

Propositional Probability II: A Propositional PPL¹

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 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, \dots, 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 $\Pr(X_1, X_2, \dots, X_n) \triangleq \prod_i^n \Pr(X_i)$.*

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)

- 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:

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

Syntax of PROP:

```

1  $\varphi ::= x \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \neg \varphi \mid \text{true} \mid \text{false}$ 
2  $w ::= [x \mapsto \theta_x, y \mapsto \theta_y, \dots]$ 
3  $p ::= (\varphi, w)$ 

```

Example PROP program:

```

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

```

- 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 be in Γ . If a term is well-typed by Γ , we write $\Gamma \vdash p$. We can formalize this description of Γ inductively on PROP programs. First, we describe whether or not a propositional term is well-typed, $\Gamma \vdash \varphi$:

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

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

The rest of the propositional connectives proceed similarly.

- Similarly, we can type the w terms:

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

- Finally, we can type programs:

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

- 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 $\llbracket \Gamma \rrbracket$.

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 $\llbracket \Gamma \vdash p \rrbracket : [0, 1]$ that maps well-typed terms from PROP to real values.
- We will need semantics for φ and the map m in order to interpret p . They have the following types:
 - $\llbracket \Gamma \vdash \varphi \rrbracket$ maps propositional formulae to the set of all instances that model them over universe drawn from Γ :

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

$$\llbracket \Gamma \vdash \varphi \rrbracket \triangleq \{I \in \llbracket \Gamma \rrbracket \mid I \models \varphi\} \quad (1)$$

- $\llbracket \Gamma \vdash w \rrbracket : \Gamma \rightarrow \mathbb{B} \rightarrow [0, 1]$ produces a map from assignments to propositional variables to real values:

$$\llbracket \Gamma \vdash [x_1 \mapsto \theta_1, \dots, x_n \mapsto \theta_n] \rrbracket(x_i)(\text{true}) \triangleq \theta_i \quad (2)$$

$$\llbracket \Gamma \vdash [x_1 \mapsto \theta_1, \dots, x_n \mapsto \theta_n] \rrbracket(x_i)(\text{false}) \triangleq 1 - \theta_i \quad (3)$$

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 p .

- For notational convenience we define the probability of an instance $\llbracket \Gamma \vdash w \rrbracket(I)$ as the product of probabilities of each variable in the instance. For instance,

$$\llbracket (x \mapsto 0.1, y \mapsto 0.3) \rrbracket(x, \bar{y}) = 0.1 * 0.7.$$

Formally, we will write:

$$\llbracket \Gamma \vdash w \rrbracket(I) = \prod_{[x_i \mapsto v] \in I} \llbracket w \rrbracket(x_i)(v) \quad (4)$$

- Now we are ready to interpret PROP programs as the sum of the probabilities of each model:

$$\llbracket \Gamma \vdash (\varphi, w) \rrbracket \triangleq \sum_{I \in \llbracket \Gamma \vdash \varphi \rrbracket} \llbracket \Gamma \vdash w \rrbracket(I). \quad (5)$$

- 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:

1 $(x \vee y, [x \mapsto 0.1, y \mapsto 0.3])$

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

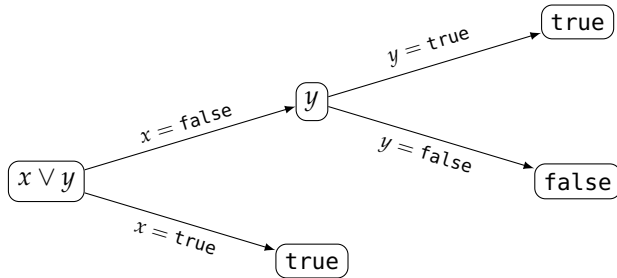
We will show this by reduction to the **SAT problem**: the problem of determining whether or not a formula has a model.

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 $(\pi, p) \Downarrow^e \mathbb{R}$:
 - π is an ordered list of propositional variables. We denote list concatenation as $x :: \pi$.
 - p is a PROP program
 - The result \mathbb{R} will be equal to $\llbracket p \rrbracket$
- Define $\varphi[x \mapsto v]$ as the substitution where we replace all instances of x with the value $v \in \{\text{true}, \text{false}\}$ 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: $(x \vee y)[x \mapsto \text{true}] = y$.

- 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:

$$\begin{array}{ll} (\text{TRUE}) & (\text{FALSE}) \\ (\pi, (\text{true}, w)) \Downarrow^e 1 & (\pi, (\text{false}, w)) \Downarrow^e 0 \end{array}$$

- Then, the inductive step:

$$\frac{(\text{SPLIT}) \quad (\pi, (\varphi[x \mapsto \text{true}], w)) \Downarrow^e v_1 \quad (\pi, (\varphi[x \mapsto \text{false}], w)) \Downarrow^e v_2}{(x :: \pi, (\varphi, w)) \Downarrow^e w(x)v_1 + w(\bar{x})v_2}$$

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 \bar{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 p be PROP program and π be an ordering on all variables in p , and let $\Gamma \vdash p$. If $(\pi, p) \Downarrow v$, then $\llbracket \Gamma \vdash p \rrbracket = v$.

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

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

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

To make progress we need to apply *Shannon expansion*: it is always the case that $\llbracket \Gamma \vdash (\varphi, w) \rrbracket = \llbracket \Gamma \vdash (\varphi \wedge x, w) \rrbracket + \llbracket \Gamma \vdash (\varphi \wedge \neg x, w) \rrbracket$. Then, observe that $\llbracket \Gamma \vdash (\varphi \wedge x, w) \rrbracket = w(x)(\text{true})\llbracket \Gamma \vdash (\varphi[x \mapsto \text{true}], w) \rrbracket$ (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 step for 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.