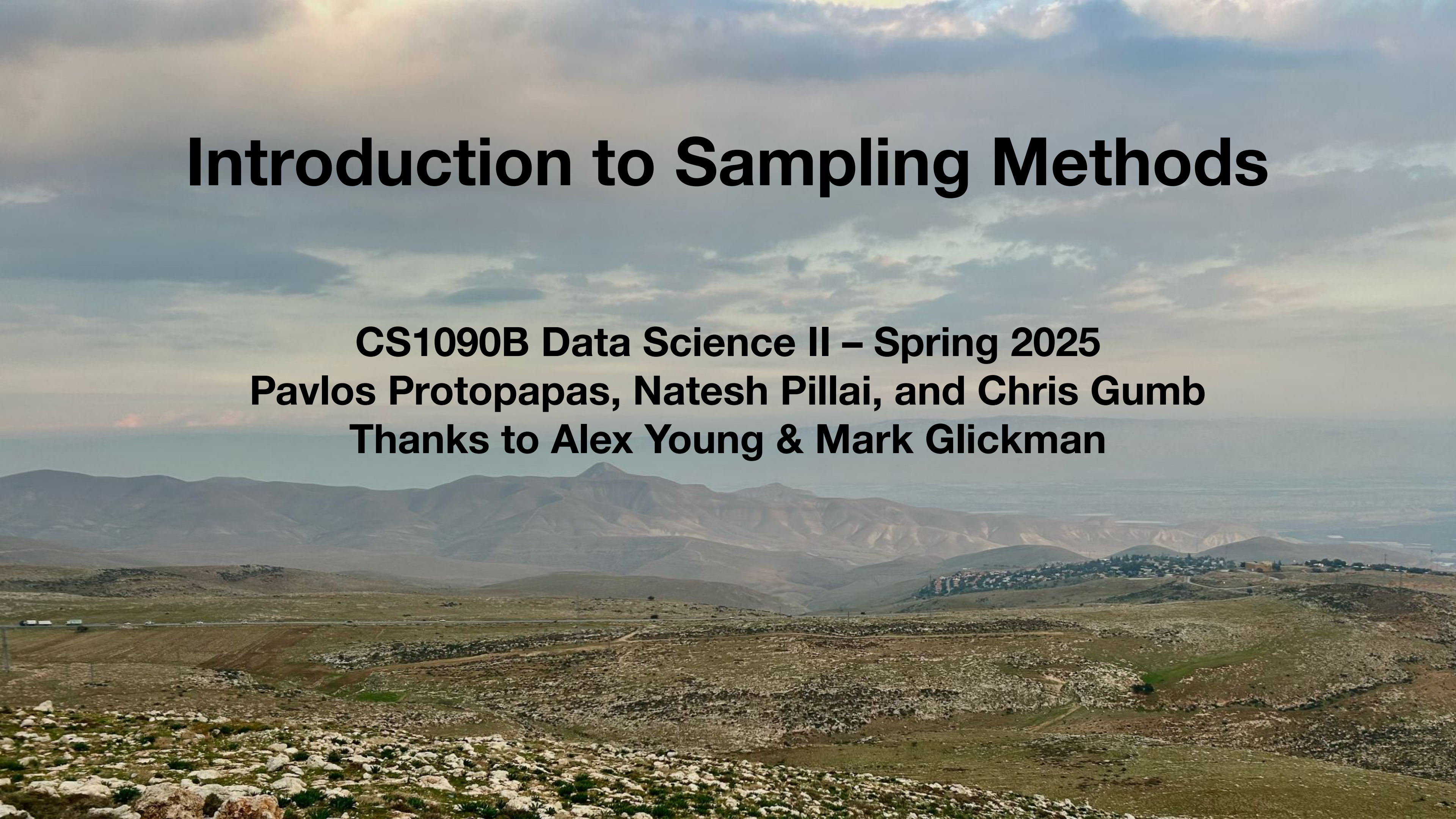


Introduction to Sampling Methods

CS1090B Data Science II – Spring 2025

Pavlos Protopapas, Natesh Pillai, and Chris Gumb

Thanks to Alex Young & Mark Glickman

Lemonade Example (revisited…)

- Last time, we used Bayesian modeling to help Lily estimate a distribution for the average sales per day (the posterior distribution). We assumed a Gamma prior and a Poisson likelihood, to get the posterior distribution:

$$\lambda \mid Y = y \ \sim \ \Gamma(\alpha + 82,\ \beta + 5).$$

- Then, we solved for the posterior predictive, or the distribution of sales Lily can expect tomorrow, or $p(Y_6 \mid \mathcal{Y})$.
- We recognized the posterior predictive as belonging to the Negative Binomial distribution, namely:

$$\boldsymbol{Y_6 \mid \mathcal{Y} \sim \mathrm{NegBin}\left(\alpha + 82, \frac{\beta + 5}{\beta + 6}\right)}$$

Lemonade Example (revisited...)

- In cases where we do not know much about the posterior predictive distribution, we can gather information about it in two ways:
  - If we recognize it belongs to the Negative Binomial distribution, we can sample from it, and compute whatever we need from this sample (mean, variance, etc.).
  - **If we don't recognize the posterior predictive distribution** (and so can't sample from it directly), we can:
    - Generate sample of λ from $p(\lambda|\mathcal{Y})$ (the posterior distribution).
    - Given this value of λ, generate a sample of Y_6 from $p(Y_6|\lambda) = \mathrm{Pois}(\lambda)$.

Lemonade Example (revisited…)

- But what if we **can't sample from the posterior predictive, but also can't sample from the posterior**? What then?
- This situation arises whenever we don't have conjugacy; our prior and likelihood don't imply any given distribution for the posterior.
- In this case, although we can't sample from the posterior directly, there still is a way to sample from it indirectly!

- **Goal: Generate a sample from a distribution we cannot sample from directly.**
    - To do this, we'll use Rejection Sampling and look at a simpler example!

Rejection Sampling

1. **Define a Target Distribution** $f(x)$: Specify the probability distribution you want to sample from.
2. **Choose a Proposal Distribution** $g(x)$: Select a simpler distribution from which you can easily sample, ensuring $f(x) \leq Mg(x)$ for all x, where $M > 1$ is a scaling factor.
3. **Draw a Candidate Sample:** Generate a random sample x from the proposal distribution $g(x)$.
4. **Generate a Uniform Random Number:** Draw $u \sim Uniform(0,1)$.
5. **Acceptance Condition:**

   Compute the acceptance ratio $r = f(x)/Mg(x)$.

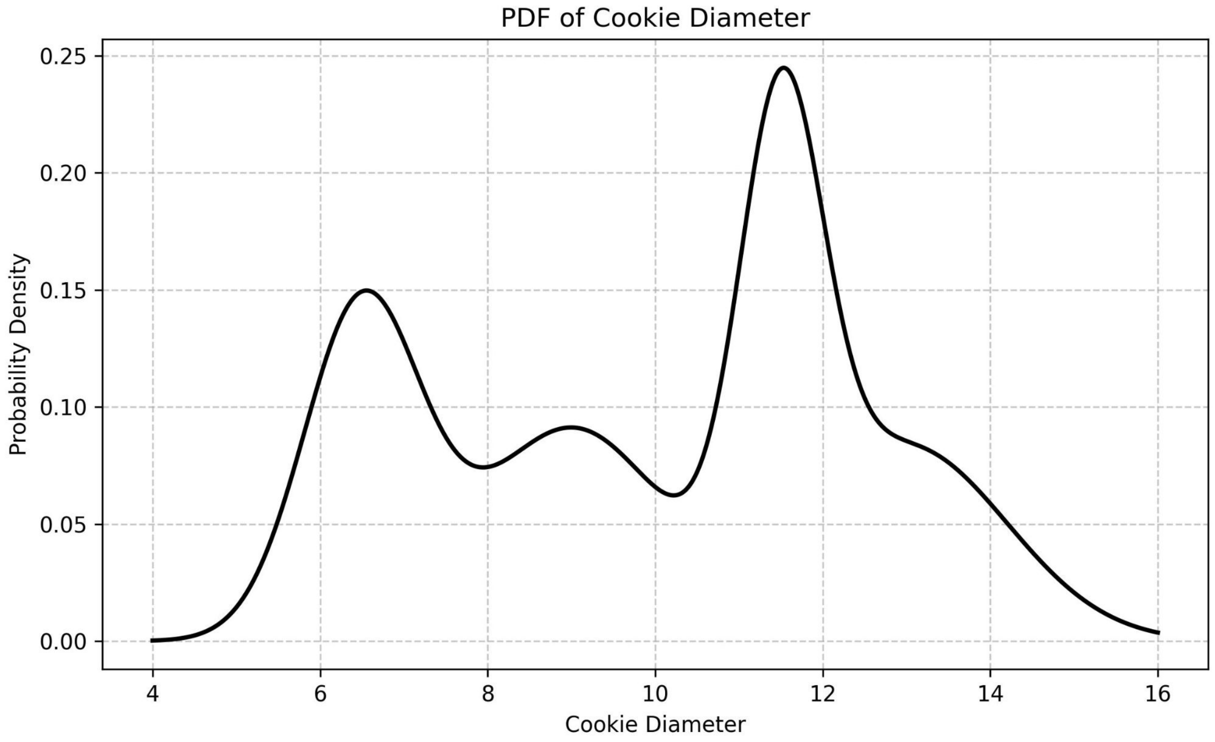
   Accept the candidate sample x if $u \leq r$; otherwise, reject it.
6. **Repeat Until Desired Sample Size:**

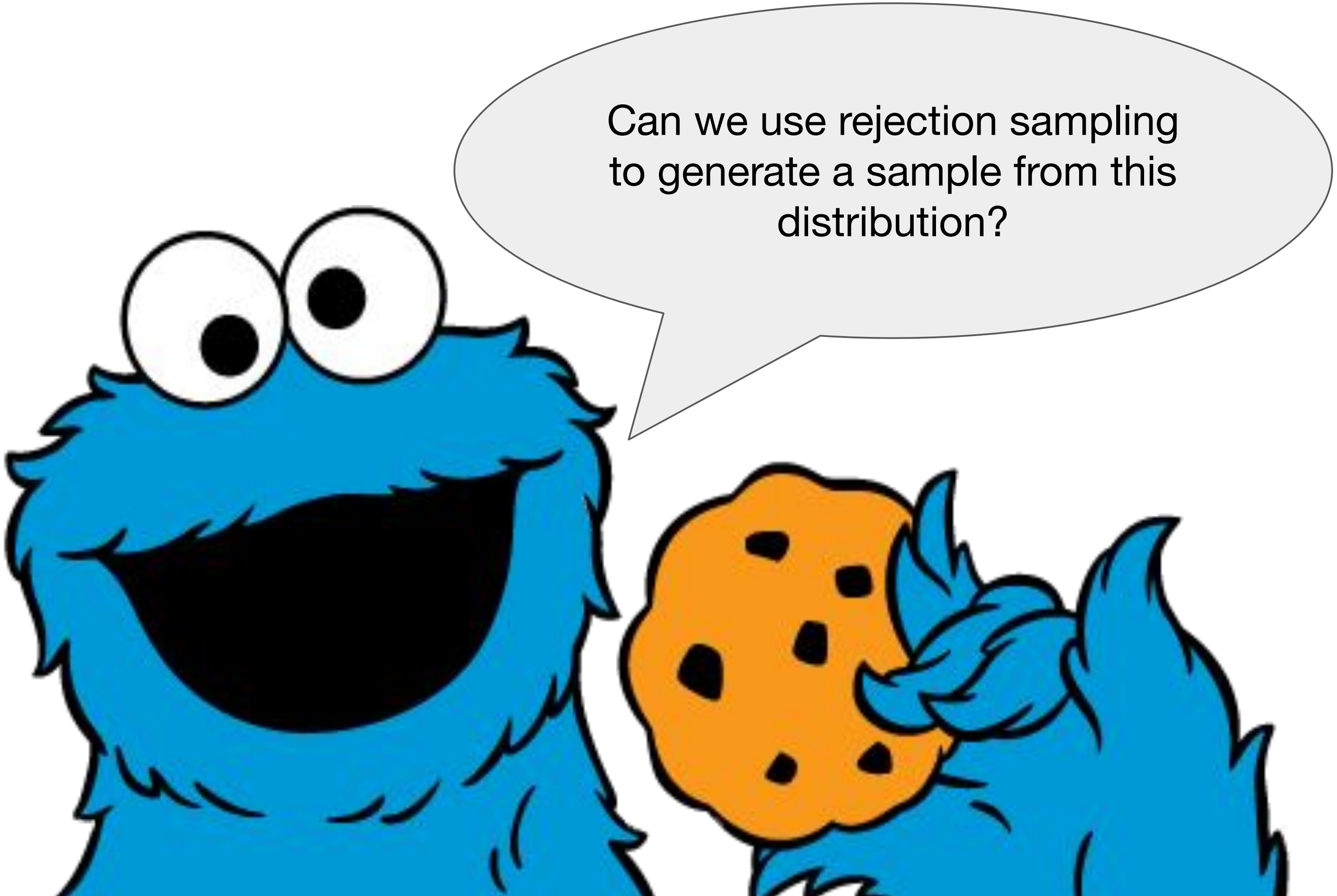
   Repeat steps 3–5 until you collect enough accepted samples.

Cookie Monster Needs Our Help!

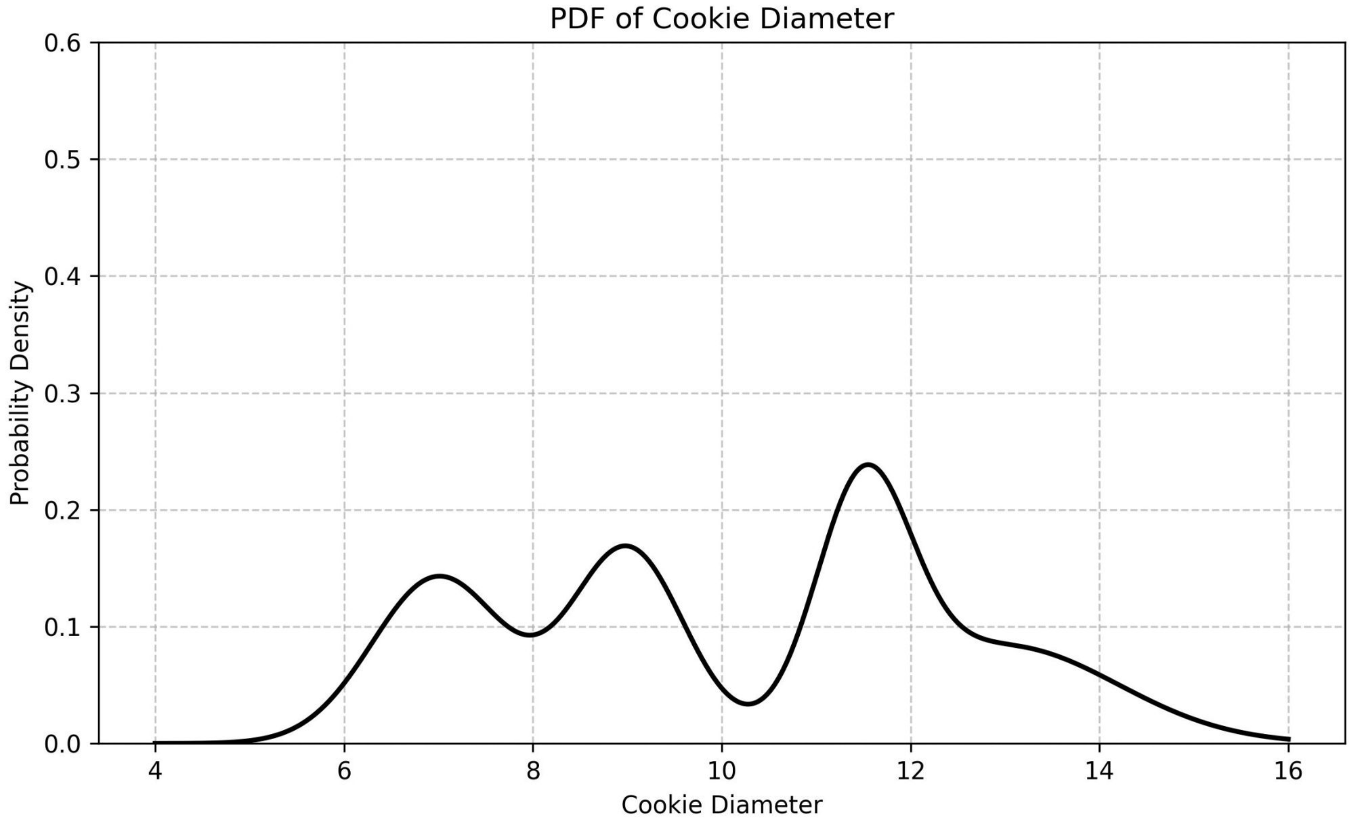
- The Cookie Monster stumbled upon a famous bakery that **sells four varieties of cookies: chocolate chip, oatmeal raisin, peanut butter, and sugar cookies.**
- Each variety has a diameter that varies slightly due to the baking process and the ingredients.
- The Cookie Monster is interested in sampling from the PDF of cookie diameters, but how?

Only Information Available: Cookie Diameter PDF

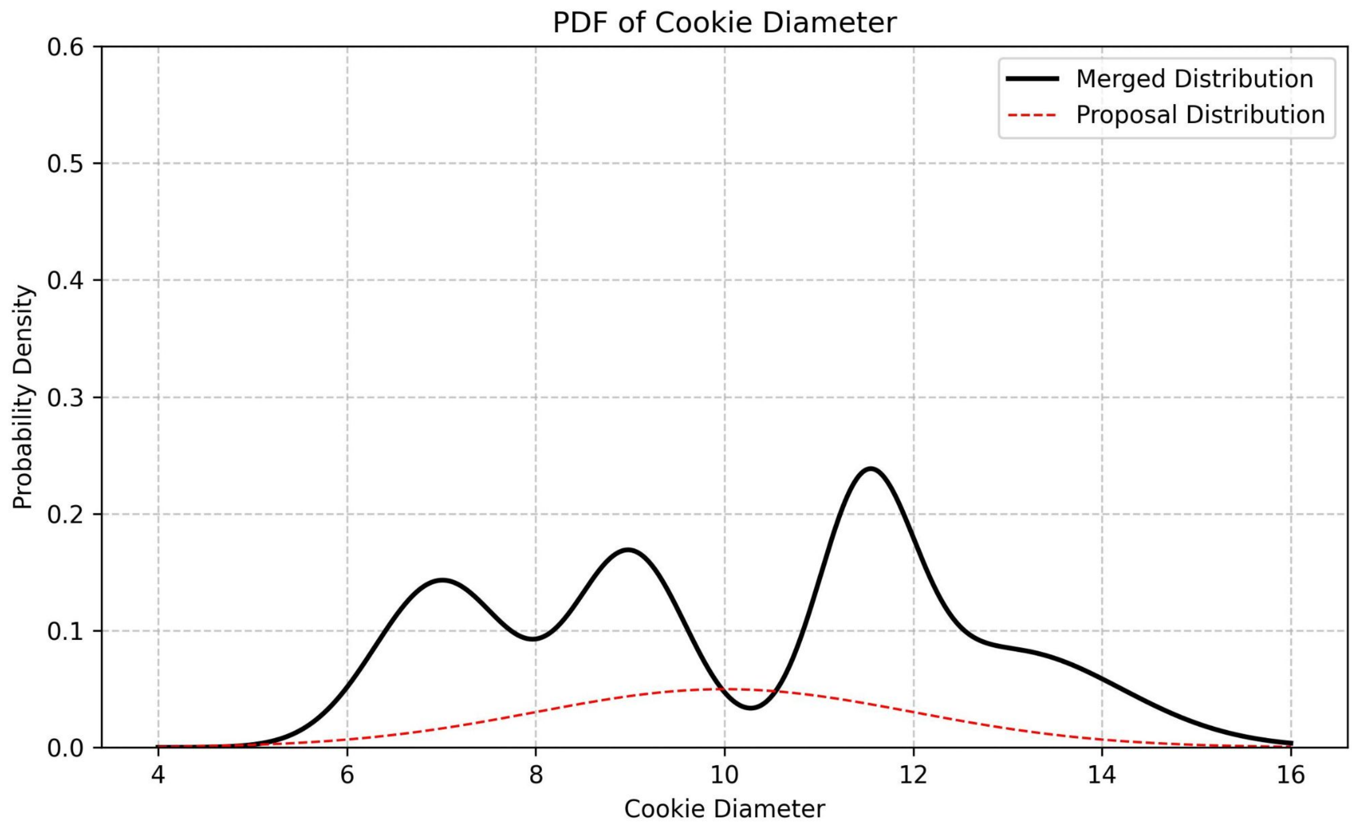



Can we use rejection sampling to generate a sample from this distribution?

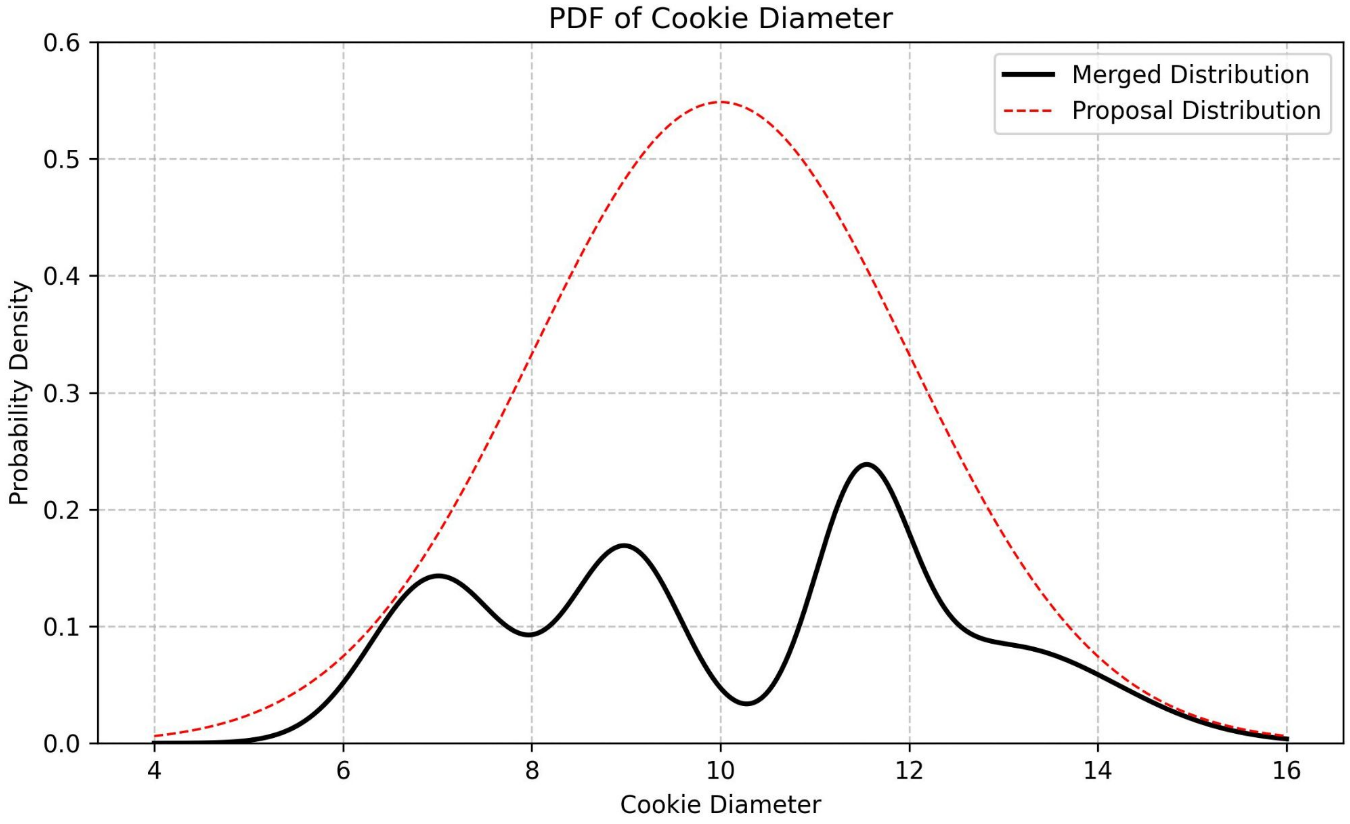
Step 1: Target Distribution



Step 2A: Proposal Distribution $\mathcal{N}(10, 2)$



Step 2B: Set Proposal Distribution $M = 11$



Rejection Sampling

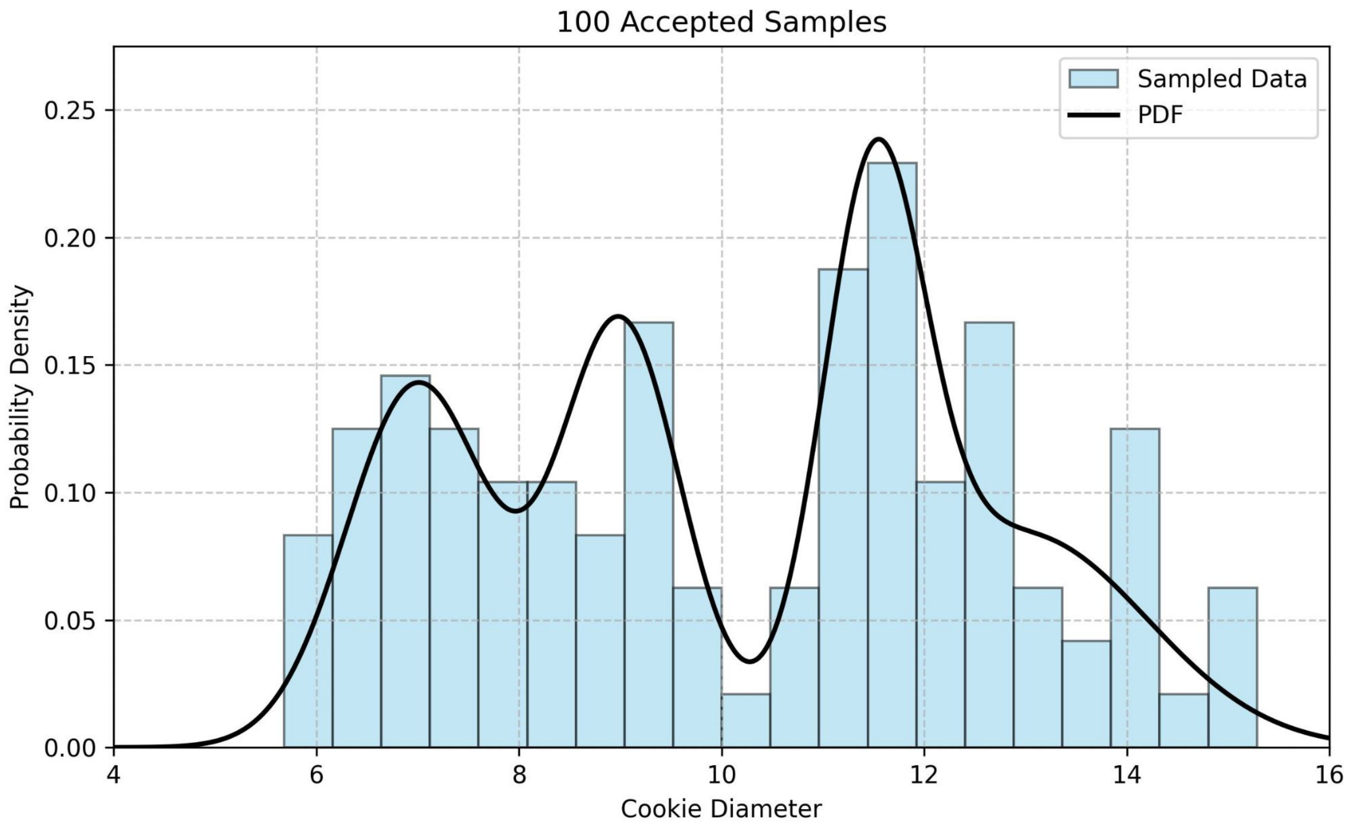
Now, we repeat the rest of the steps until we get a sample of some size:

1. **Draw a Candidate Sample:** Generate a random sample x from the proposal distribution $g(x)$.
2. **Generate a Uniform Random Number:** Draw $u \sim Uniform(0,1)$.
3. **Acceptance Condition:**

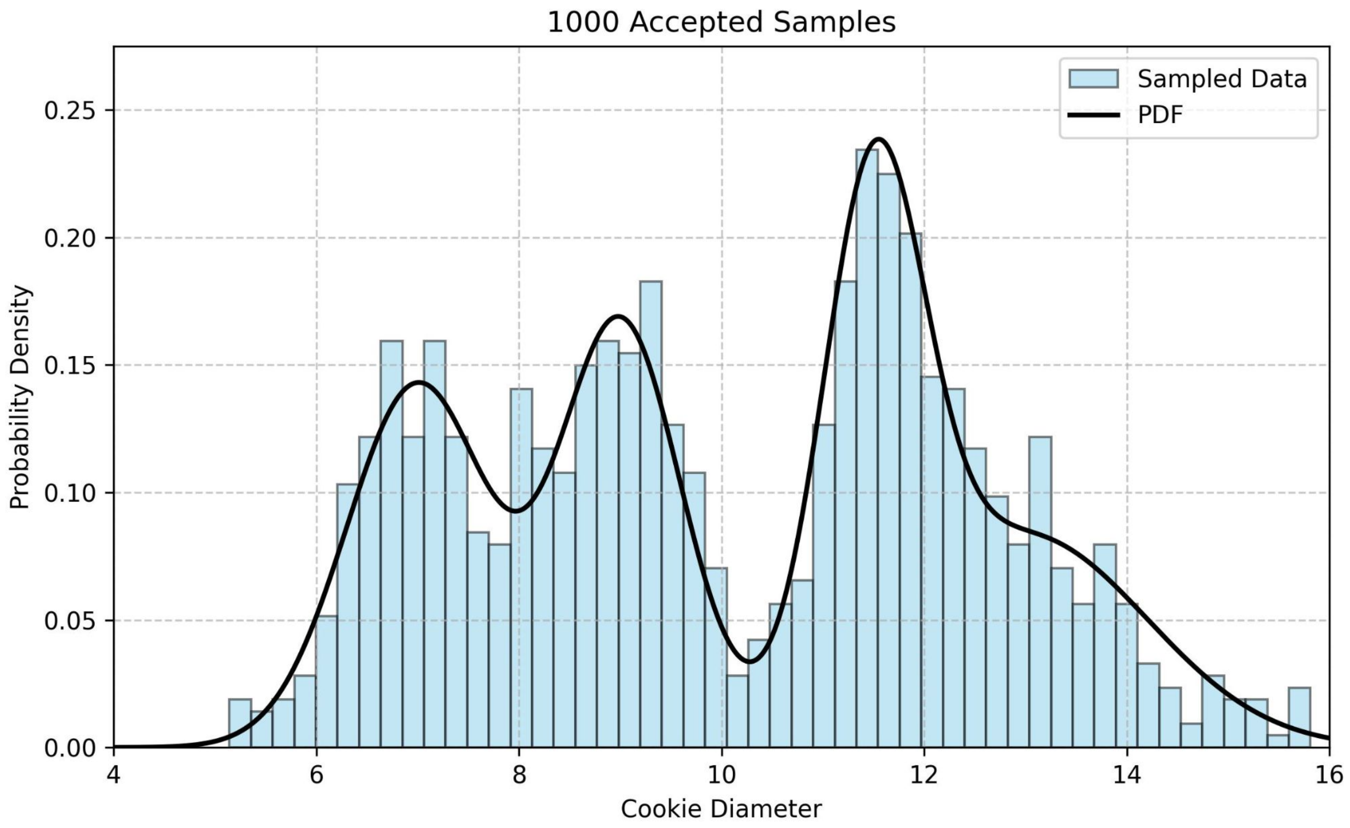
   Compute the acceptance ratio $r = f(x)/Mg(x)$.

   Accept the candidate sample x if $u \leq r$; otherwise, reject it.

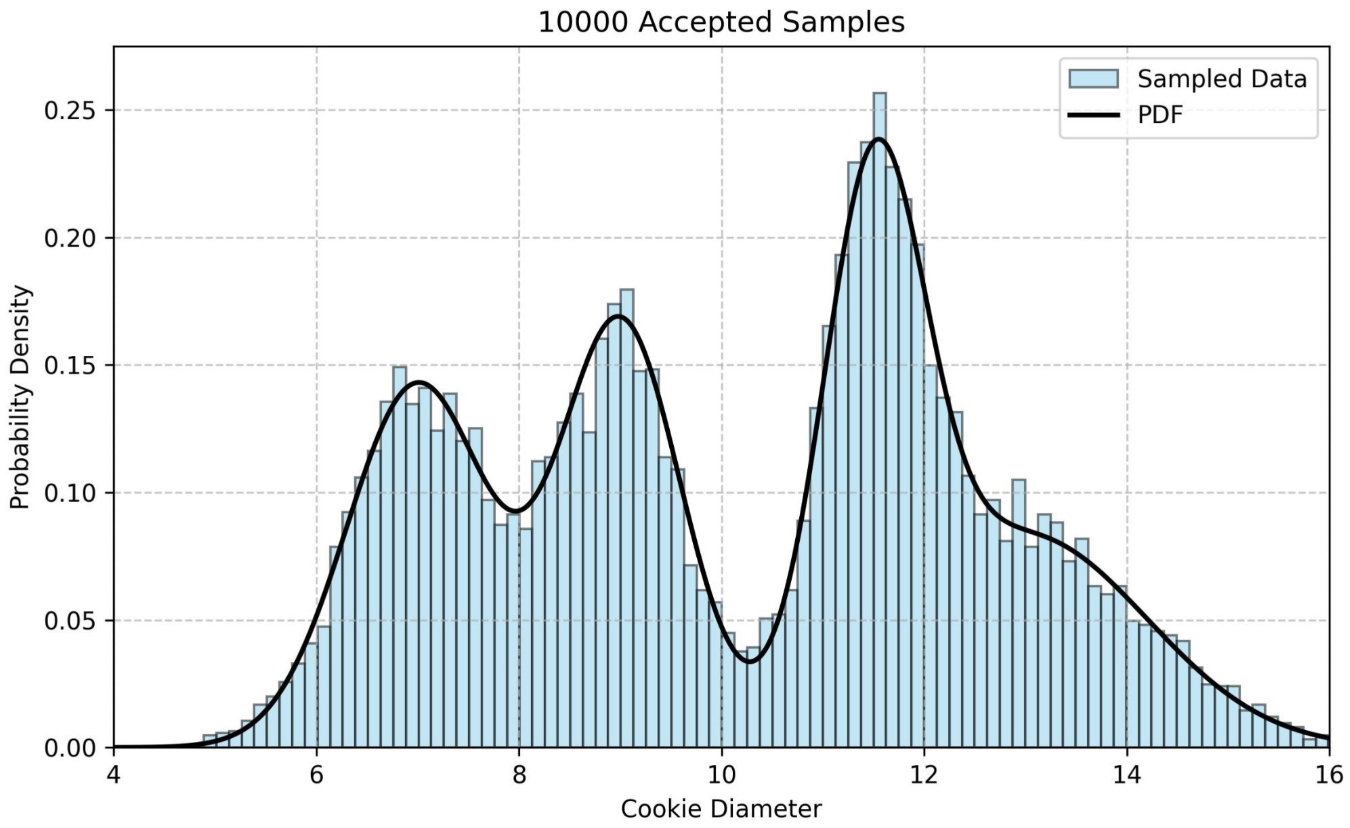
After 100 Accepted Samples…



After 1,000 Accepted Samples…



After 10,000 Accepted Samples…



Bayesian Rejection Sampling

- Connecting back to Bayesian modeling, we set:
  - The target distribution $h(\theta) = p(\theta)L(\theta|y)$ (a constant factor of the posterior density).
  - The proposal distribution $g(\theta) = p(\theta)$.
  - Let M be the minimum value for which $h(\theta)/g(\theta) \leq M$ for all θ. That is,

$$h(\theta)/g(\theta) = p(\theta)L(\theta|y)/p(\theta) = L(\theta|y) \leq M$$

  and so choose $M = L(\hat{\theta}|y)$, where $\hat{\theta}$ is the MLE.

- This way, if we wanted to sample from the **posterior predictive**, we have completed the first step of sampling from the **posterior distribution**!

Cookie Diameter: 1,000 Sample Results

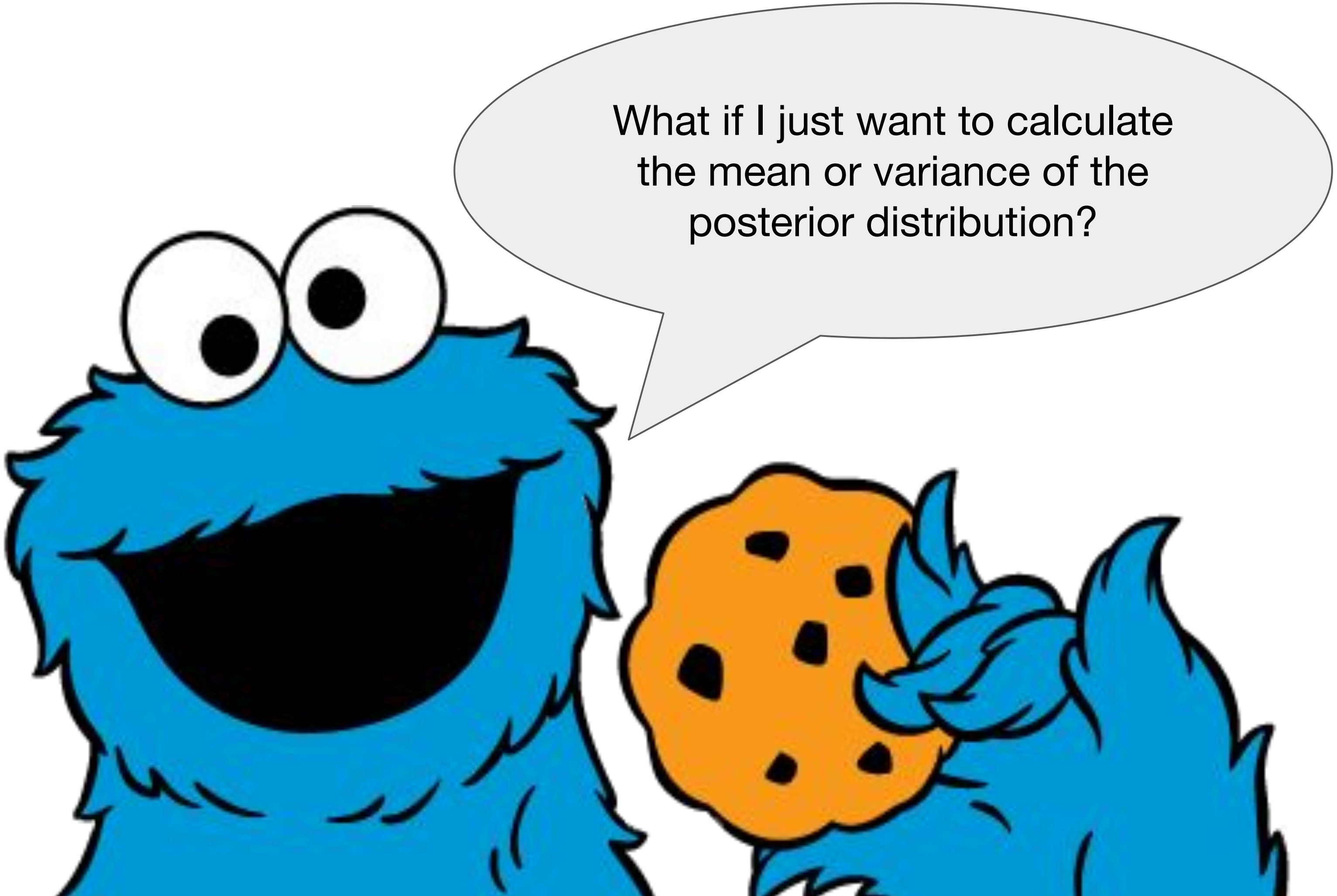
True Expected Value:	10.125
Estimated Expected Value:	10.226
True Variance:	5.932
Estimated Variance:	5.943

Cookie Diameter: 10,000 Sample Results

True Expected Value:	10.125
Estimated Expected Value:	10.167
True Variance:	5.932
Estimated Variance:	6.004

Cookie Diameter: 100,000 Sample Results **[**6 Second Runtime**]**

True Expected Value:	10.125
Estimated Expected Value:	10.130
True Variance:	5.932
Estimated Variance:	5.943


What if I just want to calculate the mean or variance of the posterior distribution?

Importance Sampling

1. Standard Monte Carlo Integration

- **Expectation Definition:**

$$\mathbb{E}_p[h(x)] = \int h(x)\, p(x)\, dx.$$

- **Monte Carlo Estimate:**

  If $x_1, x_2, \ldots, x_n \sim p(x)$, then

$$\mathbb{E}_p[h(x)] \approx \frac{1}{n} \sum_{i=1}^{n} h(x_i).$$

Importance Sampling

- However, the previous formula is **inefficient sometimes**; instead of sampling from the distribution, we can use **importance sampling**:
  1. Choose a proposal distribution *q(x)* from which it is easy to sample.
  2. Draw samples $x_1, x_2, \ldots, x_n$ from *q(x)*.
  3. Assign a weight to each sample based on the ratio of the target distribution *p(x)* to the proposal distribution *q(x)*:

$$w(x_i) = \frac{p(x_i)}{q(x_i)}$$

  4. Use the weighted samples to compute the desired expectation or integral.

Importance Sampling

- **Rewriting the Expectation:**

  When sampling from $p(x)$ is challenging, sample from a proposal distribution $q(x)$ instead:

$$\mathbb{E}_p[h(x)] = \int h(x)\, p(x)\, dx = \int h(x)\, \frac{p(x)}{q(x)}\, q(x)\, dx.$$

- **Importance Sampling Estimate:**

  With $x_1, x_2, \ldots, x_n \sim q(x)$ and weights

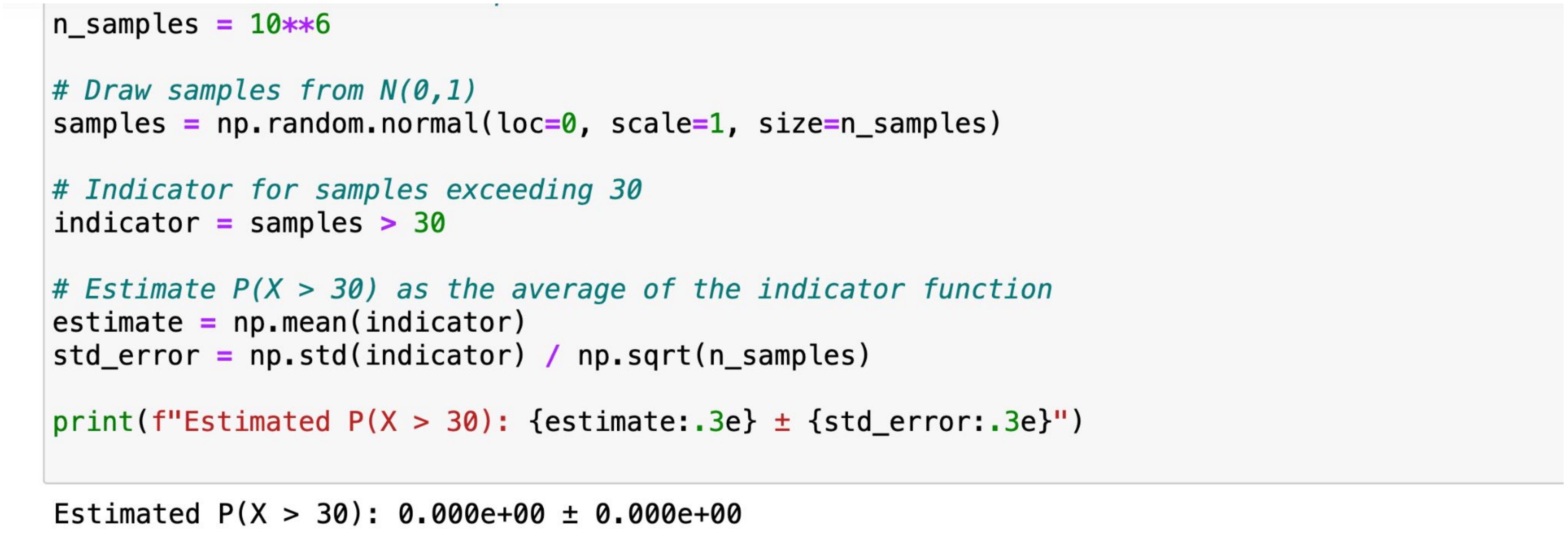
$$w(x_i) = \frac{p(x_i)}{q(x_i)},$$

  the expectation is approximated by

$$\mathbb{E}_p[h(x)] \approx \frac{1}{n} \sum_{i=1}^{n} h(x_i)\, w(x_i).$$

Naive Monte Carlo Inefficient

We want to estimate the tail probability $(P(X > 30)$ where $X \sim N(0, 1)$.
In **direct Monte Carlo sampling**, we draw samples from the target distribution (N(0,1)) and compute the fraction of samples that satisfy (X > 30).



```python
n_samples = 10**6

# Draw samples from N(0,1)
samples = np.random.normal(loc=0, scale=1, size=n_samples)

# Indicator for samples exceeding 30
indicator = samples > 30

# Estimate P(X > 30) as the average of the indicator function
estimate = np.mean(indicator)
std_error = np.std(indicator) / np.sqrt(n_samples)

print(f"Estimated P(X > 30): {estimate:.3e} ± {std_error:.3e}")
```

```
Estimated P(X > 30): 0.000e+00 ± 0.000e+00
```

Importance Sampling

1. **Sampling:**

   We sample $x_1, x_2, \ldots, x_n$ from the proposal distribution $q(x) = N(30, 1)$

   .

2. **Weights:**

   Each sample x is assigned a weight

   $$w(x) = \frac{p(x)}{q(x)} = \frac{\phi(x; 0, 1)}{\phi(x; 30, 1)},$$

   where $\phi(x; \mu, \sigma)$ denotes the normal density function.

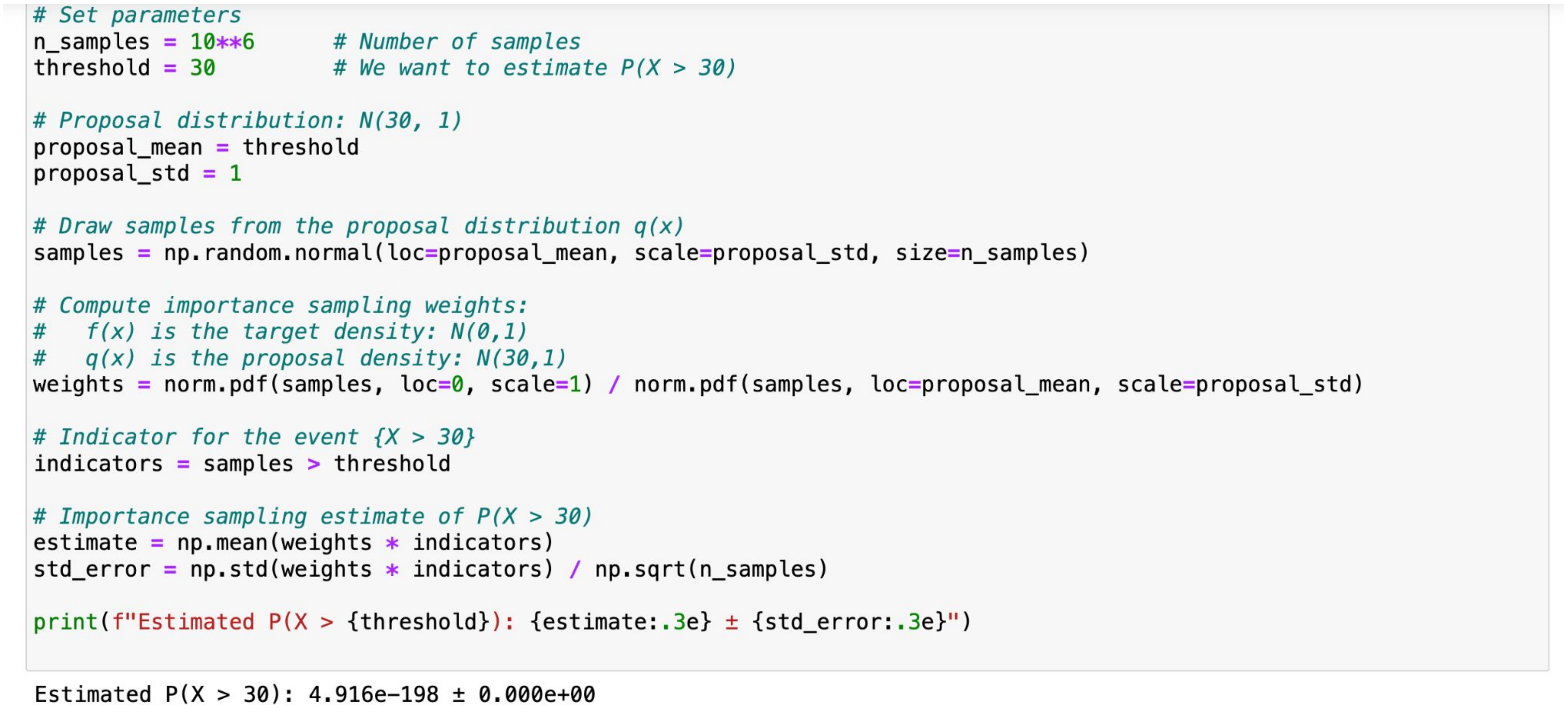
3. **Estimation:**

   The tail probability is estimated as

   $$\hat{P}(X > 30) \approx \frac{1}{n} \sum_{i=1}^{n} w(x_i)\, I(x_i > 30),$$

   with $I(x_i > 30)$ being the indicator function.

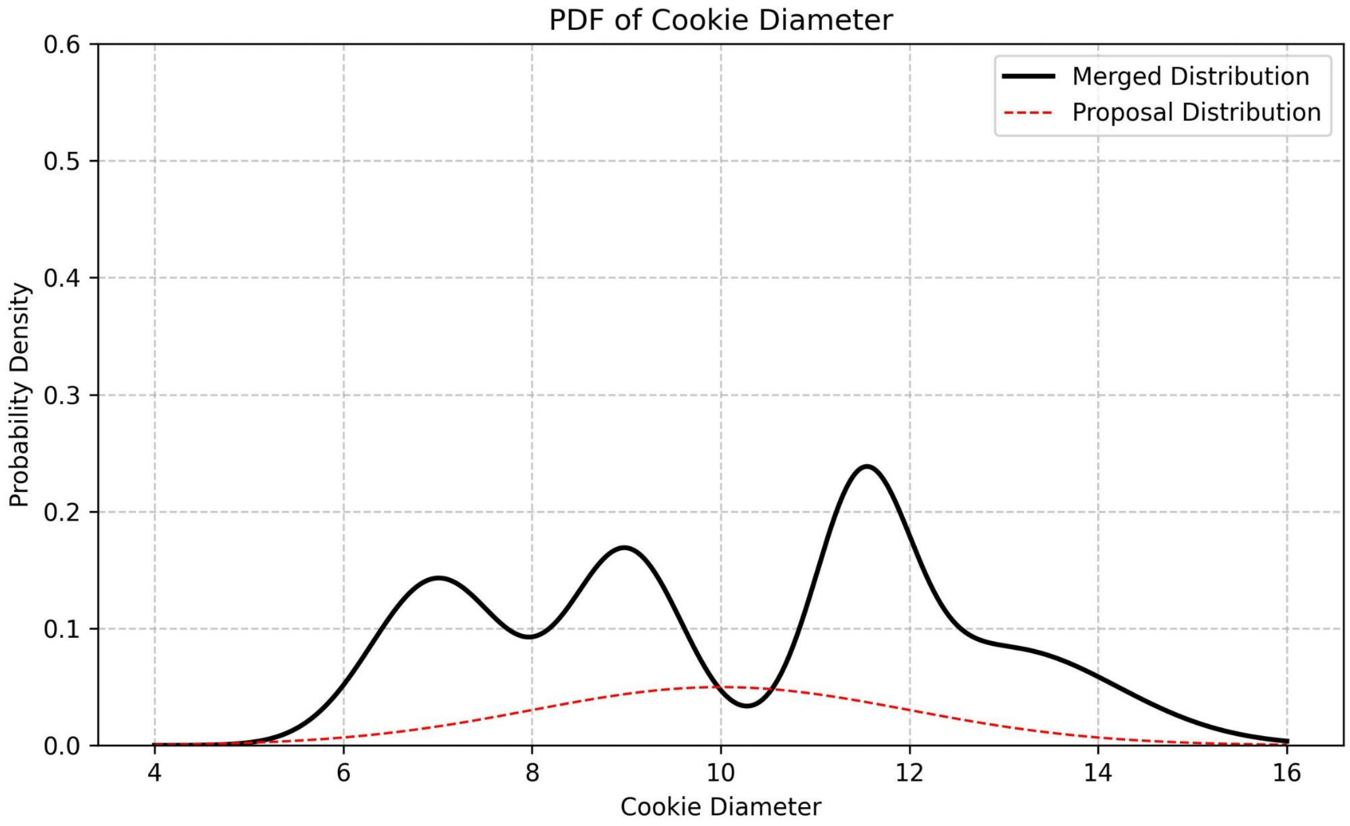
Importance Sampling



```
# Set parameters
n_samples = 10**6       # Number of samples
threshold = 30          # We want to estimate P(X > 30)

# Proposal distribution: N(30, 1)
proposal_mean = threshold
proposal_std = 1

# Draw samples from the proposal distribution q(x)
samples = np.random.normal(loc=proposal_mean, scale=proposal_std, size=n_samples)

# Compute importance sampling weights:
#   f(x) is the target density: N(0,1)
#   q(x) is the proposal density: N(30,1)
weights = norm.pdf(samples, loc=0, scale=1) / norm.pdf(samples, loc=proposal_mean, scale=proposal_std)

# Indicator for the event {X > 30}
indicators = samples > threshold

# Importance sampling estimate of P(X > 30)
estimate = np.mean(weights * indicators)
std_error = np.std(weights * indicators) / np.sqrt(n_samples)

print(f"Estimated P(X > {threshold}): {estimate:.3e} ± {std_error:.3e}")
```

```
Estimated P(X > 30): 4.916e-198 ± 0.000e+00
```

Let’s Use the Same Distribution from Rejection Sampling: $\mathcal{N}(10, 2)$



Limitation of Rejection & Importance Sampling

- The main problem with these methods is **computational efficiency**:
  - It could take ages in rejection sampling before a sample is accepted.
  - It could take many many samples in importance sampling before reweighted samples are sufficiently reliable to calculate posterior statistics.
  - With both methods, the choice of proposal distribution makes a massive difference.

Another Approach: MCMC!

- Markov Chain Monte Carlo is another method we can use.
    - **Markov Chain**: a memoryless random process.
    - **Monte Carlo**: estimates based on random samples.
- **Main Idea**: Construct a random process which, as it evolves, has outputs that start resembling samples from the posterior distribution.

What is a Markov Chain?

- A Markov Chain is an algorithm that evolves in discrete times steps.
- $\theta^{(T)} \subset \Omega$ denotes state of algorithm at step T.
  - State space Ω is a set of all possible states for the algorithm.
- We denote the evolution of the algorithm using $\theta^{(0)}, \theta^{(1)}, \ldots, \theta^{(T)}$.
- We use:

$$P(\theta^{(T)} | \theta^{(T-1)}, \ldots, \theta^{(0)})$$

to denote the probability density of $\theta^{(T)}$ given the history of the process.

The Markov Property

- A central property of Markov Chains is that updates are random but memoryless:

$$p(\theta^{(T)} \mid \theta^{(T-1)}, \ldots, \theta^{(0)}) = p(\theta^{(T)} \mid \theta^{(T-1)})$$

  - It doesn't matter where I've been.
  - It only matters where I am currently at.
- **So how do we use this idea to sample from a posterior distribution?**

Metropolis-Hastings Algorithm

- Let $q(y \mid x)$ denote a chosen proposal distribution (we pick this).
    - $q(y \mid x)$ gives density for moving to y from x.
- Let $p(\theta \mid \mathrm{data}) = Cf(\theta)$ be our target limiting distribution.
- Algorithm (One Step): Currently at $\theta^{(T)}$:
    - Generate proposal: $\theta^* \sim q(y \mid \theta^{(T)})$.
    - Compute acceptance probability:

$$a = \min\left\{1, \frac{Cf(\theta^*)q(\theta^{(T)} \mid \theta^*)}{Cf(\theta^{(T)})q(\theta^* \mid \theta^{(T)})}\right\}$$

$$= \min\left\{1, \frac{f(\theta^*)q(\theta^{(T)} \mid \theta^*)}{f(\theta^{(T)})q(\theta^* \mid \theta^{(T)})}\right\}$$

    - With probability a set $\theta^{(T+1)} = \theta^*$. Otherwise, set $\theta^{(T+1)} = \theta^{(T)}$.

Interpreting Metropolis Hastings

$$a = \min\left\{1, \frac{f(\theta^*)q(\theta^{(T)} \mid \theta^*)}{f(\theta^{(T)})q(\theta^* \mid \theta^{(T)})}\right\}$$

- Uphill proposals (ones that take the Markov Chain to a local maximum) are always accepted.
- Downhill proposals (ones that move away from a local maximum) are accepted with probability equal to the relative heights of the posterior density at the proposed and current values.

Putting Things Into Practice: Skittles

The company behind skittles is contemplating a new flavor: mango. They need to figure out how much of their secret flavoring to add to each skittle to maximize the number of people who enjoyed the new flavor. The company shared the following taste test data:

Secret Flavoring (mg)	Taste Testers	Loved the Flavor
1.6907	59	6
1.7242	60	13
1.7552	62	18
1.7842	56	28
1.8113	63	52
1.8369	59	52
1.8610	62	61
1.8839	60	60

Model Probability for Flavor Success

- For each of the 8 observations: $y_i \sim \mathrm{Bin}(n_i, p_i)$.
  - Meaning $p(y_i|p_i) = \binom{n_i}{y_i} p_i^{y_i}(1-p_i)^{n_i-y_i}$.
- We further assume that $\mathrm{logit}\ p_i = \alpha + \beta x_i$.
  - This is equivalent to saying $p_i = \dfrac{1}{1+\exp\left(-(\alpha+\beta x_i)\right)}$.
- So, our probability model for the data is:

$$y_i \sim \mathrm{Bin}\left(n_i, \frac{1}{1+\exp\left(-(\alpha+\beta x_i)\right)}\right)$$

- The goal is to perform statistical inference for two parameters, α and β.

Prior Distributions

- We choose proper priors for both parameters of interest:

$$\alpha \sim \mathcal{N}(0, 10000)$$

$$\beta \sim \mathcal{N}(0, 10000)$$

- This acts like a uniform distribution for each parameter because the variances are so large.
- By Bayes rule, the posterior density can be written as:

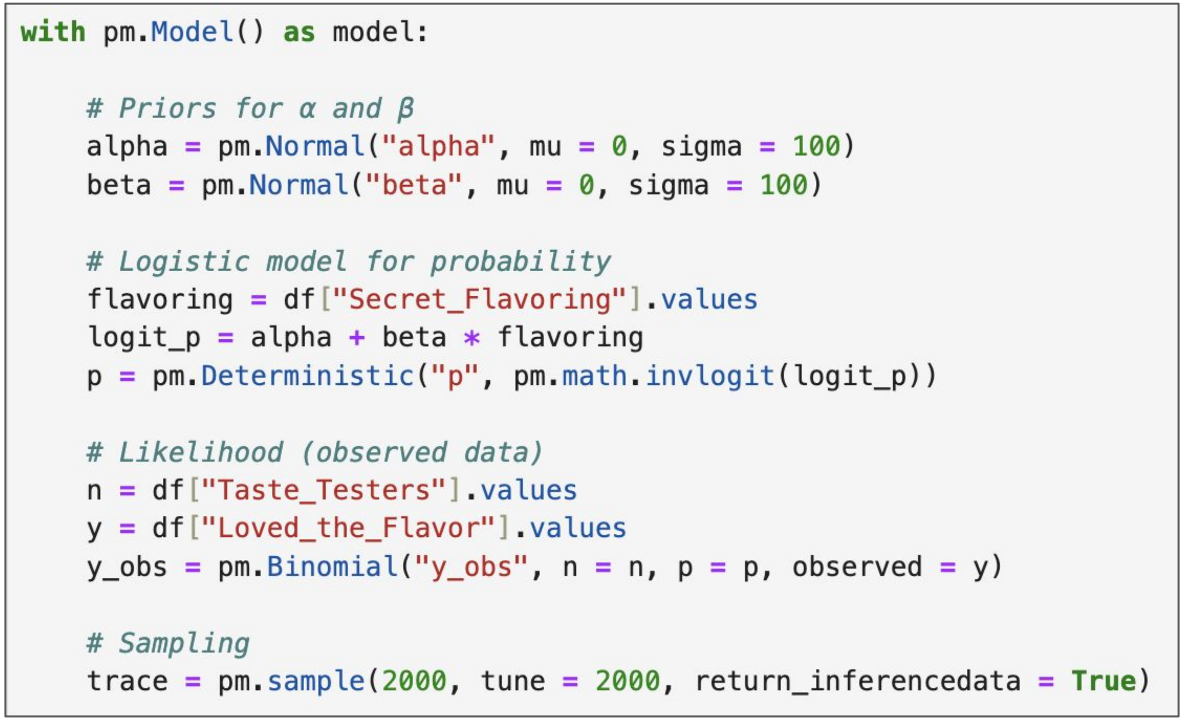
$$p(\alpha, \beta | y) = c \cdot \mathcal{N}(\alpha | 0, 10000) \cdot \mathcal{N}(\beta | 0, 10000) \cdot \prod_{i=1}^{8} p_i^{y_i} (1 - p_i)^{n_i - y_i}$$

Using pymc for Modeling

The pymc library allows us to abstract away from the details of implementing MCMC and the Metropolis-Hastings algorithm by hand:

1. It runs several parallel MCMC samplers with different starting values.
2. It simulates values from the Markov chains for a "burn-in" period (before the Markov chains have converged to the stationary distribution), and discard the burn-in simulations.
3. It saves simulated values after the burn-in period. These will be the simulated values on which we can perform inferential summaries.

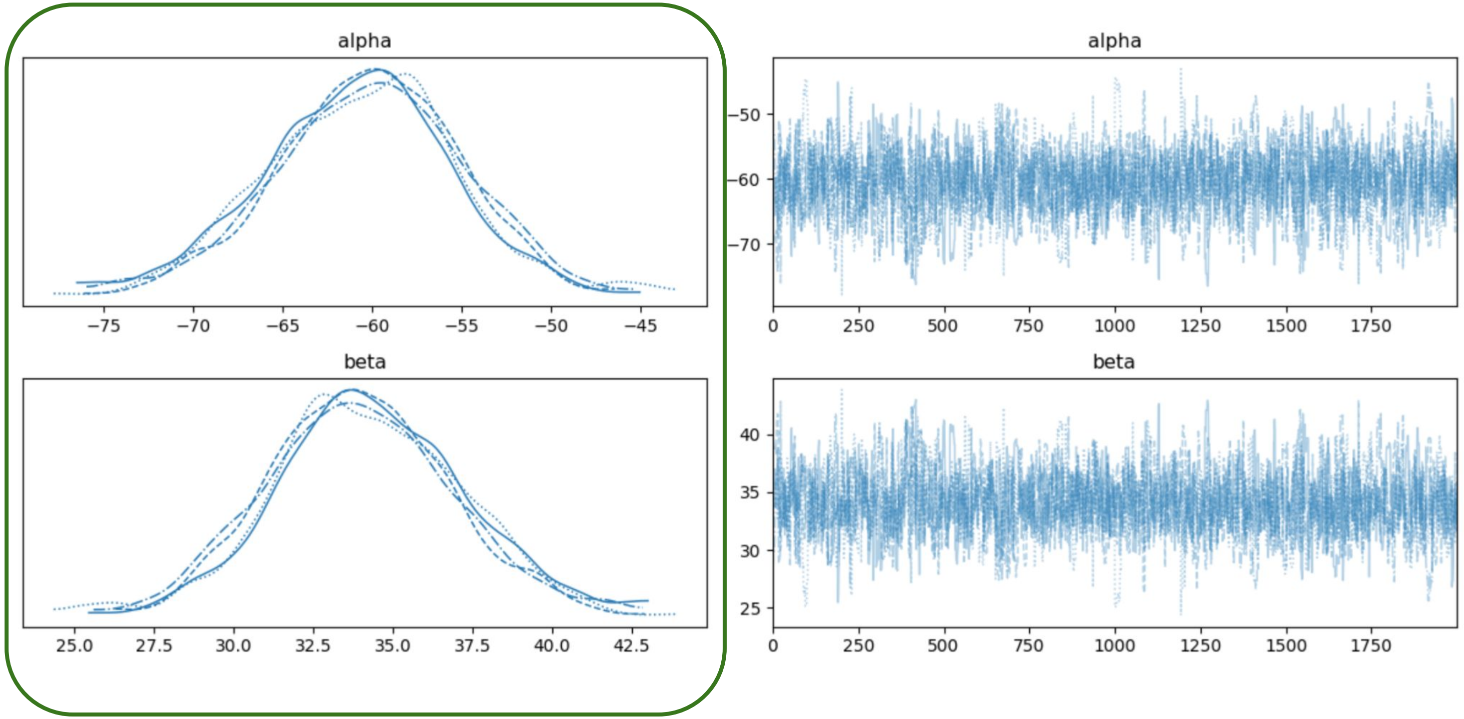
Using pymc for Modeling (continued...)



```python
with pm.Model() as model:

    # Priors for α and β
    alpha = pm.Normal("alpha", mu = 0, sigma = 100)
    beta = pm.Normal("beta", mu = 0, sigma = 100)

    # Logistic model for probability
    flavoring = df["Secret_Flavoring"].values
    logit_p = alpha + beta * flavoring
    p = pm.Deterministic("p", pm.math.invlogit(logit_p))

    # Likelihood (observed data)
    n = df["Taste_Testers"].values
    y = df["Loved_the_Flavor"].values
    y_obs = pm.Binomial("y_obs", n = n, p = p, observed = y)

    # Sampling
    trace = pm.sample(2000, tune = 2000, return_inferencedata = True)
```

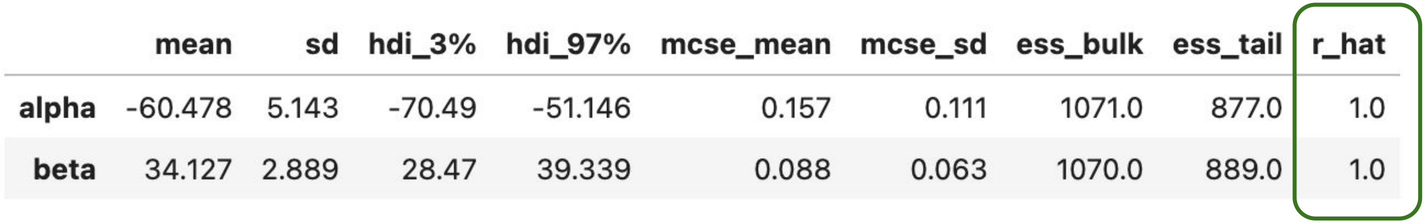
Interpreting pymc Results

The pymc library gives us sampled posterior distributions for the parameters:



Interpreting pymc Results

- We are also provided with some summary information on the stability of the sampling:



	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
alpha	-60.478	5.143	-70.49	-51.146	0.157	0.111	1071.0	877.0	1.0
beta	34.127	2.889	28.47	39.339	0.088	0.063	1070.0	889.0	1.0

- The r_hat statistic is a numerical measure that indicates whether the Markov chain was run long enough to reach convergence.
    - Values near 1.0 indicate convergence.
    - Large values (say 1.3 or greater) indicate either that the procedure has not been run long enough, or that the parameter itself may be difficult to obtain reasonable samples given strong autocorrelation in the sampler.
    - The statistic essentially computes a ratio of between-chain variance and within-chain variance.