Exact Inference for Discrete Probabilistic Programs via Generating Functions

Fabian Zaiser

University of Oxford

ANR PPS workshop, 2023-01-05

Probabilistic Programming

- Suppose your coworker receives 10 calls per week on average.
- Each call is a scam independently with probability 20%.
- At the end of the week, your coworker is surprised that they got only one scam call.

What is the posterior probability distribution of the number of calls?

Probabilistic Programming

- Suppose your coworker receives 10 calls per week on average.
- Each call is a scam independently with probability 20%.
- At the end of the week, your coworker is surprised that they got only one scam call.

What is the posterior probability distribution of the number of calls?

Probabilistic program

$$X \sim \text{Poisson}(10)$$
$$Y \sim \text{Binomial}(X, 0.2)$$
$$\text{observe}\, Y = 1$$

Bayes' rule

$$\mathbb{P}[X = x \mid Y = 1] = \frac{\mathbb{P}[X = x] \times \mathbb{P}[Y = 1 \mid X = x]}{\mathbb{P}[Y = 1]}$$

Probabilistic program

$X \sim \text{Poisson}(10)$

$Y \sim \text{Binomial}(X, 0.2)$

observe $Y = 1$

Bayes' rule

$$\mathbb{P}[X = x \mid Y = 1] = \frac{\mathbb{P}[X = x] \times \mathbb{P}[Y = 1 \mid X = x]}{\mathbb{P}[Y = 1]}$$

$$= \frac{\overbrace{\text{prior} \times \text{likelihood}}^{\text{unnormalized posterior}}}{\text{normalizing constant}}$$

$$= \text{posterior}$$

Probabilistic program

$X \sim \text{Poisson}(10)$

$Y \sim \text{Binomial}(X, 0.2)$

observe $Y = 1$

Bayes' rule

$$\mathbb{P}[X = x \mid Y = 1] = \frac{\mathbb{P}[X = x] \times \mathbb{P}[Y = 1 \mid X = x]}{\mathbb{P}[Y = 1]}$$

$$= \frac{\overbrace{\text{prior} \times \text{likelihood}}^{\text{unnormalized posterior}}}{\text{normalizing constant}}$$

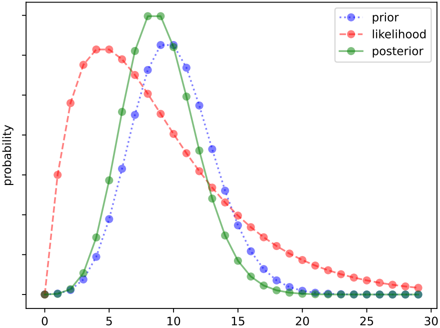
$$= \text{posterior}$$

Probabilistic program

$X \sim \text{Poisson}(10)$

$Y \sim \text{Binomial}(X, 0.2)$

observe $Y = 1$



Tools for Exact Inference

$$X \sim \text{Poisson}(10)$$
$$Y \sim \text{Binomial}(X, 0.2)$$
$$\text{observe}\, Y = 1$$

Tools for Exact Inference

$$X \sim \text{Poisson}(10)$$
$$Y \sim \text{Binomial}(X, 0.2)$$
$$\text{observe}\, Y = 1$$

Dice [Holtzen et al. 2020]

✗ Only supports finite discrete distributions.

Tools for Exact Inference

$$
\begin{aligned}
X &\sim \text{Poisson}(10) \\
Y &\sim \text{Binomial}(X, 0.2) \\
\text{observe}\, Y &= 1
\end{aligned}
$$

Dice [Holtzen et al. 2020]

✗ Only supports finite discrete distributions.

SPPL [Saad et al. 2021]

✗ Parameters of distributions must have finite support.

Tools for Exact Inference

$$X \sim \text{Poisson}(10)$$
$$Y \sim \text{Binomial}(X, 0.2)$$
$$\text{observe } Y = 1$$

Dice [Holtzen et al. 2020]

✗ Only supports finite discrete distributions.

SPPL [Saad et al. 2021]

✗ Parameters of distributions must have finite support.

PSI [Gehr et al. 2016]

✗ Outputs a symbolic expression involving infinite sums.

Probability Mass Functions

$X \sim \mathcal{D}$ (supported on $\mathbb{N}$)

$$\begin{aligned} \mathsf{pmf}_X : \mathbb{N} &\to [0,1] \\ n &\mapsto \mathbb{P}[X = n] \end{aligned}$$

Probability Mass Functions

$X \sim \mathcal{D}$ (supported on $\mathbb{N}$)

$$\begin{aligned} \mathsf{pmf}_X : \mathbb{N} &\to [0,1] \\ n &\mapsto \mathbb{P}[X = n] \end{aligned}$$

$(X, Y, Z) \sim \mathcal{D}$ (supported on $\mathbb{N}^3$)

$$\begin{aligned} \mathsf{pmf}_{X,Y,Z} : \mathbb{N}^3 &\to [0,1] \\ x, y, z &\mapsto \mathbb{P}[X = x, Y = y, Z = z] \end{aligned}$$

Infinite Support

$$X \sim \text{Poisson}(10)$$
$$Y \sim \text{Binomial}(X, 0.2)$$

Infinite Support

$$X \sim \text{Poisson}(10)$$
$$Y \sim \text{Binomial}(X, 0.2)$$

$$\mathbb{P}[Y = 9] = \sum_{x=0}^{\infty} \mathbb{P}[Y = 9, X = x] = \sum_{x=0}^{\infty} \mathbb{P}[X = x]\mathbb{P}[Y = 9 \mid X = x]$$
$$= \sum_{x=0}^{\infty} \text{pmf}_{\text{Poisson}(10)}(x)\text{pmf}_{\text{Binomial}(x,0.2)}(9)$$

Infinite Support

$$X \sim \text{Poisson}(10)$$
$$Y \sim \text{Binomial}(X, 0.2)$$

$$\mathbb{P}[Y = 9] = \sum_{x=0}^{\infty} \mathbb{P}[Y = 9, X = x] = \sum_{x=0}^{\infty} \mathbb{P}[X = x]\mathbb{P}[Y = 9 \mid X = x]$$
$$= \sum_{x=0}^{\infty} \text{pmf}_{\text{Poisson}(10)}(x)\text{pmf}_{\text{Binomial}(x,0.2)}(9)$$

Not computable exactly using probability *mass* functions!

Probability Generating Functions

$$X \sim \mathcal{D} \text{ (supported on } \mathbb{N})$$

Generating function (aka *factorial moment generating function*):

$$\begin{aligned}\mathsf{pgf}_X &: [-1, 1] \to \mathbb{R} \\ \mathsf{pgf}_X(t) &= \mathbb{E}[t^X] \quad \text{(discrete \& continuous)} \\ &= p(0) + p(1)t + p(2)t^2 + p(3)t^3 + \ldots \quad \text{(only discrete)} \\ &\quad \text{where } p(n) = \mathbb{P}[X = n]\end{aligned}$$

Probability Generating Functions

$$X \sim \mathcal{D} \text{ (supported on } \mathbb{N})$$

Generating function (aka *factorial moment generating function*):

$$\begin{aligned}
\mathsf{pgf}_X &: [-1,1] \to \mathbb{R} \\
\mathsf{pgf}_X(t) &= \mathbb{E}[t^X] \quad \text{(discrete \& continuous)} \\
&= p(0) + p(1)t + p(2)t^2 + p(3)t^3 + \dots \quad \text{(only discrete)} \\
&\quad \text{where } p(n) = \mathbb{P}[X = n]
\end{aligned}$$

Closed forms for most common distributions:

Binomial(n, p)	$(pt + 1 - p)^n$
NegBinomial(r, p)	$\left(\frac{1-p}{1-pt}\right)^r$
Geometric(p)	$\frac{p}{1-(1-p)t}$
Poisson(λ)	$e^{\lambda(t-1)}$

Several variables

$$(X, Y, Z) \sim \mathcal{D} \text{ (supported on } \mathbb{N}^3\text{)}$$

Generating function:

$$\begin{aligned} \mathsf{pgf}_{X,Y,Z} &: [-1,1]^3 \to \mathbb{R} \\ \mathsf{pgf}_{X,Y,Z}(x,y,z) &= \mathbb{E}[x^X y^Y z^Z] \\ &= \sum_{a,b,c \in \mathbb{N}} p(a,b,c) x^a y^b z^c \end{aligned}$$

where $p(a,b,c) = \mathbb{P}[X = a, Y = b, Z = c]$

Getting back the probability mass

Suppose X has generating function $g(t) = \sum_{n \in \mathbb{N}} p(n) t^n$.

Getting back the probability mass

Suppose X has generating function $g(t) = \sum_{n \in \mathbb{N}} p(n) t^n$.

Then $p(n)$ are the Taylor coefficients at $t = 0$, so $p(n) = \frac{g^{(n)}(0)}{n!}$.

Computing the (factorial) moments

Suppose X has generating function $g(t) = \mathbb{E}[t^X]$.

Computing the (factorial) moments

Suppose X has generating function $g(t) = \mathbb{E}[t^X]$.

Then $g'(1) = \frac{\mathsf{d}}{\mathsf{d}t}\mathbb{E}[t^X]\Big|_{t=1} = \mathbb{E}[X\,t^X]\Big|_{t=1} = \mathbb{E}[X]$.

Computing the (factorial) moments

Suppose X has generating function $g(t) = \mathbb{E}[t^X]$.

Then $g'(1) = \frac{\mathsf{d}}{\mathsf{d}t}\mathbb{E}[t^X]\Big|_{t=1} = \mathbb{E}[X\,t^X]\Big|_{t=1} = \mathbb{E}[X]$.

More generally, the factorial moment of order n is:

$$\mathbb{E}[X(X-1)\dots(X-n+1)] = \mathbb{E}\left[\frac{\mathsf{d}^n}{\mathsf{d}t^n}t^X\right]\Bigg|_{t=1} = g^{(n)}(1)$$

Computing the (factorial) moments

Suppose X has generating function $g(t) = \mathbb{E}[t^X]$.

Then $g'(1) = \frac{\mathsf{d}}{\mathsf{d}t}\mathbb{E}[t^X]\Big|_{t=1} = \mathbb{E}[X\,t^X]\Big|_{t=1} = \mathbb{E}[X]$.

More generally, the factorial moment of order n is:

$$\mathbb{E}[X(X-1)\dots(X-n+1)] = \mathbb{E}\left[\frac{\mathsf{d}^n}{\mathsf{d}t^n}t^X\right]\Bigg|_{t=1} = g^{(n)}(1)$$

The variance can be found as:

$$\begin{aligned}\mathbb{V}[X] &= \mathbb{E}[X^2] - \mathbb{E}[X]^2 \\ &= \mathbb{E}[X] + \mathbb{E}[X(X-1)] - \mathbb{E}[X]^2 \\ &= g'(1) + g''(1) - (g'(1))^2\end{aligned}$$

Summary: Generating Functions

Suppose X has generating function $g(t) = \mathbb{E}[t^X]$.

$$\mathbb{P}[X = n] = \frac{g^{(n)}(0)}{n!}$$
$$\mathbb{E}[X(X-1)\dots(X-n+1)] = g^{(n)}(1)$$

Summary: Generating Functions

Suppose X has generating function $g(t) = \mathbb{E}[t^X]$.

$$\mathbb{P}[X = n] = \frac{g^{(n)}(0)}{n!}$$

$$\mathbb{E}[X(X-1)\dots(X-n+1)] = g^{(n)}(1)$$

Take-aways:

- Generating functions are a finite representation of distributions (even with infinite support)
- Can compute mass and moments mechanically
- No computer algebra necessary
- Only need automatic differentiation

Related work using generating functions

Bayesian inference for graphical models:

- Winner et al., NeurIPS 2016: specific graphical model with a closed form for the generating function
- Winner et al., ICML 2017: slightly generalized graphical model, uses autodiff

Related work using generating functions

Bayesian inference for graphical models:

- Winner et al., NeurIPS 2016: specific graphical model with a closed form for the generating function
- Winner et al., ICML 2017: slightly generalized graphical model, uses autodiff

Randomized programs (no conditioning):

- Klinkenberg et al., LOPSTR 2020: analyze discrete randomized programs (no conditioning)
- Chen et al., CAV 2022: check equivalence of a restricted class of discrete randomized programs

SGCL: Statistical Guarded Command Lang.

Fixed set of variables $\mathcal{V} = \{X_1, \ldots, X_n\}$, taking values in $\mathbb{N}$.

SGCL: Statistical Guarded Command Lang.

Fixed set of variables $\mathcal{V} = \{X_1, \ldots, X_n\}$, taking values in $\mathbb{N}$.

Sampling	$X_i \sim \mathcal{D}$	
Affine transform	$X_i := aX_j + bX_k + c$	where $a, b, c \in \mathbb{N}$
Branching	if $X_i = c \, \{P_1\}$ else $\{P_2\}$	
Conditioning	observe $X_i = c$	
Nested inference	normalize $\{P\}$	

SGCL: Statistical Guarded Command Lang.

Fixed set of variables $\mathcal{V} = \{X_1, \ldots, X_n\}$, taking values in $\mathbb{N}$.

Sampling	$X_i \sim \mathcal{D}$	
Affine transform	$X_i := aX_j + bX_k + c$	where $a, b, c \in \mathbb{N}$
Branching	if $X_i = c\,\{P_1\}$ else $\{P_2\}$	
Conditioning	observe $X_i = c$	
Nested inference	normalize $\{P\}$	

Distributions

$$\mathcal{D} \in \{\text{Bernoulli}(p), \text{Binomial}(X_k, p), \text{NegBinomial}(X_k, p), \text{Geometric}(p), \text{Poisson}(\lambda X_k), \text{Uniform}\{X_k..X_k + a\} \mid p \in \mathbb{R}, \lambda \in (0, \infty), a \in \mathbb{N}\}.$$

Ordinary Transformer Semantics

Distributions on program states are represented by their probability mass function, i.e. $\sigma : \mathbb{N}^n \to [0, 1]$ where $\mathcal{V} = \{X_1, \ldots, X_n\}$ and $\sum_{x \in \mathbb{N}^n} \sigma(x) \leq 1$.

Ordinary Transformer Semantics

Distributions on program states are represented by their probability mass function, i.e. $\sigma : \mathbb{N}^n \to [0, 1]$ where $\mathcal{V} = \{X_1, \ldots, X_n\}$ and $\sum_{x \in \mathbb{N}^n} \sigma(x) \leq 1$.

Transformer semantics: $\sigma' = \langle P \rangle(\sigma)$ is the subprobability distribution of states after running P.

Ordinary Transformer Semantics

Distributions on program states are represented by their probability mass function, i.e. $\sigma : \mathbb{N}^n \to [0, 1]$ where $\mathcal{V} = \{X_1, \ldots, X_n\}$ and $\sum_{x \in \mathbb{N}^n} \sigma(x) \leq 1$.

Transformer semantics: $\sigma' = \langle P \rangle(\sigma)$ is the subprobability distribution of states after running P.

For simplicity, we only use two variables: X, Y, i.e. $\sigma(x, y) = \mathbb{P}[X = x, Y = y]$.

Ordinary Transformer Semantics

Distributions on program states are represented by their probability mass function, i.e. $\sigma : \mathbb{N}^n \to [0, 1]$ where $\mathcal{V} = \{X_1, \ldots, X_n\}$ and $\sum_{x \in \mathbb{N}^n} \sigma(x) \leq 1$.

Transformer semantics: $\sigma' = \langle P \rangle(\sigma)$ is the subprobability distribution of states after running P.

For simplicity, we only use two variables: X, Y, i.e. $\sigma(x, y) = \mathbb{P}[X = x, Y = y]$.

Affine transform:
$\langle X := aX + bY + c \rangle(\sigma)(x, y) = \sum_{x' : ax' + by + c = x} \sigma(x', y)$.

Ordinary Transformer Semantics

Distributions on program states are represented by their probability mass function, i.e. $\sigma : \mathbb{N}^n \to [0,1]$ where $\mathcal{V} = \{X_1, \ldots, X_n\}$ and $\sum_{x \in \mathbb{N}^n} \sigma(x) \leq 1$.

Transformer semantics: $\sigma' = \langle P \rangle(\sigma)$ is the subprobability distribution of states after running P.

For simplicity, we only use two variables: X, Y, i.e. $\sigma(x, y) = \mathbb{P}[X = x, Y = y]$.

Affine transform:
$\langle X := aX + bY + c \rangle(\sigma)(x, y) = \sum_{x' : ax' + by + c = x} \sigma(x', y)$.

Sampling: $\langle X \sim \mathcal{D} \rangle(\sigma)(x, y) = \sum_{a \in \mathbb{N}} \sigma(a, y) \cdot \text{pmf}_{\mathcal{D}}(x)$.

Ordinary Transformer Semantics

Distributions on program states are represented by their probability mass function, i.e. $\sigma : \mathbb{N}^n \to [0,1]$ where $\mathcal{V} = \{X_1, \dots, X_n\}$ and $\sum_{x \in \mathbb{N}^n} \sigma(x) \leq 1$.

Transformer semantics: $\sigma' = \langle P \rangle(\sigma)$ is the subprobability distribution of states after running P.

For simplicity, we only use two variables: X, Y, i.e. $\sigma(x, y) = \mathbb{P}[X = x, Y = y]$.

Affine transform:
$\langle X := aX + bY + c \rangle(\sigma)(x, y) = \sum_{x' : ax' + by + c = x} \sigma(x', y)$.

Sampling: $\langle X \sim \mathcal{D} \rangle(\sigma)(x, y) = \sum_{a \in \mathbb{N}} \sigma(a, y) \cdot \mathrm{pmf}_{\mathcal{D}}(x)$.

Conditioning: $\langle \text{observe}\, X = c \rangle(\sigma)(x, y) = \sigma(x, y) \cdot [x = c]$.

Ordinary Transformer Semantics

Distributions on program states are represented by their probability mass function, i.e. $\sigma : \mathbb{N}^n \to [0,1]$ where $\mathcal{V} = \{X_1, \ldots, X_n\}$ and $\sum_{x \in \mathbb{N}^n} \sigma(x) \leq 1$.

Transformer semantics: $\sigma' = \langle P \rangle(\sigma)$ is the subprobability distribution of states after running P.

For simplicity, we only use two variables: X, Y, i.e. $\sigma(x,y) = \mathbb{P}[X = x, Y = y]$.

Affine transform:
$\langle X := aX + bY + c \rangle(\sigma)(x,y) = \sum_{x' : ax' + by + c = x} \sigma(x', y)$.

Sampling: $\langle X \sim \mathcal{D} \rangle(\sigma)(x,y) = \sum_{a \in \mathbb{N}} \sigma(a,y) \cdot \text{pmf}_{\mathcal{D}}(x)$.

Conditioning: $\langle \text{observe}\, X = c \rangle(\sigma)(x,y) = \sigma(x,y) \cdot [x = c]$.

Normalize: $\langle \text{normalize}\, \{P\} \rangle(\sigma)(x,y) = \frac{\sum_{x,y \in \mathbb{N}} \sigma(x,y)}{\sum_{x,y \in \mathbb{N}} \langle P \rangle(\sigma)(x,y)} \langle P \rangle(\sigma)(x,y)$.

Generating Function Semantics [Klinkenberg et al., 2020]

Subprobability distributions of states are represented as a generating function, i.e. $G : [-1, 1]^n \to \mathbb{R}$ where $\mathcal{V} = \{X_1, \ldots, X_n\}$.

Generating Function Semantics [Klinkenberg et al., 2020]

Subprobability distributions of states are represented as a generating function, i.e. $G : [-1, 1]^n \to \mathbb{R}$ where $\mathcal{V} = \{X_1, \ldots, X_n\}$.

Generating function semantics: $G' = [\![P]\!](G)$ is the generating function of the state distribution after running P.

Generating Function Semantics [Klinkenberg et al., 2020]

Subprobability distributions of states are represented as a generating function, i.e. $G : [-1, 1]^n \to \mathbb{R}$ where $\mathcal{V} = \{X_1, \ldots, X_n\}$.

Generating function semantics: $G' = [\![P]\!](G)$ is the generating function of the state distribution after running P.

For simplicity, we only use two variables X, and Y, i.e. $G(x, y) = \mathbb{E}[x^X \cdot y^Y]$.

Generating Function Semantics [Klinkenberg et al., 2020]

Subprobability distributions of states are represented as a generating function, i.e. $G : [-1, 1]^n \to \mathbb{R}$ where $\mathcal{V} = \{X_1, \ldots, X_n\}$.

Generating function semantics: $G' = [\![P]\!](G)$ is the generating function of the state distribution after running P.

For simplicity, we only use two variables X, and Y, i.e. $G(x, y) = \mathbb{E}[x^X \cdot y^Y]$.

Marginalizing: $[\![X = 0]\!](G)(x, y) = \mathbb{E}[x^0 y^Y] = \mathbb{E}[1^X y^Y] = G(1, y)$

Generating Function Semantics [Klinkenberg et al., 2020]

Subprobability distributions of states are represented as a generating function, i.e. $G : [-1, 1]^n \to \mathbb{R}$ where $\mathcal{V} = \{X_1, \ldots, X_n\}$.

Generating function semantics: $G' = [\![P]\!](G)$ is the generating function of the state distribution after running P.

For simplicity, we only use two variables X, and Y, i.e. $G(x, y) = \mathbb{E}[x^X \cdot y^Y]$.

Marginalizing: $[\![X = 0]\!](G)(x, y) = \mathbb{E}[x^0 y^Y] = \mathbb{E}[1^X y^Y] = G(1, y)$

Affine transform: $[\![X = aX + bY + c]\!](G)(x, y) = \mathbb{E}[x^{aX+bY+c} y^Y] = \mathbb{E}[(x^a)^X (x^b y)^Y] \cdot x^c = G(x^a, x^b y) \cdot x^c$.

Semantics of Sampling

For a distribution $\mathcal{D}$ with constant parameters [Klinkenberg et al., 2020]:

$$
\begin{aligned}
[\![X \sim \mathcal{D}]\!](G)(x, y) &= \mathbb{E}_{X\sim D}[x^X y^Y] \\
&= \mathbb{E}[y^Y]\mathbb{E}_{X\sim D}[x^X] \\
&= G(1, y)\mathsf{pgf}_{\mathcal{D}}(x)
\end{aligned}
$$

Semantics of Sampling

For a distribution $\mathcal{D}$ with constant parameters [Klinkenberg et al., 2020]:

$$\begin{aligned} [\![X \sim \mathcal{D}]\!](G)(x,y) &= \mathbb{E}_{X\sim D}[x^X y^Y] \\ &= \mathbb{E}[y^Y]\mathbb{E}_{X\sim D}[x^X] \\ &= G(1,y)\mathsf{pgf}_{\mathcal{D}}(x) \end{aligned}$$

For distributions with random parameters, we find:

$$[\![X \sim \mathsf{Binomial}(Y,p)]\!](G)(x,y) = G(1, y(px+1-p))$$ [Winner et al., 2016]

$$[\![X \sim \mathsf{NegBinomial}(Y,p)]\!](G)(x,y) = G(1, y\frac{1-p}{1-px})$$ new

$$[\![X \sim \mathsf{Poisson}(\lambda Y)]\!](G)(x,y) = G(1, ye^{\lambda(x-1)})$$ new

Semantics of Conditioning (new)

$$
\begin{aligned}
&[\![\textsf{observe}\, X = c]\!](G)(x, y) \\
&= \mathbb{E}[x^X y^Y [X = c]] \\
&= x^c \mathbb{E}[x^{X-c} y^Y [X = c]] \\
&= \frac{x^c}{c!} \mathbb{E}\left[X(X-1)\dots(X-c+1) x^{X-c}\big|_{x=0} \cdot y^Y \cdot [X = c]\right] \\
&= \frac{x^c}{c!} \left(\frac{\partial^c}{\partial x^c} \mathbb{E}[x^X y^Y]\right)\Bigg|_{x=0} \\
&= \frac{x^c}{c!} \frac{\partial^c}{\partial x^c} G(0, y)
\end{aligned}
$$

Semantics of Conditioning (new)

$$
\begin{aligned}
&[\![\textsf{observe}\, X = c]\!](G)(x,y) \\
&= \mathbb{E}[x^X y^Y [X = c]] \\
&= x^c \mathbb{E}[x^{X-c} y^Y [X = c]] \\
&= \frac{x^c}{c!} \mathbb{E}\left[X(X-1)\ldots(X-c+1) x^{X-c}\big|_{x=0} \cdot y^Y \cdot [X = c]\right] \\
&= \frac{x^c}{c!} \left(\frac{\partial^c}{\partial x^c} \mathbb{E}[x^X y^Y]\right)\Bigg|_{x=0} \\
&= \frac{x^c}{c!} \frac{\partial^c}{\partial x^c} G(0, y)
\end{aligned}
$$

Observing a value c requires evaluating the c-th derivative of the generating function!

Semantics of Normalization [new]

Total probability mass:

$$\mathbb{E}[1] = \mathbb{E}[1^X 1^Y] = G(1, 1).$$

Semantics of Normalization [new]

Total probability mass:

$$\mathbb{E}[1] = \mathbb{E}[1^X 1^Y] = G(1,1).$$

So to normalize a subprogram P:

$$[\![\mathsf{normalize}\,\{P\}]\!](G)(x,y) = \frac{G(1,1)}{[\![P]\!](G)(1,1)}[\![P]\!](G).$$

Example

$$\text{normalize}\,\{X \sim \text{Poisson}(10); Y \sim \text{Binomial}(X, 0.2); \text{observe}\, Y = 1\}$$

Example

normalize $\{X \sim \text{Poisson}(10); Y \sim \text{Binomial}(X, 0.2); \text{observe}\ Y = 1\}$

- $A(x, y) = \mathbb{E}[x^0 y^0] = 1$.

Example

$$\text{normalize}\,\{X \sim \text{Poisson}(10); Y \sim \text{Binomial}(X, 0.2); \text{observe}\, Y = 1\}$$

- $A(x, y) = \mathbb{E}[x^0 y^0] = 1$.
- Sampling Poisson: $B(x, y) = A(1, y)e^{10(x-1)} = e^{10(x-1)}$.

Example

normalize $\{X \sim \text{Poisson}(10); Y \sim \text{Binomial}(X, 0.2); \text{observe } Y = 1\}$

- $A(x, y) = \mathbb{E}[x^0 y^0] = 1$.
- Sampling Poisson: $B(x, y) = A(1, y)e^{10(x-1)} = e^{10(x-1)}$.
- Sampling Binomial:
  $C(x, y) = B(x(0.2y + 0.8), 1) = \exp(10(x(0.2y + 0.8) - 1))$.

Example

normalize $\{X \sim \text{Poisson}(10); Y \sim \text{Binomial}(X, 0.2); \text{observe}\, Y = 1\}$

- $A(x, y) = \mathbb{E}[x^0 y^0] = 1$.
- Sampling Poisson: $B(x, y) = A(1, y)e^{10(x-1)} = e^{10(x-1)}$.
- Sampling Binomial:
  $C(x, y) = B(x(0.2y + 0.8), 1) = \exp(10(x(0.2y + 0.8) - 1))$.
- Observing $Y = 1$: $D(x, y) = \frac{1}{1!} y \frac{\partial}{\partial y} C(x, 0) = 2xye^{8x-10}$.

Example

normalize $\{X \sim \text{Poisson}(10); Y \sim \text{Binomial}(X, 0.2); \text{observe } Y = 1\}$

- $A(x, y) = \mathbb{E}[x^0 y^0] = 1$.
- Sampling Poisson: $B(x, y) = A(1, y)e^{10(x-1)} = e^{10(x-1)}$.
- Sampling Binomial:
  $C(x, y) = B(x(0.2y + 0.8), 1) = \exp(10(x(0.2y + 0.8) - 1))$.
- Observing $Y = 1$: $D(x, y) = \frac{1}{1!} y \frac{\partial}{\partial y} C(x, 0) = 2xye^{8x-10}$.
- Normalizing: $E(x, y) = \frac{A(1,1)}{D(1,1)} D(x, y) = \frac{D(x,y)}{D(1,1)} = xye^{8x-8}$.

Extracting information from the generating function

$$E(x, y) = xe^{8x-8}y$$

Extracting information from that generating function:

Extracting information from the generating function

$$E(x, y) = xe^{8x-8}y$$

Extracting information from that generating function:

- $\mathbb{P}[X = 10] = \frac{1}{10!}\frac{\partial^{10}}{\partial x^{10}}E(0, 1) = \frac{1048576}{2835}e^{-8}$

Extracting information from the generating function

$$E(x, y) = xe^{8x-8}y$$

Extracting information from that generating function:

- $\mathbb{P}[X = 10] = \frac{1}{10!}\frac{\partial^{10}}{\partial x^{10}}E(0, 1) = \frac{1048576}{2835}e^{-8}$
- $\mathbb{E}[X] = \frac{\partial}{\partial x}E(1, 1) = 9$

Extracting information from the generating function

$$E(x, y) = xe^{8x-8}y$$

Extracting information from that generating function:

- $\mathbb{P}[X = 10] = \frac{1}{10!}\frac{\partial^{10}}{\partial x^{10}}E(0, 1) = \frac{1048576}{2835}e^{-8}$
- $\mathbb{E}[X] = \frac{\partial}{\partial x}E(1, 1) = 9$
- The program

  $$X \sim \text{Poisson}(8); X := X + 1; Y = 1$$

  has the same GF $G(x, y) = xe^{8x-8}y$!

Example 2: Population modeling (HMM)

Modeling animal populations [Winner et al., NeurIPS 2016]:

$$
\begin{array}{l}
population := 0; \\
\\
arrivals \sim \textsf{Poisson}(\lambda); \\
survivors \sim \textsf{Binomial}(population, \delta); \\
population := arrivals + survivors; \\
observed \sim \textsf{Binomial}(population, \rho); \\
\textsf{observe}\, observed = \ldots; \\
\vdots
\end{array}
$$

Example 3: Bayesian change point analysis

From the PyMC3 tutorial:

- number of coal mining disasters d_t over the last 100 years
- reason to believe that the rate has changed
- model as Poisson distribution with two different rates.

Example 3: Bayesian change point analysis

From the PyMC3 tutorial:

- number of coal mining disasters d_t over the last 100 years
- reason to believe that the rate has changed
- model as Poisson distribution with two different rates.

$$switchpoint \sim \text{Uniform}(0, 100);$$
$$\lambda_1 \sim \text{Exponential}(1);$$
$$\lambda_2 \sim \text{Exponential}(1);$$

$$\text{for } t \in \{0, \ldots, 100\} \; \{$$
$$\quad \text{if } switchpoint \leq t \; \{obs \sim \text{Poisson}(\lambda_2)\} \text{ else } \{obs \sim \text{Poisson}(\lambda_1)\}$$
$$\quad \text{observe } obs = d_t$$
$$\}$$

Example 3: Bayesian change point analysis

From the PyMC3 tutorial:

- number of coal mining disasters d_t over the last 100 years
- reason to believe that the rate has changed
- model as Poisson distribution with two different rates.

$switchpoint \sim \text{Uniform}(0, 100);$

$\lambda_1 \sim \text{Geometric}(0.2);$

$\lambda_2 \sim \text{Geometric}(0.2);$

for $t \in \{0, \ldots, 100\}$ {

  if $switchpoint \leq t$ $\{obs \sim \text{Poisson}(\lambda_2)\}$ else $\{obs \sim \text{Poisson}(\lambda_1)\}$

  observe $obs = d_t$

}

Implementation – Lessons

I implemented the semantics in Rust.

Implementation – Lessons

I implemented the semantics in Rust.

- Computation of derivatives is the bottleneck.

Implementation – Lessons

I implemented the semantics in Rust.

- Computation of derivatives is the bottleneck.
- Existing autodiff frameworks focus on 1st & 2nd derivative, very slow for higher order

Implementation – Lessons

I implemented the semantics in Rust.

- Computation of derivatives is the bottleneck.
- Existing autodiff frameworks focus on 1st & 2nd derivative, very slow for higher order
- Manual implementation of higher-order derivatives is still slow

Implementation – Lessons

I implemented the semantics in Rust.

- Computation of derivatives is the bottleneck.
- Existing autodiff frameworks focus on 1st & 2nd derivative, very slow for higher order
- Manual implementation of higher-order derivatives is still slow
- Maximize sharing of subexpressions in the computation graph

Implementation – Lessons

I implemented the semantics in Rust.

- Computation of derivatives is the bottleneck.
- Existing autodiff frameworks focus on 1st & 2nd derivative, very slow for higher order
- Manual implementation of higher-order derivatives is still slow
- Maximize sharing of subexpressions in the computation graph
- Computing directly with Taylor expansions is more efficient than autodiff (100x speedup)

Implementation – Lessons

I implemented the semantics in Rust.

- Computation of derivatives is the bottleneck.
- Existing autodiff frameworks focus on 1st & 2nd derivative, very slow for higher order
- Manual implementation of higher-order derivatives is still slow
- Maximize sharing of subexpressions in the computation graph
- Computing directly with Taylor expansions is more efficient than autodiff (100x speedup)

Exponential blowup:

Implementation – Lessons

I implemented the semantics in Rust.

- Computation of derivatives is the bottleneck.
- Existing autodiff frameworks focus on 1st & 2nd derivative, very slow for higher order
- Manual implementation of higher-order derivatives is still slow
- Maximize sharing of subexpressions in the computation graph
- Computing directly with Taylor expansions is more efficient than autodiff (100x speedup)

Exponential blowup:

- Size of the generating function can grow exponentially with the constants in the program.

Implementation – Lessons

I implemented the semantics in Rust.

- Computation of derivatives is the bottleneck.
- Existing autodiff frameworks focus on 1st & 2nd derivative, very slow for higher order
- Manual implementation of higher-order derivatives is still slow
- Maximize sharing of subexpressions in the computation graph
- Computing directly with Taylor expansions is more efficient than autodiff (100x speedup)

Exponential blowup:

- Size of the generating function can grow exponentially with the constants in the program.
- Heavy use of conditionals can lead to path explosion (but not common in probabilistic models).

Demo of implementation

Limitations

Language features:

- only affine functions (e.g. no X^2)
- only comparisons between variables and constants (e.g. no $X = Y$)
- only discrete distributions

Limitations

Language features:

- only affine functions (e.g. no X^2)
- only comparisons between variables and constants (e.g. no $X = Y$)
- only discrete distributions

Performance:

- worst-case exponential in the constants appearing in the program
- But works well for some models.
- Path explosion with many if statements.

Future Work

- Extensions to loops and recursion (lower bounds should be easy)
- Extension to a higher-order functional language

Generating Functions – Summary

- GFs are a finite closed-form representation for infinite distributions.
- Probability mass and moments can be extracted mechanically from GFs.
- No computer algebra needed.
- Needed: autodiff/Taylor expansion.
- Supports many language features: affine transformations, discrete distributions (even with random parameters), conditionals, conditioning, nested inference.
- Practical examples: population modeling & Bayesian change point analysis.
- Implementation promising for practical probabilistic programs.
- Limitations: exponential blowup.

Backup slides

Why only discrete distributions?

- $\mathbb{P}[X = n]$ is uninteresting for continuous distribution (always zero)
- reconstructing the density function from the factorial moment generating function requires solving integrals
- the factorial moment generating function does not exist for all distributions (e.g. Cauchy distribution)
- Observations from continuous distributions cannot be expressed as conditioning on an event (instead it's multiplication by the probability density function).