Lecture 1: Why Probabilistic Programming?[1]

Steven Holtzen
s.holtzen@northeastern.edu

September 7, 2023

[1] CS7470 Fall 2023: Foundations of Probabilistic Programming.

This course is all about probabilistic programs. To make a course about something, it has to be suitably important. What are probabilistic programs, and what makes them worth studying? In short, **probabilistic programming languages** (PPLs) use the syntax and semantics of programs to define probability distributions.[2] In a typical programming language – like OCaml, Python, or C – a program's execution is deterministic: program outputs are fully determined by inputs (and perhaps an environment). For instance, the following tiny program produces an output of 15:[3]

[2] **Syntax** is a textual representation of a program. **Semantics** tells us what a program means. We will see more about these in later lectures.

[3] Throughout the class and in these notes, we will use functional ML-style syntax for programs.

```
let x = 5 in
let y = 10 in
x + y
```

Probabilistic programs define probability distributions on their outputs. To do this, they have extra syntax and semantics for introducing and manipulating uncertain quantities. For instance, the following tiny probabilistic program flips two coins and returns the value `true` if both coins are heads:[4]

[4] The syntax `flip` θ produces a random Boolean value that is `true` with probability θ and `false` with probability $1-\theta$.

```
let x = flip 0.5 in
let y = flip 0.5 in
x && y
```

This probabilistic program does not output a single value like `true` or `false`: rather, it outputs a *probability distribution* over all the possible output values. In this case, it outputs the following **probability distribution**, which is a function that associates values with probabilities:

$$[\texttt{true} \mapsto 0.25, \qquad \texttt{false} \mapsto 0.75].$$

Why would one want programs that manipulate probabilities in this way, and what are the challenges that come along with this significant enrichment of our usual programming model? There are many reasons, and today PPLs are a vibrant area of active research and development, but there is one theme in particular that I aim to explore in this course: that *PPLs are best understood as a cross-disciplinary effort between programming languages and artificial intelligence*. This is because many of the motivations and methods necessary for designing, building, and studying PPLs stem from core research questions and methods studied from both of these areas. In the following sections, we will explore this intersection, and survey

some of the modern topics in PPL research that we will encounter during this course.

1 The Programming Languages Perspective

Our first collection of answers to the question of "why probabilistic programming languages" comes from formal study and application of programming languages. One of the main goals of programming language research is to develop programming languages foundations that enable the practical development of robust and reliable software systems. Here is a small collection of some of the themes and success stories within programming languages towards this goal over the past decades:

- *Deductive verification* and proof systems that verify that programs satisfy certain specifications by the application of proof rules that reason abstractly about program's behavior. Examples include the weakest pre-condition calculus and Hoare logic, separation logic, refinement logics, etc. These techniques are typically not fully automated and require some sort of guidance from the user in order to be applied.
- *Type systems* that verify that programs satisfy certain properties. Type systems range broadly in sophistication. Some type systems embed entire program logics [Jhala et al., 2021]. Others are more lightweight. Almost all programmers today interact with some form of type system while writing their software.
- *Semantic foundations*. How do we formally specify what it means to "run a program"? Can we associate programs with more abstract mathematical objects that are easier to reason about? These are core questions within programming language research that fall under the category of semantics, and the area is surprisingly deep: it can be difficult to precisely describe what it means to execute a program.[5]
- *Software model checking* that statically verify that programs satisfy certain properties by somehow representing the entire state space of the program. Common techniques used in this area are SAT/SMT solvers such as Z3 [De Moura and Bjørner, 2008] and bounded model checking.

[5] See Pierce [2002] Chapter 3 for a discussion the different kinds of semantics that have been investigated for programs.

These techniques are applied broadly in industry in places like NASA [Calcagno et al., 2015], Facebook [Distefano et al., 2019], Microsoft [Ball et al., 2004], Amazon [Newcombe et al., 2015], and many others.

However, there is a fundamental problem: what happens when the software system you want to reason about has *inherent uncertainty* in it? When you look hard enough for it, you encounter uncertainty in almost every sufficiently large software system:

1. *Interaction with the environment*. The world is complicated and software often has to interact with it. Hardware can fail randomly, physical processes (like turbulant flow) are computationally intractable to precisely model, and sometimes software has to interact with human beings. All these situations involve inherent uncertainty that software systems need to be able to deal with robustly.

2. *Randomized algorithms*. Some algorithms only output correct answers with a certain probability [Motwani and Raghavan, 1995]. If we want to verify that an implementation of these algorithms is correct, we need to be able to formally reason about programs with uncertain outcomes.[6]

3. *Machine learning*. Today, not all aspects of programs are written down as code: often a program's behavior is determined by components that are learned from data. Machine learning is riddled with uncertainty: both the learning process and the prediction process are inherently uncertain processes.

[6] Verification of randomized algorithms was the original context in which Kozen [1979] proposed the study of giving a formal semantic foundation to probabilistic programs.

The net result is that, in order to verify and formally reason about today's software systems, we often need to reason about uncertainty. This requires revisiting each of the above research themes: weakest pre-condition calculus becomes weakest pre-expectation calculus [McIver and Morgan, 2005], model checking becomes probabilistic model checking [Katoen, 2016], semantic foundations must incorporate probabilities [Kozen, 1979], and so on. In short, PPL research within the programming languages community has been animated by a motivation to handle the need of reasoning formally about the uncertainty that naturally arises in software systems.

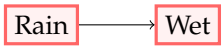
2 *The Artificial Intelligence Perspective*

One of the main intellectual endeavors within artificial intelligence is the design and implementation of intelligent agents that reason rationally about their environments. This inherently requires reasoning about uncertainty, an within the AI community there has been an enormous effort to scale, automate, and productize systems for automatically reasoning about uncertainty.[7] Within the AI community, these tools are broadly known as *probabilistic models*.

[7] There is even a major conference focusing on this problem called *Uncertainty in Artificial Intelligence (UAI)*.

A **probabilistic model** is a language for describing a probability distribution. The most popular and widely-known probabilistic model within the AI community is the *probabilistic graphical model* (PGM), a graphical representation of uncertainty introduced by Pearl [1988].[8] PGMs use graphical notation to describe the relationships between random quantities. For instance, the relationship "if it is raining, then the ground is likely to be wet" can be represented graphically as:

[8] Several good textbooks on PGMs have been written in recent years, including Darwiche [2009] and Koller and Friedman [2009].



The directed edges denote dependencies: the fact that it is wet is statistically dependent on whether or not it is raining. The above is an example of a *directed graphical model* or *Bayesian network*.[9] Seen from this vantage point, probabilistic programming languages are a natural evolution of existing graphical languages for representing uncertainty; indeed, many early PPL papers explicitly reference this lineage [Sato and Kameya, 1997, Ramsey and Pfeffer, 2002, Goodman et al., 2012].

[9] There are other kinds of graphical models that are undirected, such as factor graphs and Markov networks.

Concretely, we can represent the above rain–wet scenario using a simple probabilistic program:

```
let rain = flip 0.01 in
let wet = if rain then flip 0.9 else flip 0.01 in
wet
```

This program expresses that, on a given day, there is a 1% chance that it is raining. In the event that it rains, then it is wet 90% of the time; if it is not raining, then it is wet 1% of the time. The program then outputs the variable that indicates whether or not it is wet. Notice how, rather than use a graph structure to describe relationships between uncertain quantities, a PPL uses standard programming constructs (such as `if`-statements).

WHAT DO AI RESEARCHERS use probabilistic models for? One of the primary applications is *Bayesian learning*. AI researchers want to design intelligent agents that reason rationally about their environment. A core part of this process is the ability to incorporate new information into the agent's model of the world as it is discovered. As a very simple example of this process, suppose we observe that it is wet and wish to to infer what the probability is that it rained. This process of observing new information and updating our beliefs can be encoded directly into the program using an `observe` keyword:

```
let rain = flip 0.01 in
let wet = if rain then flip 0.9 else flip 0.01 in
observe wet;
rain
```

This program *does not* output the distribution $[\texttt{true} \mapsto 0.01, \texttt{false} \mapsto 0.99]$! By observing that the ground was wet, we correspondingly increase our estimate of the probability that it rained. Intuitively, this is accomplished by discarding all possible program executions that violate the `observe`'s constraints: in this case, all paths in which `wet = false` are discarded, and a new *posterior probability* of `rain` is computed. The outputted posterior probability on `rain` is:

$$\left[\texttt{true} \mapsto \frac{0.01 * 0.9}{0.01 * 0.9 + 0.99 * 0.01}, \texttt{false} \mapsto \frac{0.99 * 0.01}{0.01 * 0.9 + 0.99 * 0.01}\right]$$

This process of updating beliefs is called **Bayesian conditioning**, and we will discuss observation in more detail a later lecture. Computing the posterior probability of an event is called **Bayesian inference**.[10]

This ability to update our beliefs is critical for many AI applications that involve probabilistic reasoning:

- *Medical diagnosis*. A doctor posits the relationships between symptoms and diseases, and wishes to infer the posterior probability of disease given observations of symptoms.
- *Image processing*. One assumes an underlying generative model of images, observes a particular image, and infers the posterior probability on features of the generative model. This has been used for infilling, denoising, and other forms of image processing.
- *Game playing*. Many kinds of games involve *imperfect information*: players don't get to know everything about each other. In imperfect information games, in order to play optimally, it is necessary to infer aspects of the other player's state given observations about their actions.
- *Error correcting codes*. Today's software and hardware systems need to be robust to events that randomly corrupt data. The problem of recovering possibly corrupted data can be formulated as a Bayesian inference problem.

There are many more applications than listed here. Think of some for yourself!

[10] As a preview, you have probably seen the rule for Bayes's rule before. Let $\Pr(X, Y)$ be a probability distribution on two random variables X and Y. Then, the posterior probability of X given $Y = y$ is:

$$\Pr(X = x \mid Y = y) = \frac{\Pr(X = x, Y = y)}{\Pr(Y = y)}.$$

3 *Modern Research Directions & Challenges*

We have seen two very different avenues that arrive at the same destination: probabilistic programming languages are well-motivated from both an AI and PL perspective. So, at this point you should be asking: if probabilistic programming languages are so great and useful, why isn't everyone using them? Now we will survey some of the

modern challenges in building practical PPLs that we will encounter during this course.

- *Hardness of inference*. The first and perhaps primary challenge for building practically useful PPLs is the fundamental computational complexity of probabilistic inference. We will see that, even for extremely restricted languages, inference will be worst-case computationally intractable. This means that all PPLs walk a delicate line between *expressivity and tractability*: a more expressive language gives the user more freedom to write programs, but makes the inference task correspondingly harder. To address this, PPL researchers develop inference strategies that identify and exploit various kinds of program structure, such as independences, smoothness, unimodality, and other kinds of structure; we will see much more on this later in the course.
- *Semantic foundations*. PPLs give the user an incredible amount of flexibility in defining probability distributions: programs may have recursion, conditioning, higher-order functions, etc. Formally reasoning about probability distributions on these objects is very hard! We will encounter this in more detail when we start writing down formal semantics for languages in later lectures.
- *Deductive verification*. One way around the computational complexity of inference is to reason about what probabilistic programs do *without running them*: this is the goal of deductive verification strategies that aim to give proof rules for reasoning about probabilistic programs. Deductive verification is an active area of PPL research that we will explore in later lectures.

This is just a sample of some of the modern ongoing areas of research within PPLs today. To see more, check out some of the following research venues that regularly publish PPL papers:

- Programming Language Design and Implementation (PLDI)
- Uncertainty in Artificial Intelligence (UAI)
- Symposium on Programming Languages (POPL)
- Object-oriented Programming Languages & Applications (OOPSLA)
- Workshop on Languages for Inference (LAFI)
- Neural Information Processing Symposium (NeurIPS)

References

Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K Rajamani. Slam and static driver verifier: Technology transfer of formal methods inside microsoft. In *Integrated Formal Methods: 4th International Conference, IFM 2004, Cnaterbury, UK, April 4-7, 2004. Proceedings 4*, pages 1–20. Springer, 2004.

Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In *NASA Formal Methods Symposium*, pages 3–11. Springer, 2015.

Adnan Darwiche. *Modeling and reasoning with Bayesian networks*. Cambridge university press, 2009.

Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W O'Hearn. Scaling static analyses at facebook. *Communications of the ACM*, 62(8):62–70, 2019.

Noah Goodman, Vikash Mansinghka, Daniel M Roy, Keith Bonawitz, and Joshua B Tenenbaum. Church: a language for generative models. *arXiv preprint arXiv:1206.3255*, 2012.

Ranjit Jhala, Niki Vazou, et al. Refinement types: A tutorial. *Foundations and Trends® in Programming Languages*, 6(3–4):159–317, 2021.

Joost-Pieter Katoen. The probabilistic model checking landscape. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 31–45, 2016.

Daphne Koller and Nir Friedman. *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

Dexter Kozen. Semantics of probabilistic programs. In *20th Annual Symposium on Foundations of Computer Science (sfcs 1979)*, pages 101–114. IEEE, 1979.

Annabelle McIver and Carroll Morgan. *Abstraction, refinement and proof for probabilistic systems*. Springer Science & Business Media, 2005.

Rajeev Motwani and Prabhakar Raghavan. *Randomized algorithms*. Cambridge university press, 1995.

Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How amazon web services uses formal methods. *Communications of the ACM*, 58(4):66–73, 2015.

Judea Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan kaufmann, 1988.

Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

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*, pages 154–165, 2002.

Taisuke Sato and Yoshitaka Kameya. Prism: a language for symbolic-statistical modeling. In *IJCAI*, volume 97, pages 1330–1339, 1997.