Probabilistic Programming From the Ground Up: Lecture 1

Steven Holtzen

10 June 2024

Introduction

Probabilistic programming languages (PPLs) are of interest in PL and AI. In answering the question of why we might want to study them, we consider these two perspectives.

PL Perspective

The goal of PL is to reason about computer programs. However, many programs deal with uncertainty inherent in their problem domains. For instance:

- Randomized algorithms, such as primality testing and others, depend on random values which are not known ahead of time.
- Distributed systems and networking need to deal with consensus decisions, such as leader selection, which may require tie-breaking.
- Differential privacy, a property ensuring that programs don't leak identity information, requires that noise be added to data.

In order to analyze and reason about programs with uncertainty, it is useful to define probabilistic semantics for programming languages. The study of probabilistic programming languages began over 40 years ago with Kozen's [2] initial formulation of probabilistic semantics.

AI Perspective

Artificial intelligence research is largely concerned with building *agents* that can reason rationally about the world. In order for an agent to reason about the world, a model is needed to represent it. Building a deterministic model of the world is infeasible due to its scale, so it is useful to apply probabilistic reasoning to "smooth over the fuzzy parts" and simplify the underlying structure. The Turing award laureate Judea Pearl championed this worldview in the AI community, but initially had to fight for any sort of attention; he even created a new conference in which probabilistic AI papers could be published.

Lecture series overview

The goal of this lecture series is to learn to build and reason about probabilistic programming languages.

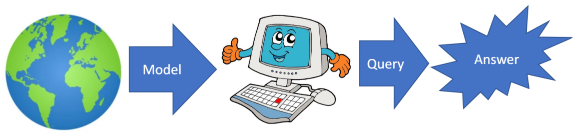

Figure 1: World Representation

1. Lecture 1 (this lecture): Syntax and semantics of PPLs
2. Lecture 2: Sampling
3. Lecture 3: Tractability and expressivity
4. Lecture 4: Special topics

How does Probabilistic Programming make reasoning better?

- *Networks of Plausible Inference* - Judea Pearl [3]
- Ordinary logic is monotonic, and is insufficient for a changing world without modifications to support modifying or deleting existing facts (e.g. to forget irrelevant facts).

Probabilistic programs: Programs that denote probability distributions.

An example PPL program:

```
// x and y would get tt with probability 1/2
x <- flip 1/2;
y <- flip 1/2;
return x ∨ y
```

Probability that return value is true is 3/4. $[tt \mapsto 3/4, ff \mapsto 1/4]$

Given some finite sample space Ω: $\Omega = \{\text{tt}, \text{ff}\}$

A probability distribution is a function $\Omega \to [0, 1]$ such that

$$\sum_{\omega \in \Omega} \Pr[\omega] = 1$$

We use semantic brackets of this form: $[\![\cdot]\!]$

Tiny PPL

Syntax

In these notes we follow the convention of using “tt” to denote the *semantic* truth value, and “true” to denote the syntactic truth value.

```
# Pure terms
p ::= true | false
   | x
   | if p then p else p
   | p ∧ p
   | p ∨ p

# Effectful or probabilistic terms
e := x <- e; e
   | return p    # Take a pure thing and make it impure
   | flip θ     # Introduce randomness, where θ is a probability ∈ ℝ
```

An example program:

```
x <- return true
return x
```

Semantics

We denote the semantics of pure terms with respect to environments containing variables.

$$[\![p]\!] : \mathit{Env} \to \mathit{Bool}$$

$[\![p]\!]$ is a partial function. If p references some variable which is not in the environment, the value of $[\![p]\!]$ is undefined.

- $[\![x]\!]_p(x \mapsto tt) = tt$
- $[\![true]\!]_p(\rho) = tt$, where ρ is any env
- $[\![p_1 \wedge p_2]\!]_p(\rho) = [\![p_1]\!]_p(\rho) \wedge [\![p_2]\!]_p(\rho)$

Semantics of probabilistic terms:

Given an environment and value, there is a probability associated with it.

$$[\![e]\!] : \text{Env} \to (\text{Bool} \to [0,1])$$

$$[\![\text{flip } r]\!](\rho) = \{\text{tt} \mapsto r, \text{ff} \mapsto 1-r\}, r \in [0,1]$$

$$[\![\text{return } p]\!](\rho) = \forall v : v \mapsto \begin{cases} 1 & \text{if } [\![p]\!]_p(\rho) = v \\ 0 & \text{otherwise} \end{cases}$$

$$[\![x \leftarrow e_1; e_2]\!](\rho)(v) = \sum_{v'} [\![e_1]\!](\rho)(v) * [\![e_2]\!](\rho[x \mapsto v'])(v)$$

An example:

$$[\![x \leftarrow \text{flip } 1/2]\!](\rho)$$

Performing case analysis on x, being a pure term, we get:

$$\begin{aligned}&[\![x \leftarrow \text{flip } 1/2;\ \text{return } x]\!](\rho)(v)\\ &\quad = [\![\text{flip } 1/2]\!](\rho)(\text{tt}) * [\![\text{return } x]\!](\rho[x \mapsto \text{tt}])(v) + [\![\text{flip } 1/2]\!](\rho)(\text{ff}) * [\![\text{return } x]\!](\rho[x \mapsto \text{ff}])(v)\end{aligned}$$

How can we consider our events to be independent?

- We can because the only way to introduce randomness here is flip, which is independent.

The semantics of flip are defined monadically. Further reading on monadic probabilistic semantics can be found in [1] and [4].

Complexity of the semantics

The semantics of TinyPPL, though quite simple, is in fact computationally intractible. As it turns out, evaluating the probability distribution of a program is #P-hard; there is a relatively straightforward Cook reduction from #SAT to TinyPPL's semantics. (#SAT is the problem of determining the number of satisfying assignments for a given Boolean formula.) By Toda's theorem, #P oracles can decide quantified Boolean formulas in PH, meaning that there is no known algorithm which can compute the probability distribution of a TinyPPL program efficiently. In general,

$$\#\text{SAT}(\phi) = 2^{\#vars} * [\![\mathit{program}]\!](\phi)(\text{tt})$$

A full proof of this reduction is left as an exercise to the motivated reader.

Why do we care about complexity of denotational semantics, given that what the computer is running will be operational semantics? The complexity of the denotational semantics tells us how "good" we can be. We are forced to consider the computational price of adding new features such as tuples, functions, heaps, and so forth. TinyPPL is almost nothing, and yet it is still quite computationally inefficient; more features can only make things worse.

Examples of Systems in the Wild

- Stan from Columbia: https://mc-stan.org/
  -
- Pyro
- Tensorflow Probability: https://www.tensorflow.org/probability
- R2 from Microsoft: https://www.microsoft.com/en-us/research/project/r2-a-probabilistic-programming-system/

Can you elaborate on the difference between this and deep learning approaches?

- This is a Bayesian learning model: Different style from deep learning
  - An advantage of Probabilistic Programming is that it is very interpretable. This is more difficult with Deep Learning.

References

[1] Michèle Giry. “A Categorical Approach to Probability Theory”. In: *Categorical Aspects of Topology and Analysis*. Ed. by B. Banaschewski. Berlin, Heidelberg: Springer, 1982, pp. 68–85. ISBN: 978-3-540-39041-1. DOI: 10.1007/BFb0092872.

[2] Dexter Kozen. “Semantics of Probabilistic Programs”. In: *20th Annual Symposium on Foundations of Computer Science (Sfcs 1979)*. Oct. 1979, pp. 101–114. DOI: 10.1109/SFCS.1979.38.

[3] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Rev. 2. ed., transferred to digital printing. The Morgan Kaufmann Series in Representation and Reasoning. San Francisco, Calif: Morgan Kaufmann, 2009. ISBN: 978-1-55860-479-7.

[4] Norman Ramsey and Avi Pfeffer. “Stochastic Lambda Calculus and Monads of Probability Distributions”. In: *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’02. New York, NY, USA: Association for Computing Machinery, Jan. 2002, pp. 154–165. ISBN: 978-1-58113-450-6. DOI: 10.1145/503272.503288.