Propositional Probability II: A Propositional PPL[1]

Steven Holtzen
s.holtzen@northeastern.edu

September 17, 2023

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

Now we are ready to ask: *can we use propositional logic as the basis for an effective probabilistic model*? The first question is: what is our sample space? It's clear so far that a logical choice of sample space is the set of all possibly instances for a fixed set of Boolean formulae; this sample space will have size 2^n, where n is the number of propositional variables. Now, if we want to efficiently evaluate queries over this sample space, we need need two more components:

1. A way to efficiently represent a probability distribution on the sample space;

2. A way to efficiently represent queries over that sample space.

Let's tackle (1) first. What is an alternative representation of a distribution that avoids the space-explosion we saw in the lookup-table representation? We will need to be more clever about how we represent probabilities. One useful choice is to assume that all random variables are *independent from each other*, and therefore we can concisely describe a joint distribution over all of them:

Definition 1 (Fully factorized probabilistic model)**.** *Let $X_1, X_2, \cdots, X_n$ be jointly independent random variables (i.e., for any pair X_i, X_j, it is the case that $X_i \perp\!\!\!\perp X_j$). Then, a fully-factorized probabilistic model is a collection of n probability lookup tables $\Pr(X_i)$, one for each i. The joint probability is computed as $\Pr(X_1, X_2, \cdots, X_n) \triangleq \prod_i^n \Pr(X_i)$.*

Observe that a fully-factorized model only requires $O(\sum_i |X_i|)$ space, which is significantly smaller than $|\Omega|$.[2] This is a big improvement over plain lookup tables, but it comes at a cost of expressivity: there are some distributions that cannot be represented in a fully-factorized way.

[2] The notation $|X|$ refers to the size of a random variable's co-domain.

1 *A Propositional PPL (Prop)*

These are fresh semantics and there may be bugs! If you see any let me know.

- We will introduce a simple probabilistic programming language based on propositional logic called Prop.

- The syntax of Prop has two parts: a query φ, written in propositional logic, and a fully-factorized distribution w on propositional variables:

Syntax of Prop:

$$
\begin{aligned}
&\varphi ::= x \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \neg\varphi \mid \texttt{true} \mid \texttt{false} \\
&w ::= [x \mapsto \theta_x, y \mapsto \theta_y, \cdots] \\
&\texttt{p} ::= (\varphi, w)
\end{aligned}
$$

Example Prop program:

$$(x \vee y,\ [x \mapsto 0.1, y \mapsto 0.2])$$

- In order to interpret programs, we need to define a **propositional universe**, which tells us which propositional variables the program is defined over. We denote this universe Γ, which is a finite set of propositional variables.

- A syntactic term in Prop is **well-typed by universe** Γ if every propositional variable that is free in Prop can is in Γ. If a term is well-typed by Γ, we write $\Gamma \vdash \texttt{p}$. We can formalize this description of Γ inductively on Prop programs. First, we describe whether or not a propositional term is well-typed, $\Gamma \vdash \varphi$:

$$\Gamma \vdash \texttt{true} \qquad \Gamma \vdash \texttt{false} \qquad \frac{x \in \Gamma}{\Gamma \vdash x}$$

$$\frac{\Gamma \vdash \alpha \qquad \Gamma \vdash \beta}{\Gamma \vdash \alpha \wedge \beta}$$

  The rest of the propositional connectives proceed similarly.

- Similarly, we can type the w terms:

$$\frac{x \in \Gamma \qquad \Gamma \vdash r}{\Gamma \vdash [x \mapsto \theta, r]} \qquad \Gamma \vdash []$$

- Finally, we can type programs:

$$\frac{\Gamma \vdash \varphi \qquad \Gamma \vdash w}{\Gamma \vdash (\varphi, w)}$$

- We can interpret a propositional universe Γ as the set of all propositional instances that can be formed from variables in Γ; we write this set as $[\![\Gamma]\!]$.

Denotational semantics

- Denotational semantics of Prop: we want to associate every well-typed program in Prop with the probability that φ holds according to the fully-factorized distribution described by w

- **Goal**: Define a map $[\![\Gamma \vdash \texttt{p}]\!] : [0,1]$ that maps well-typed terms from Prop to real values.

For more on the history and context of different styles of semantics, see Pierce [2002, Chapter 3]

- We will need semantics for φ and the map m in order to interpret $\texttt{p}$. They have the following types:
  - $[\![\Gamma \vdash \varphi]\!]$ maps propositional formulae to the set of all instances that model them over universe drawn from Γ:
$$[\![\Gamma \vdash \varphi]\!] \triangleq \{I \in [\![\Gamma]\!] \mid I \models \varphi\} \tag{1}$$
  - $[\![\Gamma \vdash w]\!] : \Gamma \to \mathbb{B} \to [0,1]$ produces a map from assignments to propositional variables to real values:
$$[\![\Gamma \vdash [x_1 \mapsto \theta_1, \ldots, x_n \mapsto \theta_n]]\!](x_i)(\texttt{true}) \triangleq \theta_i \tag{2}$$
$$[\![\Gamma \vdash [x_i \mapsto \theta_1, \ldots, x_n \mapsto \theta_n]]\!](x_i)(\texttt{false}) \triangleq 1 - \theta_i \tag{3}$$

  We need to assume a propositional universe so that these equations are well-defined. Assume that all instances are defined on all free variables in $\texttt{p}$.

- For notational convenience we define the probability of an instance $[\![\Gamma \vdash w]\!](I)$ as the product of probabilities of each variable in the instance. For instance,
$$[\![(x \mapsto 0.1, y \mapsto 0.3)]\!](x, \overline{y}) = 0.1 * 0.7.$$
  Formally, we will write:
$$[\![\Gamma \vdash w]\!](I) = \prod_{[x_i \mapsto v] \in I} [\![w]\!](x_i)(v) \tag{4}$$

- Now we are ready to interpret Prop programs as the sum of the probabilities of each model:
$$[\![\Gamma \vdash (\varphi, w)]\!] \triangleq \sum_{I \in [\![\Gamma \vdash \varphi]\!]} [\![\Gamma \vdash w]\!](I). \tag{5}$$

- Is this a *useful* PPL? What kinds of programs can we write in it? Exercise: write a Prop program that computes the probability that a dice roll is even.

2 *Big-step semantics*

- Our denotational model does not tell us how to efficiently compute probabilities
- **Goal**: given a Prop program, *efficiently evaluate it*

  Running example: The simple Prop program:

```
(x ∨ y, [x ↦ 0.1, y ↦ 0.3])
```

- Worst-case hardness: computing $\Pr(\varphi_{ex})$ is NP-hard for arbitrary formulae.

  We will show this by reduction to the **SAT problem**: the problem of determining whether or not a formula has a model.

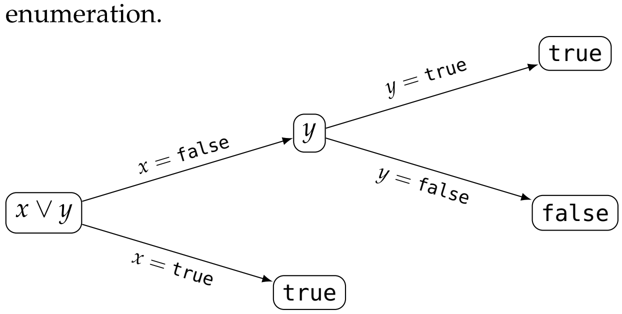
  Reduction: Let φ be a formula. Assign a probability of 0.5 to every assignment of variables. Then, φ has a model if and only if $\Pr(\varphi) > 0$.

- In fact, computing $\Pr(\varphi_{ex})$ is #P-hard. #P is the class of problems that are polytime reducible to counting the number of solutions to an arbitrary Boolean formula. See Goldreich [2008, Chapter 6] for more discussion on this complexity class, as well as Roth [1996].

Evaluating queries via search

- Need to avoid worst-case exponential behavior.

- *Idea*: We want to *search* through the space of models of the formula to potentially avoid exploring all possible worlds. Aim for better average-case/common-case behavior than the naive table enumeration.



- We will describe a relation $(\pi, \mathtt{p}) \Downarrow^e \mathbb{R}$:

  - π is an ordered list of propositional variables. We denote list concatenation as $x :: \pi$.
  - $\mathtt{p}$ is a Prop program
  - The result $\mathbb{R}$ will be equal to $[\![\mathtt{p}]\!]$

- Define $\varphi[x \mapsto v]$ as the substitution where we replace all instances of x with the value $v \in \{\mathtt{true}, \mathtt{false}\}$ in φ, and "simplify" φ by evaluating connectives until either (1) there are no more `true` or `false` constants, or (2) the formula is equal to `true` or `false`. Example: $(x \vee y)[x \mapsto \mathtt{true}] = y$.

- Now, let's describe our procedure for searching to solve our inference problem. We will define $\Downarrow^e$ inductively. The base-cases will be formulas without any free variables:

$$\frac{}{(\pi, (\texttt{true}, w)) \Downarrow^e 1}\ (\textsc{True}) \qquad \frac{}{(\pi, (\texttt{false}, w)) \Downarrow^e 0}\ (\textsc{False})$$

- Then, the inductive step:

$$\frac{(\pi, (\varphi[x \mapsto \texttt{true}], w)) \Downarrow^e v_1 \qquad (\pi, (\varphi[x \mapsto \texttt{false}], w)) \Downarrow v_2}{(x :: \pi, (\varphi, w)) \Downarrow^e w(x)v_1 + w(\overline{x})v_2}\ (\textsc{Split})$$

  Here we are being a bit loose with the weight function w; we can give an evaluation semantics for w as well, but assume for now that we can look up the weight of x and $\overline{x}$.

- Example derivation tree for our running example for the order $\pi = [x, y]$ and our program $(x \vee y,\ [x \mapsto 0.1, y \mapsto 0.3])$
- The **cost** of evaluating a formula φ for order π under these semantics is equal to the total number of derivations in the derivation tree.

Theorem 1. *Let* $\texttt{p}$ *be* Prop *program and* π *be an ordering on all variables in* $\texttt{p}$*, and let* $\Gamma \vdash \texttt{p}$*. If* $(\pi, \texttt{p}) \Downarrow v$*, then* $[\![\Gamma \vdash \texttt{p}]\!] = v$*.*

Proof. We will show this by structural induction on $\texttt{p}$. The base cases are straightforward: If $\texttt{p} = (\texttt{true}, w)$, then $(\pi, (\texttt{true}, w)) \Downarrow^e 1$. By definition, $[\![\Gamma \vdash (\texttt{true}, w)]\!] = \sum_{I \in [\![\Gamma \vdash \texttt{true}]\!]} w(I) = 1$; similar for the false case.

Induction hypothesis (IH): Let $(x :: \pi, (\varphi[x \mapsto \texttt{true}], w)) \Downarrow^e v_1$ and $(x :: \pi, (\varphi[x \mapsto \texttt{false}], w)) \Downarrow^e v_2$, and $(\pi, (\varphi, w)) \Downarrow^e v$. Assume by induction that $v_1 = [\![\Gamma \vdash (\varphi[x \mapsto \texttt{false}], w)]\!]$ and $v_2 = [\![\Gamma \vdash (\varphi[x \mapsto \texttt{true}], w)]\!]$.

We are done if we can show that $[\![\Gamma \vdash (\varphi, w)]\!] = [\![w]\!](x)(\texttt{true})[\![\Gamma \vdash (\varphi[x \mapsto \texttt{true}], w)]\!] + [\![w]\!](x)(\texttt{false})[\![\Gamma \vdash (\varphi[x \mapsto \texttt{false}], w)]\!]$.

To make progress we need to apply *Shannon expansion*: it is always the case that $[\![\Gamma \vdash (\varphi, w)]\!] = [\![\Gamma \vdash (\varphi \wedge x, w)]\!] + [\![\Gamma \vdash (\varphi \wedge \neg x, w)]\!]$. Then, observe that $[\![\Gamma \vdash (\varphi \wedge x), w]\!] = w(x)(\texttt{true})[\![\Gamma \vdash (\varphi[x \mapsto \texttt{true}])]\!]$ (this fact is not as straightforward to show as Shannon expansion, and proving it rigorously relies on a full inductive definition of substitution, so we elide this proof the time being).

The proof of Shannon expansion is a good exercise, and relies on splitting up the models of φ into 2 sets.

□

- What is the cost of enumerating the formula $x \vee y \vee z \vee w$ with these semantics? It will be linear-time for any order; a big improvement over the exhaustive enumeration!

- However, what about a formula like $(a \wedge b) \vee (c \wedge d)$? We will see how to handle this case next time.

References

Oded Goldreich. Computational complexity: a conceptual perspective. *ACM Sigact News*, 39(3):35–39, 2008.

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

Dan Roth. On the hardness of approximate reasoning. *Artificial Intelligence*, 82(1-2):273–302, 1996.