rangle, \sigma \rightarrow^{*} \langle[v_1/x_1][v_2/x_2][\ldots]e, \sigma \rangle$
We can evaluate Church Languages to define the structure (i.e. the relationships between variables) of a probabilistic model.

But we also want probabilistic inferences: given the observation, assume the relationship assume holds, what is the probabilistic distribution of the unobserved variables?
(query

  (define var₁ e₁) ;;
Variable Definition

  (define var₂ e₂)

  ... 

  e
  ;; Query

  (p var₁ ...)
  ;; Conditions
(query

(define a ( flip 0.3 ) )

(define b ( flip 0.3 ) )

(define c ( flip 0.3 ) )

(+ a b c )                 Query

(- (+ a b ) 1 )           Condition

What is a+b+c? assuming that a+b is 1.
(query

  (define a ( flip 0.3 ))
  (define b ( flip 0.3 ))
  (define c ( flip 0.3 ))

  (+ a b c )              Query

  (- ( + a b ) 1 )            Condition
)

47
Memoization

Strength
Laziness
Team
Pulling
Winners
Memoization

(define \textbf{strength} (\texttt{mem} ( \lambda \ (\text{person}) \ (\text{gaussian} \ 10 \ 3) )))

(define \textbf{lazy} (\lambda \ (\text{person}) \ (\text{flip} \ 0.1) ))

(define \textbf{pulling} \text{person}) (\text{if} \ (\text{lazy} \ \text{person})

\quad (/ \ (\text{strength} \ \text{person}) \ 2) \ (\text{strength} \ \text{person})

)
Memoization

(define total-pulling team (sum (map pulling team)))

(define (winner team1 team2)
  (if (> (total-pulling team1) (total-pulling team2)) team1 team2))
An Naive Multi-stepping of Memoization

Formally, Memoization is equivalent to (but not implemented in):

If \( <e_1, \sigma> \rightarrow^* <(\text{lambda} (x_1 \ x_2 \ ...) \ e), \sigma_2> \)

\( <(\text{mem} \ e_1), \sigma \rightarrow <(\text{lambda} (x_1 \ x_2 \ ...) \ \sigma'[x_1 \ x_2 \ ...]), \sigma'> \)

Where \( \sigma' \) is an environment that stored all possible values of \( (x_1 \ x_2 \ ...) \) and match each of the \( \sigma'[x_1 \ x_2 \ ...] \) to a sample from the result in \( (e \ x_1 \ x_2 \ ...) \)
More Variables with Memoization

Memoization in some sense support infinite many variables.

This provide it with support to nonparametric graphical models like HDP, because we need infinitely many variables in such models.

It also makes implementing large finite graphical models like LDA easier: not all words have to be defined as variable. They can be calls to function instead.
More Variables with Memoization

(define samples
  (mh-query 100 100)

(define document->mixture-params (mem (lambda (doc-id) (dirichlet (make-list (length topics) 1.0))))))

(define topic->mixture-params (mem (lambda (topic) (dirichlet (make-list (length vocabulary) 0.1))))))

(define document->topics (mem (lambda (doc-id)
  (repeat doc-length (lambda () (multinomial topics (document->mixture-params doc-id)))))))

(define document->words (mem (lambda (doc-id)
  (map (lambda (topic)
    (multinomial vocabulary (topic->mixture-params topic)))
    (document->topics doc-id))))))

(map topic->mixture-params topics)

(and
  (factor-equal? (document->words 'doc1) doc1)
  (factor-equal? (document->words 'doc2) doc2)
  (factor-equal? (document->words 'doc3) doc3)
  (factor-equal? (document->words 'doc4) doc4)
  (factor-equal? (document->words 'doc5) doc5)))
Semantic Correctness

We've talked a lot about the Syntax of Church.

Why are the correct?

Do they really imply a probabilistic distribution that is determinable?

Answer: Yes*

*with terms of conditions
Semantic Correctness

Recall we define $\mu(e, \sigma)$ be the conditional distribution of expression $e$ under environment $\sigma$, what’s the implementation here on multi-stepping?

Example:

The distribution $(\text{flip } 0.3)$ have probability space $\{0, 1\}$

Each with probability 0.3 and 0.7

$(\text{flip } 0.3), \sigma \rightarrow <1, \sigma>$ with probability 0.3

$(\text{flip } 0.3), \sigma \rightarrow <0, \sigma>$ with probability 0.7
Semantic Correctness

Define \( \text{weight}(e, s, \sigma) \) to be the the probabilities of one of its possible future complete finite small steps \( s \) evaluations starting from \( e \).

When \( \Sigma_s \text{weight}(e, s, \sigma) = 1 \), we say that \( e \) is admissible in \( \sigma \).

Example:
\[
\Sigma_s \text{weight}((\text{flip 0.3}), s, \sigma) = 1
\]

Non-Example:
\[
\Sigma_s \text{weight}((\text{if (flip 0.3) 1 (infinite_loop)}), s, \sigma) = 0.3
\]
Define weight(e, s, σ) to be the the probabilities of one of its possible future complete finite small steps s evaluations starting from e.

When \( \Sigma_s \text{weight}(e, s, \sigma) = 1 \), we say that e is admissible in \( \sigma \).

Inductive formula of weight:

If \( \langle e, \sigma \rangle \rightarrow \langle e', \sigma' \rangle \) with probability p, we have:

\[
\text{weight}(e', s, \sigma') \times p = \text{weight}(e, (e', s), \sigma')
\]
Semantic Correctness

Define weight(e, s, σ) to be the the probabilities of one of its possible future complete finite small steps s evaluations starting from e.

When $\Sigma_s \text{weight}(e, s, \sigma) = 1$, we say that e is admissible in $\sigma$.

Claim: $\forall \sigma, \forall e$ that does not use mem, then $\Sigma_s \text{weight}(e, s, \sigma)$ is well defined and always less than or equal to 1.

Proof: Induction on depth of the Small Steps.

Base Case: c, x, and Simple Primitive Probabilistic Distributions.
Semantic Correctness

When we use mem, prove the well defined property of admissibility on the number of called to mem function.

Base case: no call to mem, proved.

Induction step: assume called to n mem satisfy the admissibility property. Consider another expression called mem n + 1 times.

We have \( <\text{mem } e_1, \sigma> \rightarrow <\text{lambda } (x_1 x_2 \ldots) \sigma'[x_1 x_2 \ldots]), \sigma'> \).
This essentially reduces to the cases where we call mem n times.
Semantic Correctness: Query is well defined

Recall that in query, we need to have observation \( p \) hold.

Let \( V \) be an variable in the query. What’s the conditional distribution of \( V \) given by the fact that \( p \) holds?

\[
P(V \mid \langle p \; V \rangle, \sigma) \rightarrow^* \langle \text{True}, \sigma' \rangle)
\]

If we can show that this is well defined, then we can show that the function query is well defined.
Consider variable $V$ follow the distribution $\mu(e, \sigma)$. Assume $e$ is admissible.

Then, as long as $(p \ V)$ is also admissible for all values of $V$:

$$P(V \mid <(p \ V), \sigma> \rightarrow^* <\text{True}, \sigma'>)$$

$$= P(V, <(p \ V), \sigma> \rightarrow^* <\text{True}, \sigma'>) \times P(<(p \ V), \sigma> \rightarrow^* <\text{True}, \sigma'>)$$

Is well defined.
Inference Correctness

We also like to show the inference work properly in terms of value, otherwise the program could just randomly output answer and that’s not we want.

The most common one used in Church and the one had been show up in most of the literature is Metropolis-Hasting Query (with syntax mh-query), based on the Metropolis-Hasting MCMC Algorithm.

Unfortunately proving Inference Correctness in MH require prerequisite in Measure Theory, so I will essentially skip the proof for this part. I’ll just give you some snapshots of the proof to let you know that it is provable based on small step.
Inference Correctness

Define a metric space $||e||$.

Idea: $d(x, x) = 0$, $d(c, d) = |c - d|

$$d(e_1 e_2, e_3 e_4) = d(e_1, e_3) + d(e_2, e_4)$$

This induce a Borel Measurable set on all expression.

Subsequently, a trace, which can be seen as $e^n$, can be also defined with a Borel Measure.
Inference Correctness

$P(e, s)$ is a measurable function which can be used to provide us with a definition of what sample distribution at step $n$ of the MH Markov chain would be, and the convergence of such function can be proved.
Demo

https://probmods.org/play-space.html
Questions?
References

http://probabilistic-programming.org


Videos:


Noah Goodman: “Stanford Seminar-Stories from CoCoLab: Probabilistic Programs, Cognative Modeling, & Smart Web Pages” https://www.youtube.com/watch?v=rGz0nagM-TA