

```
In [1]: import numpy as np
import pandas as pd
from scipy import stats

%matplotlib inline

import matplotlib.pyplot as plt
```

```
In [2]: planets = pd.read_csv('exoplanets.csv')
planets.shape
```

```
Out[2]: (517, 6)
```

Independence and Conditional Independence

In progress

Approximate Inference and Sampling, Part 1

Why we need posterior distributions

In general, we need the posterior distribution so that we can make statements and decisions about our unknown quantity of interest, θ . We saw that for simple models like the product review model or the model for heights, it was easy to compute the posterior exactly, because we chose a conjugate prior.

In the product review example:

- Our parameter of interest θ represents the probability of a positive review.
- If we chose a Beta prior, i.e., $\theta \sim \text{Beta}(\alpha, \beta)$, then the posterior distribution also belonged to the Beta family: $\theta|x \sim \text{Beta}(\alpha + \sum x_i, \beta + n - \sum x_i)$.
- This made it easy to determine things like the MAP estimate or LMSE estimate, simply by using known properties of the Beta distribution.

But what if our posterior distribution didn't have such a convenient form? In that case, we would have to compute the posterior (and any estimates from it) ourselves:

$$p(\theta|x) = \frac{p(x|\theta)p(\theta)}{p(x)} \quad (1)$$

$$= \frac{p(x|\theta)p(\theta)}{\int p(x|\theta)p(\theta) d\theta} \quad (2)$$

In general, the integral in the denominator could be impossible to compute. We call the denominator the **normalizing constant**: it's a constant because it doesn't depend on θ , and it's normalizing because we need it for the distribution or density to sum or integrate to 1.

In the next section, we'll see a few examples that illustrate why computing the normalizing constant is hard, but first, let's examine why we need to know it in the first place.

MAP Estimation

Suppose we want to compute the MAP estimate:

$$\hat{\theta}_{MAP} = \operatorname{argmax}_{\theta} p(\theta|x) \quad (3)$$

$$= \operatorname{argmax}_{\theta} \frac{p(x|\theta)p(\theta)}{p(x)} \quad (4)$$

$$= \operatorname{argmax}_{\theta} p(x|\theta)p(\theta) \quad (5)$$

In the last step, we used the fact that $p(x)$ doesn't depend on θ .

If θ is low-dimensional and continuous, we can easily optimize this either analytically or sometimes numerically. If θ is discrete and doesn't take on too many different values, we can search over all possible values. However, if θ is discrete and takes on an intractably large number of possible values, then we'd need to search over all of them, which would be impossible.

To summarize: for low-dimensional continuous variables, or discrete random variables with a low number of possible values, we can compute the MAP estimate without needing to know the exact posterior. For higher-dimensional random variables and/or discrete random variables with many possible values, this won't work.

LMSE Estimation

Suppose we want to compute the LMSE estimate. Recall the definition of conditional expectation (see Data 140 textbook, [Chapter 9](#) and [Chapter 15](#)):

$$\hat{\theta}_{LMSE} = E_{\theta|x}[\theta] \quad (6)$$

$$= \int \theta \cdot p(\theta|x) d\theta \quad (7)$$

$$= \int \theta \cdot \frac{p(x|\theta)p(\theta)}{p(x)} d\theta \quad (8)$$

$$= \frac{1}{p(x)} \int \theta \cdot p(x|\theta)p(\theta) d\theta \quad (9)$$

In order to compute the LMSE estimate, we need to compute the denominator, $p(x)$. If we don't know it, then our estimate will be off by a multiplicative factor that we don't know, making it effectively useless.

The same is true for computing the expected value of any other function of θ , or any other probability involving the posterior distribution. Answering any of the following questions will lead to the same problem:

- According to the posterior distribution, what is the variance of θ ?
- According to the posterior distribution, what is the probability that θ is greater than 0.5?

To summarize: any computations involving the posteriors (probabilities, expectations, etc.) require us to have the full normalized distribution: the numerator in Bayes' rule isn't enough.

Why computing posterior distributions is hard

In simple models like our product review model or our model for heights, it was easy to compute the exact posterior for the unknown variable that we were interested in. This happened because we chose a

conjugate prior. In most other cases, computing the exact posterior is hard! Here are two examples:

One-dimensional non-conjugate prior

Let's return to the product review example, but this time, instead of a Beta prior, we choose

$$p(\theta) = \frac{2}{\pi} \cos\left(\frac{\pi}{2}\theta\right) \text{ for } \theta \in [0, 1].$$

$$p(\theta|x) \propto p(x|\theta)p(\theta) \quad (10)$$

$$\propto \left[\theta^{\sum_i x_i} (1 - \theta)^{\sum_i (1-x_i)} \right] \cos\left(\frac{\pi}{2}\theta\right) \quad (11)$$

This distribution looks much more complicated: we can't reduce it to a known distribution at all. So, in order to properly compute $p(\theta|x)$, we'd need to figure out the normalizing constant. This requires solving the integral:

$$p(x) = \int_0^1 \left[\theta^{\sum_i x_i} (1 - \theta)^{\sum_i (1-x_i)} \right] \cos\left(\frac{\pi}{2}\theta\right) d\theta \quad (12)$$

This integral is difficult to solve in closed form. However, since this is a one-dimensional problem, we could take advantage of numerical integration. For a particular sequence of values x_1, \dots, x_n , we can compute a numerical approximation to the integral, and find the normalizing constant that way. As we saw above, we don't need the normalizing constant if we're only interested in the MAP estimate, but we can't compute the LMSE estimate without it.

Multi-dimensional example

Consider the exoplanet model from last time: x_i is the (observed) radius of planet i , z_i is whether the planet belongs to group 0 (small, possibly habitable planets) or group 1 (large, possibly inhabitable planets), and μ_0 and μ_1 are the mean radii of those two groups, respectively.

$$z_i \sim \text{Bernoulli}(\pi) \quad i = 1, \dots, n \quad (13)$$

$$\mu_k \sim \mathcal{N}(\mu_p, \sigma_p) \quad k = 0, 1 \quad (14)$$

$$x_i | z_i, \mu_0, \mu_1 \sim \mathcal{N}(\mu_{z_i}, \sigma) \quad i = 1, \dots, n \quad (15)$$

We can write the likelihood and prior. To simplify, we'll write $\mathcal{N}(y; m, s) = \frac{1}{s\sqrt{2\pi}} \exp\left\{-\frac{1}{2s^2}(y - m)^2\right\}$

$$p(z_i) = \pi^{z_i} (1 - \pi)^{1-z_i} \quad (16)$$

$$p(\mu_k) = \mathcal{N}(\mu_k; \mu_p, \sigma_p) \quad (17)$$

$$p(x_i | z_i, \mu_0, \mu_1) = \mathcal{N}(x_i; \mu_{z_i}, \sigma) \quad (18)$$

We can try computing the posterior over the hidden variables z_i , μ_0 , and μ_1 . We'll use the notation $z_{1:n}$ to represent z_1, \dots, z_n (and similarly for $x_{1:n}$).

$$p(z_{1:n}, \mu_0, \mu_1 | x_{1:n}) \propto p(\mu_0)p(\mu_1) \prod_i [p(z_i)p(x_i | z_i, \mu_0, \mu_1)] \quad (19)$$

This distribution is more complicated than anything we've seen up until now. It's the joint distribution over $n + 2$ random variables (the group labels z_1, \dots, z_n and the two group means μ_0 and μ_1).

Computing the normalization constant $p(x_{1:n})$ requires a complicated combination of sums and integrals:

$$p(x_{1:n}) = \sum_{z_1=0}^1 \sum_{z_2=0}^1 \cdots \sum_{z_n=0}^1 \int \int p(\mu_0) p(\mu_1) \prod_i [p(z_i) p(x_i | z_i, \mu_0, \mu_1)] d\mu_0 d\mu_1 \quad (20)$$

For our dataset of over 500 planets, the sum alone would require a completely intractable amount of computation:

In [3]: `2**517`

Out[3]: 429049853758163107186368799942587076079339706258956588087153966199096448962353503257659977541340909686081019461967553627320124249982290238285876768194691072

Worse still, we can't even compute the MAP estimate for the labels z_i : in order to find the one that maximizes the numerator, we'd have to search over all 2^{517} combinations, which is also completely intractable.

Even in this fairly simple model, with two groups, we've found that exact inference is completely hopeless: there's no way we can compute the exact posterior for all our unknowns. In the rest of this notebook, we'll talk about ways to get around this problem using approximations to the posterior distribution.

Approximation with Samples

We've seen before that we can compute an empirical distribution from a sample of data points. In this next section, we'll use sampling to approximate distributions.

Let's start by using samples to approximate a known, easy-to-compute distribution: Beta(3, 4).

```
In [4]: import numpy as np
from scipy import stats

%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()

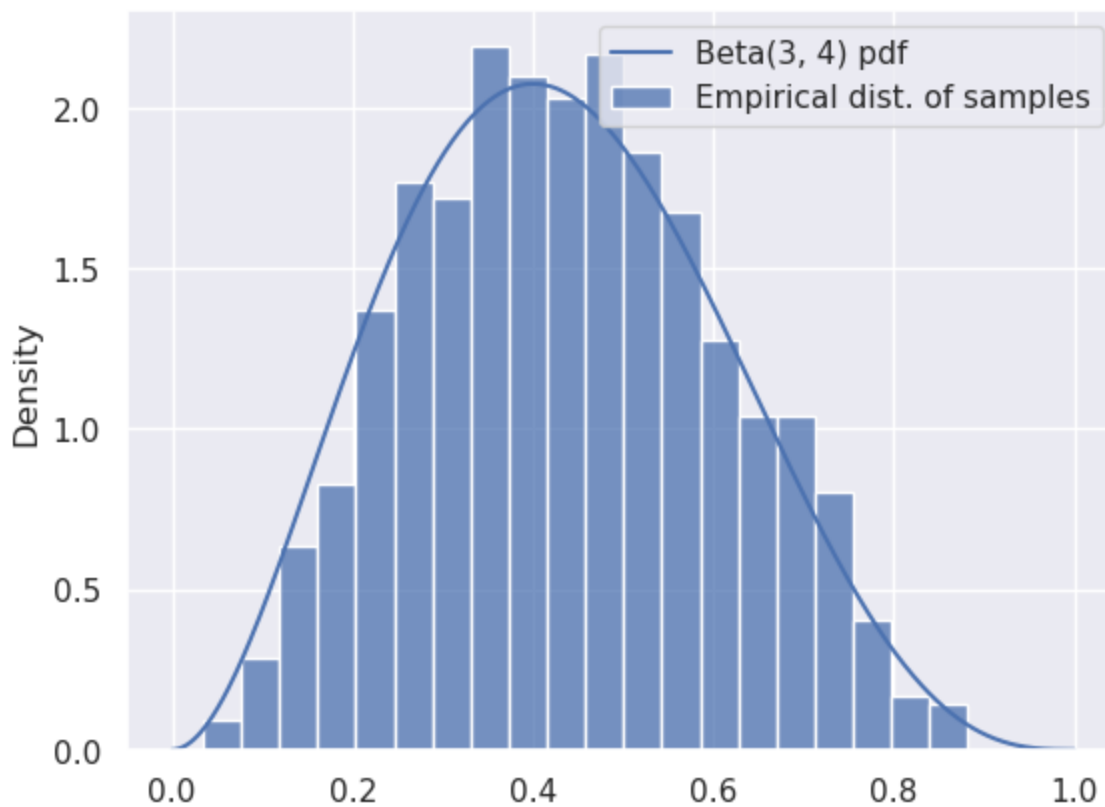
distribution = stats.beta(3, 4)

# Compute the exact PDF:
t = np.linspace(0, 1, 500)
pdf = distribution.pdf(t)

# Draw 1000 samples, and look at the empirical distribution of those samples:
samples = distribution.rvs(1000)
f, ax = plt.subplots(1, 1)

sns.histplot(x=samples, stat='density', bins=20, label='Empirical dist. of samples')
ax.plot(t, pdf, label='Beta(3, 4) pdf')
ax.legend()
```

Out[4]: <matplotlib.legend.Legend at 0x7fd0c573b880>



We can see that the samples are a good representation for the distribution, as long as we have enough. We can use the mean of the samples to approximate the mean of the distribution:

```
In [5]: # The mean of a Beta(a, b) distribution is a/(a+b):
true_mean = 3 / (3 + 4)

approx_mean = np.mean(samples)
print(true_mean, approx_mean)

0.42857142857142855 0.43921158050695713
```

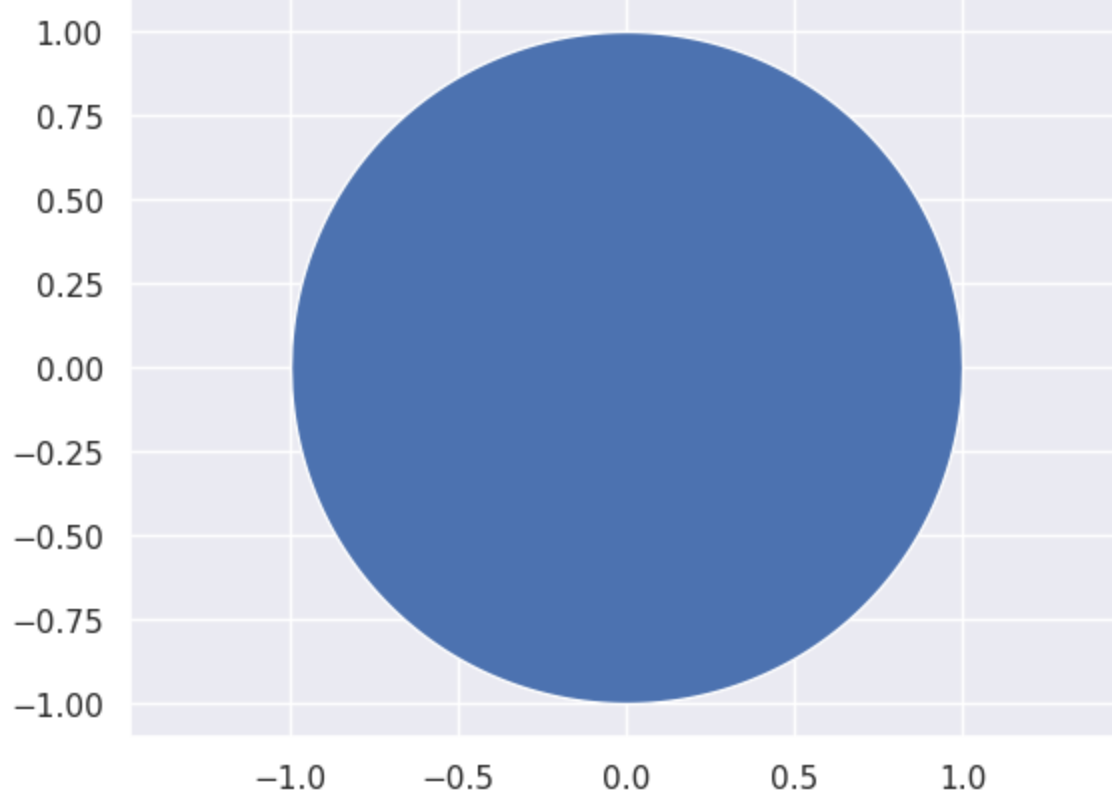
Rejection Sampling

As a warmup, let's suppose that we want to sample a pair of random variables (x_1, x_2) drawn uniformly from the unit circle. In other words, we want the uniform distribution over the blue region below:

How can we go about doing this?

(Hint: first sample uniformly over the unit square.)

```
In [6]: x_ = np.linspace(-1, 1, 1000)
semicircle = np.sqrt(1-x_**2)
plt.fill_between(x_, -semicircle, semicircle)
plt.axis('equal');
```



```
In [7]: # Number of samples
N = 400

# Samples in the unit square
samples = np.random.uniform(-1, 1, [N, 2])
```

```
In [8]: # Number of samples
N = 400

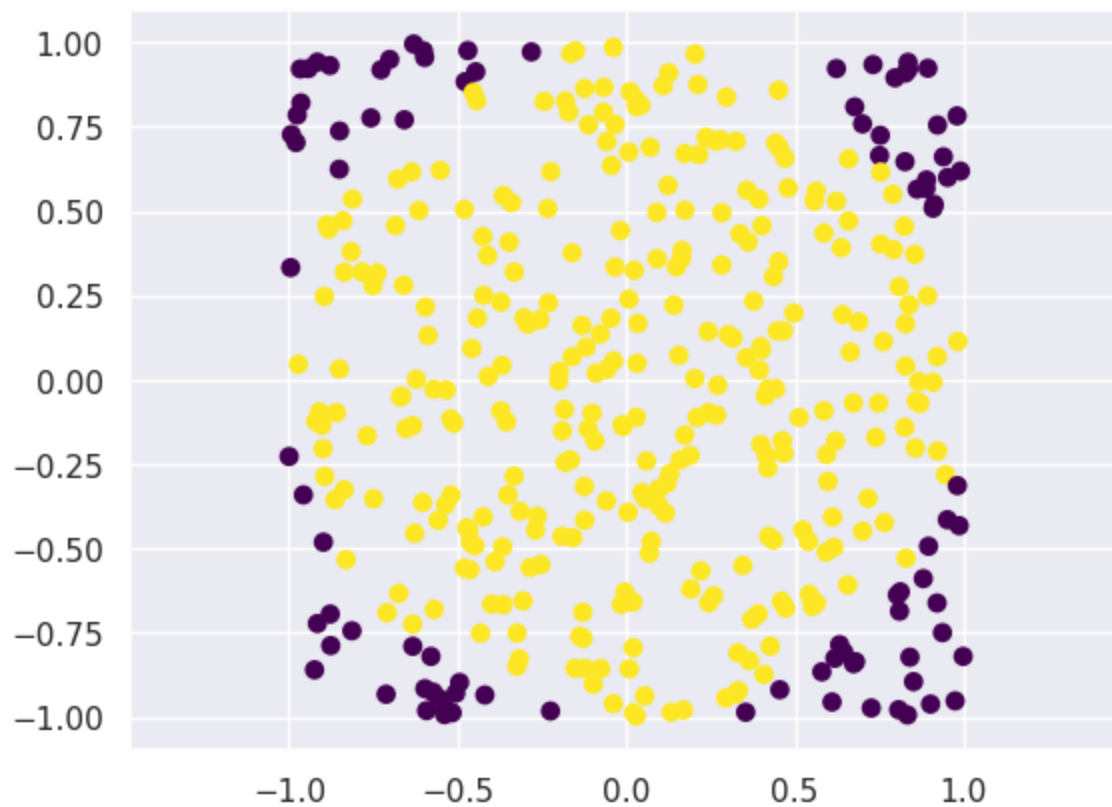
# Samples in the unit square
samples = np.random.uniform(-1, 1, [N, 2])

# Which ones are inside the unit circle?
is_in_circle = (samples[:,0]**2 + samples[:, 1]**2) < 1

plt.figure()
plt.scatter(samples[:, 0], samples[:, 1], c=is_in_circle, cmap='viridis')
plt.axis('equal')

good_samples = samples[is_in_circle]
x1 = good_samples[:, 0]
x2 = good_samples[:, 1]
print('Variance of x1 (estimated from samples): %.3f' % np.var(x1))
```

Variance of x1 (estimated from samples): 0.231



Next, let's think about sampling from a distribution with a complicated density. Suppose we want to sample from the distribution with density $p(\theta|x) \propto \theta \cdot (1.5 - \theta) \cdot \sin(\theta)$ for $\theta \in [0, 1.5]$:

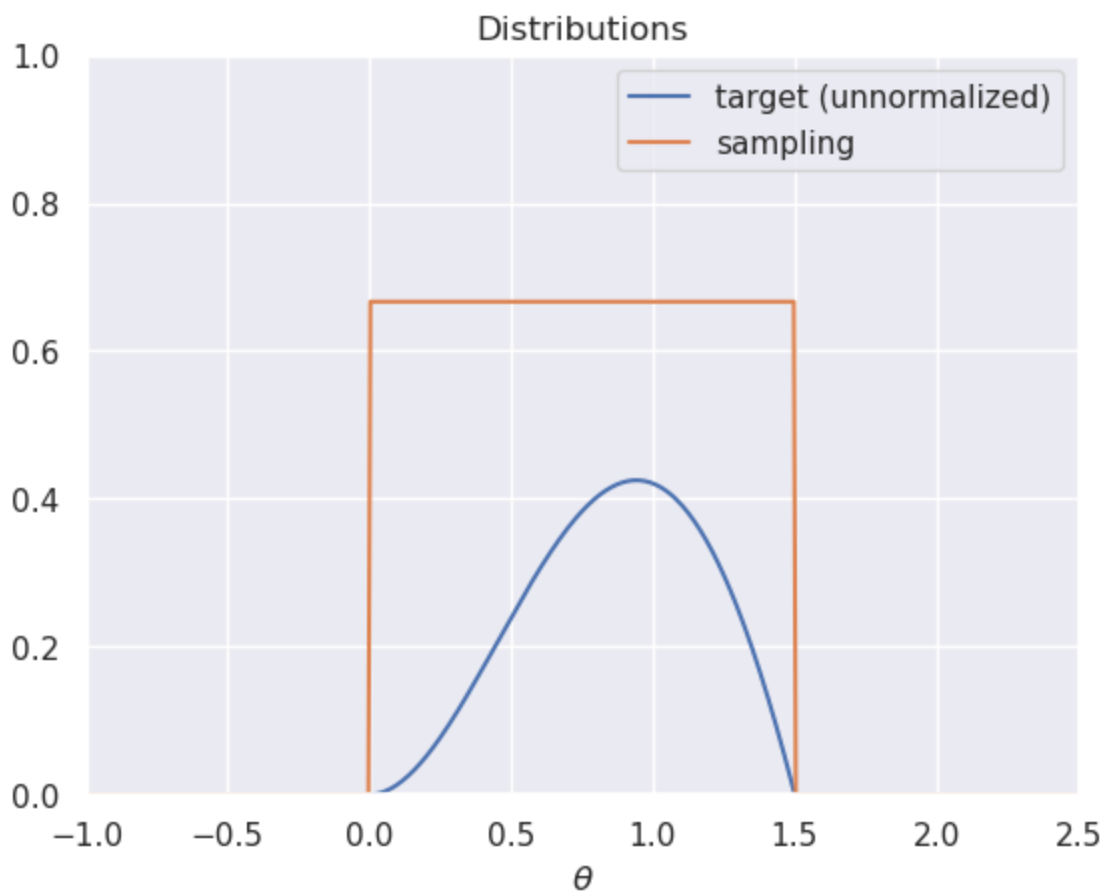
```
In [9]: t = np.linspace(-1, 2.5, 500)
def target(t):
    """The unnormalized distribution we want to sample from"""
    return t * (1.5-t) * np.sin(t) * ((t > 0) & (t < 1.5))
plt.plot(t, target(t))
plt.title('Target distribution')
plt.xlabel(r'\theta$')
plt.axis([-1,2.5,0,1])
plt.show()
```



How can we make this look like the geometric example from before? Idea: "lifting" (add one dimension).

```
In [10]: x = np.linspace(-1, 2.5, 500)
def uniform_sampling_dist(t):
    """PDF of distribution we're sampling from: Uniform[0, 1.5]"""
    return stats.uniform.pdf(t, 0, 1.5)

plt.plot(t, target(t), label='target (unnormalized)')
plt.plot(t, uniform_sampling_dist(t), label='sampling')
plt.axis([-1, 2.5, 0, 1])
plt.legend()
plt.title('Distributions')
plt.xlabel(r'$\theta$')
plt.show()
```

```
In [11]: def rejection_sample_uniform(num_samples=100):
# Generate proposals for samples: these are  $\theta$ -values.
# We'll keep some and reject the rest.
proposals = np.random.uniform(low=0, high=1.5, size=num_samples)

# Acceptance probability is the ratio of the two curves
# These had better all be between 0 and 1!
accept_probs = target(proposals) / uniform_sampling_dist(proposals)

print('Max accept prob: %.3f' % np.max(accept_probs))

# For each sample, we make a decision whether or not to accept.
# Convince yourself that this line makes that decision for each
# sample with prob equal to the value in "accept_probs"!
accept = np.random.uniform(size=num_samples) < accept_probs

num_accept = np.sum(accept)
print('Accepted %d out of %d proposals' % (num_accept, num_samples))
return proposals[accept]
```

```
In [12]: samples = rejection_sample_uniform(num_samples=100000)

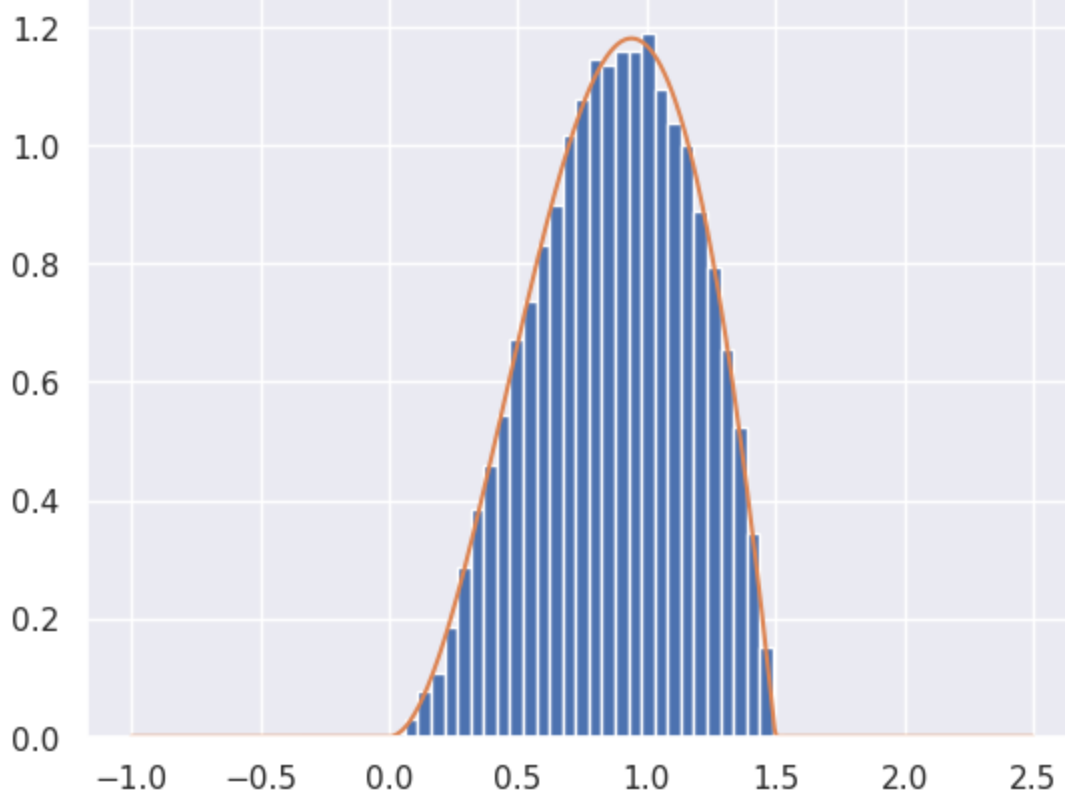
# Plot a true histogram (comparable with density functions) using density=True
plt.hist(samples, bins=np.linspace(-0.5, 2, 50), density=True)

# Where did this magic number 0.36 come from? What happens if you change it?
plt.plot(t, target(t)/0.36)
```

Max accept prob: 0.638

Accepted 36401 out of 100000 proposals

```
Out[12]: [<matplotlib.lines.Line2D at 0x7fd0bc3a7490>]
```



As a final example, what happens if we want to sample across the entire real line? For instance, suppose our density is $p(\theta|x) \propto \exp(-\theta) |\sin(2\theta)|$ for $\theta \in [0, \infty)$. We certainly can't use a uniform proposal distribution, but using the exponential distribution works just fine.

```
In [13]: def decaying_target_distribution(t):
          """Unnormalized target distribution as described above"""
          return np.exp(-t) * np.abs(np.sin(2*t))

def sampling_distribution_exponential(t):
    """Sampling distribution: exponential distribution"""
    # stats.expon has a loc parameter which says how far to shift
    # the distribution from its usual starting point of theta=0
    return stats.expon.pdf(t, loc=0, scale=1.0)

def rejection_sample_exponential(num_samples=100):
    proposals = np.random.exponential(scale=1.0, size=num_samples)

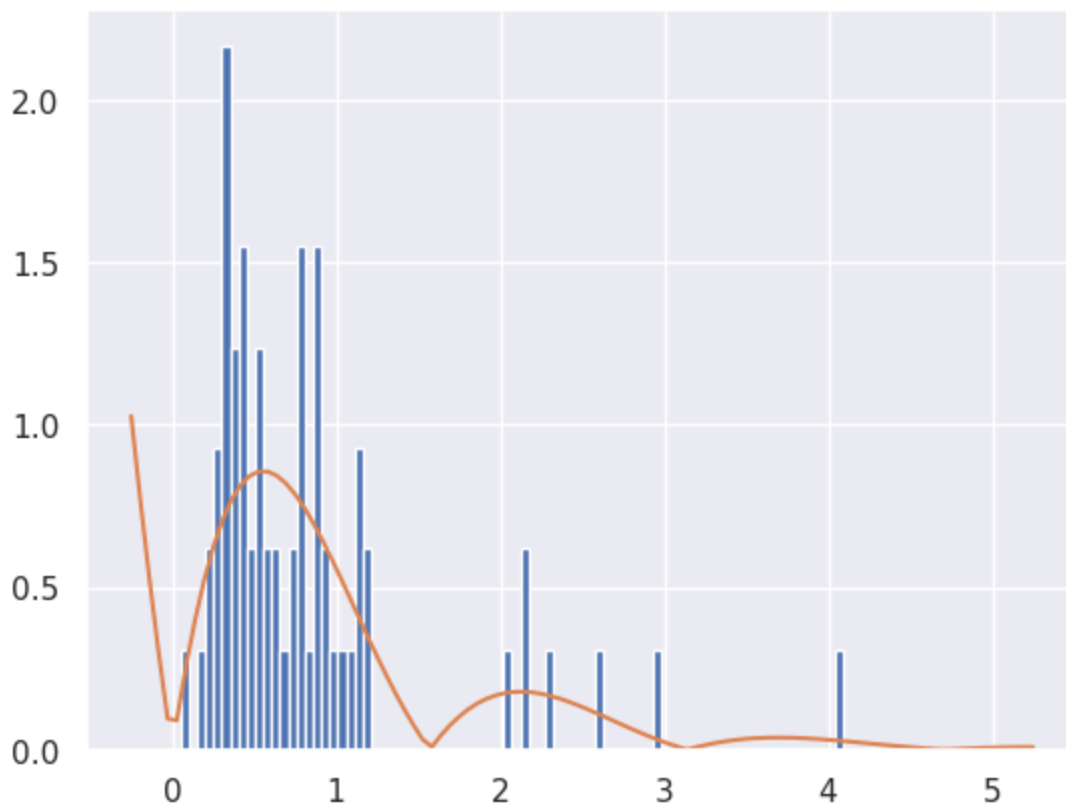
    accept_probs = decaying_target_distribution(proposals) / sampling_distribution_expon
    accept = np.random.uniform(0, 1, num_samples) < accept_probs
    num_accept = np.sum(accept)
    print('Accepted %d out of %d proposals' % (num_accept, num_samples))
    return proposals[accept]

samples = rejection_sample_exponential(num_samples=100)
plt.hist(samples, bins=np.linspace(0, 5, 100), density=True)
# Find how far the axis goes and draw the unnormalized distribution over it

tmin, tmax, _, _ = plt.axis()
t_inf = np.linspace(tmin, tmax, 100)

# Where did this magic number 0.6 come from? What happens if you change it?
plt.plot(t_inf, decaying_target_distribution(t_inf) / 0.6)
plt.show()
```

Accepted 65 out of 100 proposals



Implementing models in PyMC3

Let's go back to the simple review model:

$$x_i \sim \text{Bernoulli}(\theta) \quad (21)$$

$$\theta \sim \text{Beta}(\alpha, \beta) \quad (22)$$

Is there a way to implement this model computationally and have Python do all the work of inference for us? It turns out the answer is yes!

```
In [14]: reviews_a = np.array([1, 1, 1])
reviews_b = np.array([1] * 19 + [0])
```

```
In [15]: import pymc3 as pm
import arviz as az

# Parameters of the prior
alpha = 1
beta = 5

with pm.Model() as model:
    # Define a Beta-distributed random variable called theta
    theta = pm.Beta('theta', alpha=alpha, beta=beta)

    # Defines a Bernoulli RV called x. Since x is observed, we
    # pass in the observed= argument to provide our data
    x = pm.Bernoulli('x', p=theta, observed=reviews_b)

    # This line asks PyMC3 to approximate the posterior.
    # Don't worry too much about how it works for now.
    trace = pm.sample(2000, chains=2, tune=1000, return_inferencedata=True)
```

```
Auto-assigning NUTS sampler...
Initializing NUTS using jitter+adapt_diag...
Multiprocess sampling (2 chains in 4 jobs)
NUTS: [theta]
```

```
100.00% [6000/6000 00:02<00:00 Sampling 2 chains, 0
divergences]
```

```
Sampling 2 chains for 1_000 tune and 2_000 draw iterations (2_000 + 4_000 draws total) t
ook 3 seconds.
```

```
In [16]: trace
```

```
Out[16]: arviz.InferenceData
```




- posterior
- log_likelihood
- sample_stats
- observed_data

```
In [17]: trace.posterior
```

```
Out[17]: xarray.Dataset
```

► Dimensions: (chain: 2, draw: 2000)

▼ Coordinates:

chain	(chain)	int64	0 1	 
draw	(draw)	int64	0 1 2 3 4 ... 1996 1997 1998 1999	 

▼ Data variables:

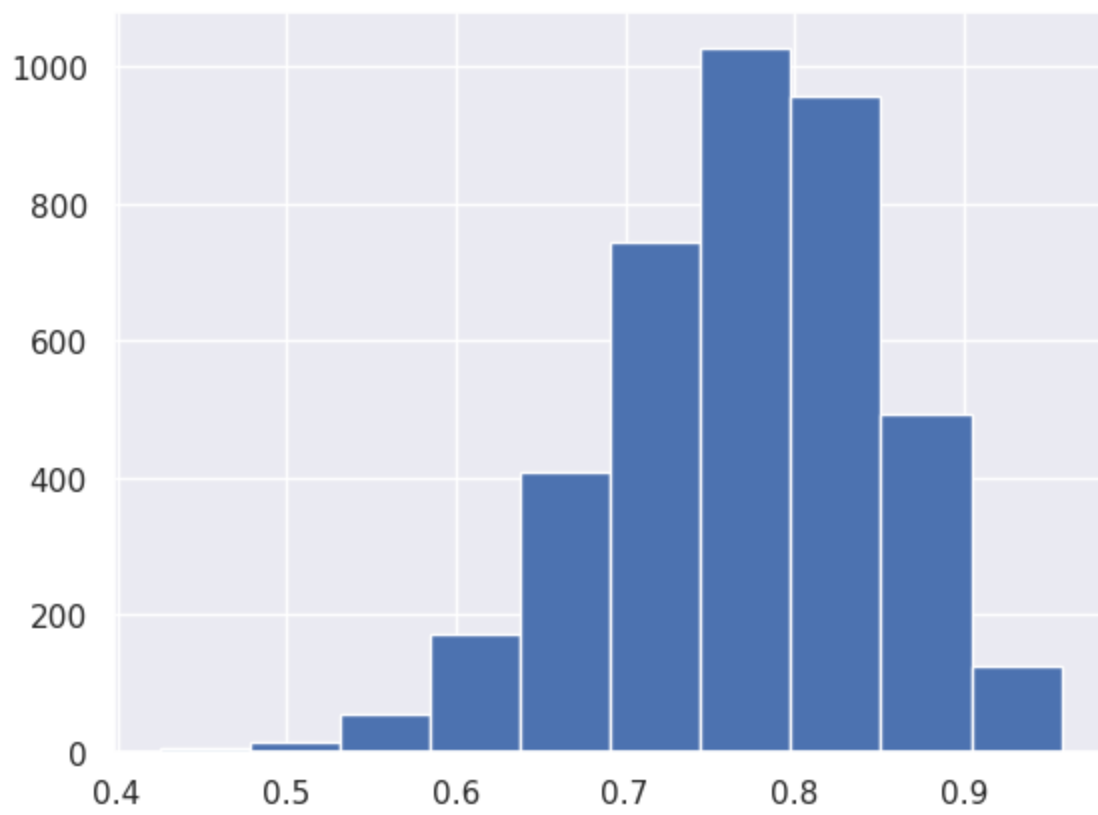
theta	(chain, draw)	float64	0.7928 0.8196 ... 0.825 0.6199	 
-------	---------------	---------	--------------------------------	---

▼ Attributes:

```
created_at : 2022-10-04T18:50:47.331016
arviz_version : 0.12.1
inference_librar... pymc3
inference_librar... 3.11.2
sampling_time : 3.1248276233673096
tuning_steps : 1000
```

```
In [18]: plt.hist(trace.posterior['theta'].values.flatten())
```

```
Out[18]: (array([  5.,  15.,  56., 171., 407., 743., 1028., 957., 494.,
124.]),
array([0.42490487, 0.47815495, 0.53140503, 0.58465511, 0.63790519,
0.69115527, 0.74440535, 0.79765543, 0.85090551, 0.90415559,
0.95740567])),
<BarContainer object of 10 artists>)
```



First, we'll need a trick called "fancy indexing". Here's how it works: