Mathematical Distributions

January 14, 2021

1 Lab 6 - Mathematical Distributions - Solution

[1]:
```
% matplotlib inline
```

This will make all the `matplotlib` images appear in the notebook.

[2]:
```
import numpy as np
import random as py_random
import numpy.random as np_random
import time
import seaborn as sns
import matplotlib.pyplot as plt

sns.set(style="whitegrid")
```

1.1 Generating Samples from Probability Distributions

Doctors work on cadavers and other professionals learn on similar, if less gruesome, models. For data scientists, we have at our disposal an infinite amount of synthetic data. The next questions introduce you to this technique in the form of random numbers and mathematical distributions.

1.1.1 Reproducible Random Numbers

Before you begin working with random numbers in any situation, in Data Science, as opposed to Machine Learning, it is desirable to set the random seed and record it. We do this for several reasons:

1. For reproducible research, we need to record the random seed that was used to generate our results so they will be regenerated exactly the same.
2. For sharing with others, if our text said there was some result, and the user re-runs the notebook, we want to get the same results.
3. If we are creating a model, and we accidentally generate the best model ever, we want to be able to build it again.

Although Python has *some* random number generators, we will be using `NumPy`'s random number generators throughout the course because it has a broader range of distributions.

`np.random.seed(N) # Numpy library`

You have several options for setting the seed:

- Just come up with a number, some integer, like: 27192759.
- Run:

```
int( time.time())
```

to print out a value you can use in either case. Do not just feed `int( time.time())` into the seed function. The whole point is to make the seed a constant. Numpy has ample documentation on its random module.

Here's an example:

[3]:
```
int( time.time())
```

[3]:
```
1538259736
```

[4]:
```
np.random.seed([1482004723]) # note the use of a number iside a list.
```

Note that the two above don't match. They did the very first time I ran the notebook but they don't now because I've had to re-run the notebook several times. This is, in fact, the point. In fact, in general, once you execute `int( time.time())` to get your seed, you can just delete the cell or just make up a seed. I like to use my birthday: YYYYMMDD and variations of other dates (don't use the same seed for *everything*).

[5]:
```
np.random.rand(10)
# do some stuff
```

[5]:
```
array([0.37677145, 0.43518392, 0.71651458, 0.54653675, 0.98604431,
       0.45821284, 0.27999283, 0.03121421, 0.14613549, 0.10168693])
```

[6]:
```
np.random.rand( 10)
# do more stuff
```

[6]:
```
array([0.38032926, 0.60617514, 0.63969352, 0.52414294, 0.53436926,
       0.05148294, 0.75952124, 0.44076097, 0.76423589, 0.52040212])
```

[7]:
```
np.random.seed([1482004723])
```

[8]:
```
np.random.rand(10)
# do some stuff
```

[8]:
```
array([0.37677145, 0.43518392, 0.71651458, 0.54653675, 0.98604431,
       0.45821284, 0.27999283, 0.03121421, 0.14613549, 0.10168693])
```

There are a few things to note here:

1. We asked what time it was to get the seed but we did not put it directionly into `np.random.seed()`.
2. The argument to `np.random.seed()` must be a List.

3. We set the seed then got 10 random numbers by calling `np.random.rand(10)`. In practice, this might just be all we want to do (get numbers from the distribution) or we may want to do more calculations.
4. We get 10 more random numbers by calling `np.random.rand( 10)`. Notice that these are not the same as the first call. You can think of `random` *in general* as generating a stream of random numbers according to some distribution which we just tap into. We get the first 10, the next 10, the next 25, etc.
5. We set the random seed to the same random seed as before.
6. We got the same first 10 random numbers. This demonstrates that setting the seed "resets" the stream of random numbers. This is what we want.

In general, before answering each question, we are going to want to set the random seed to some value. Do not do it inside a function that is getting called over and over again, set it at the start of the experiment.

We will talk a lot more about visualization later but for right now I'm going to introduce the *histogram*. A histogram is a means for visualizing the distribution of a variable. There are several variants and the libraries are sometimes confusing on this score. Absent any directions to the contrary, the histogram will calculate the absolute counts of the data. The usual alternative, at least with continuous variables, is to set `normed=True` and you will get a *density*. It is also possible through weigting to get a `mass` or *relative frequency* for a discrete variable.

1.2 Uniform Distribution

Consider the following problem. I want to generate 100 data points on the range (-5.0, 10.0) that are from a *uniform distribution*. How do I do this?

1. I set the random seed.
2. I look through the documentation to see if there is a function that will generate the data directly or via a *transformation*.
3. I then visualize the data I generated.

Let's do that:

[9]:
```
int( time.time())
```

[9]:
```
1538259736
```

[10]:
```
np.random.seed([1482003424]) # this will be different
```

Looking at the documentation, there is a function `uniform` that takes *low*, *high* and *size* arguments. Let's do it that way first. I'm going to arbitrarily look at the first 20 values just to see what I got:

[11]:
```
xs = np.random.uniform(-5.0, 10.0, 100)
print( xs[0:20])
```

```
[ 6.42438573 -3.22910746 -1.51821304 -3.87133369 -0.81487703  6.3629005
  8.50156041  4.86376343 -2.84230612  2.52226997  9.06166161  1.64070685
 -2.2239334   5.66352015  2.11413729 -3.75793272  3.8597046   5.83393762
 -3.78916267 -3.62696669]
```

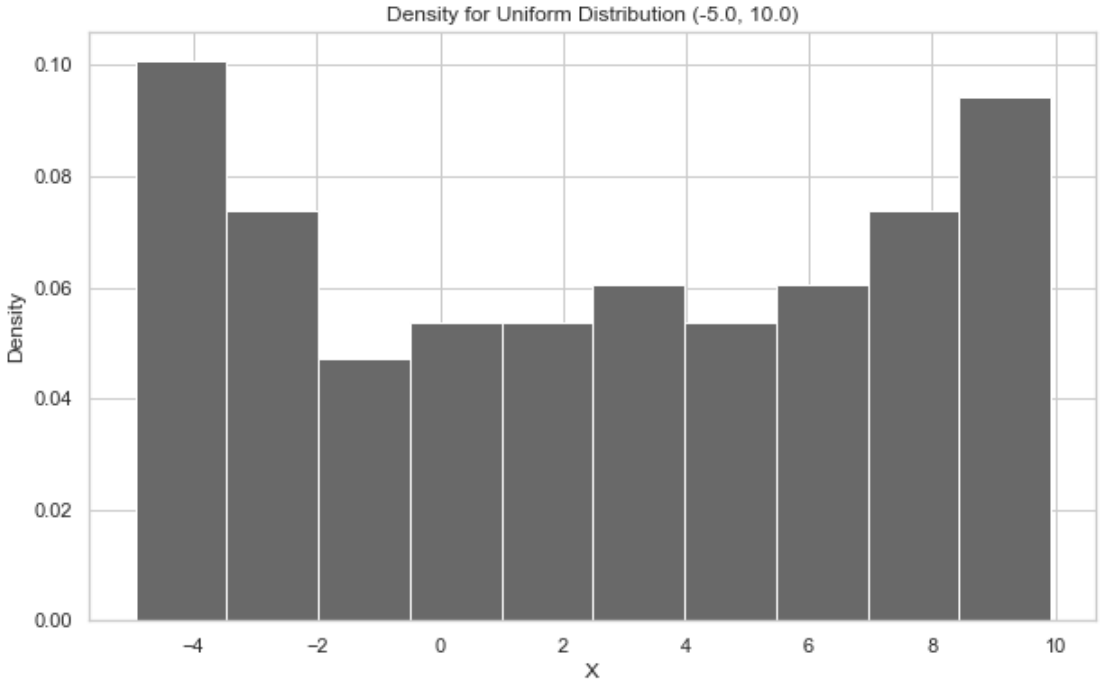
Then I'm going to plot the data.

[12]:
```
figure = plt.figure(figsize=(10, 6)) # first element is width, second is height.

axes = figure.add_subplot(1, 1, 1)

axes.hist( xs, density=True, color="DimGray") # a density
axes.set_ylabel( "Density")
axes.set_xlabel( "X")
axes.set_title( "Density for Uniform Distribution (-5.0, 10.0)")

plt.show()
```



I'm also going to explore the data a little bit. I used the parameters *low* and *high* to generate the data. What values did I get back?

[13]:
```
print( "low =", min( xs))
print( "high=", max( xs))
```

```
low = -4.948704934136167
high= 9.924238206597877
```

Treating the synthetic data as an *empirical distribution*, I can also calculate the moments of the data. However, there aren't `m1` or `m2` functions. Instead, I'm going to use the identity between `m1` and the *arithmetic mean* and `m2` and the variance but I am not going to commit any particular Mathematical distribution as a model for this data even though I know that I used a Uniform

distribution to generate it.

This last point can be a bit confusing. I would say all software packages and libraries will calculate the mean and variance but this is actually conflating two steps: calculating the first two moments and using those moments to parameterize a Mathematical distribution as a model (mean and variance). This is mostly because of the Central Limit Theorem which indicates that we are going to run into the Normal distribution a lot of the time but it is still combining *two steps*. As data scientists, we want to pull these two steps apart and make our own judgements.

[14]:
```
print( "m1=", np.mean(xs))
print( "m2=", np.var( xs))
```

```
m1= 2.55922405435618
m2= 22.368135384226754
```

Now, what if `uniform` hadn't existed? I would have had to have used `random` which generates uniformly distributed random numbers on the range (0, 1) and projected it into the range (-5.0, 10.0):

[15]:
```
np.random.seed([1482003424])
xs = np.random.random(100) * 15.0 - 5.0
print( xs[0:20])
print( "low =", min( xs))
print( "high=", max( xs))
```

```
[ 6.42438573 -3.22910746 -1.51821304 -3.87133369 -0.81487703  6.3629005
  8.50156041  4.86376343 -2.84230612  2.52226997  9.06166161  1.64070685
 -2.2239334   5.66352015  2.11413729 -3.75793272  3.8597046   5.83393762
 -3.78916267 -3.62696669]
low = -4.948704934136167
high= 9.924238206597877
```

Note that this is a *new* experiment so I set the random seed. However, I specifically wanted to see if the two different methods generated the same random numbers (they do) so I set it the same random seed. In general, each experiment will have a different random seed.

1.2.1 Exercises.

1. A coin has a probability of heads, $\theta = 0.67$. Simulate 25 events (coin tosses) from this Bernouilli distribution (25 Bernoulli *Trials*).

1. Set the random seed.
2. Generate the samples, `x`. (There may be multiple ways to do this).
3. Calculate the first moment using `np.mean( x)` to get the estimate of p (it's a Python trick). How close are you?

** Step 1. ** Set the seed

[16]:
```
np.random.seed([125]) # made this up.
```

** Step 2.** Calculate the data. We can use θ as a threshold value against a uniformly distributed

variable on the range (0, 1) to simulate a Bernoulli trial. We indicate 1 for success and 0 for failure. This is a "roll your own" method using a *List Comprehension*.

[17]:
```
theta = 0.67
xs = [1 if np.random.random() < theta else 0 for _ in range( 25)]
```

Step 3. We can use `np.mean` over the 0's and 1's to get m_1 which is then the Method of Moments estimator for θ. We can compare this to the θ we started with:

[18]:
```
m1 = np.mean( xs)
print( "m1 = theta_hat =", m1)
print( "theta_hat/theta =", m1/theta)
```

```
m1 = theta_hat = 0.72
theta_hat/theta = 1.0746268656716418
```

Not very close (0.72/0.67) is more than 7% higher than it "really" was. We'll see in the next Module what this means and how to bound our estimates.

** Working with the Normal Distribution.**

2. $\mu = 32.5$ **and** $\sigma = 0.325$

1. Set the random seed.
2. Find the function for the normal distribution in the NumPy documentation.
3. Generate **25** samples for x from a normal distribution with $\mu = 32.5$ and $\sigma = 0.325$ (1%).
4. Plot a histogram of the data (change the labels!)
5. Calculate the first moment of x.
6. Using the Method of Moments, estimate the mean from the first moment. How far off is your estimate in percent?

Step 1. Set the random seed:

[19]:
```
np.random.seed([12873])
```

Step 2. Looking in the documentation for NumPy, the function that creates random numbers from the normal distribution is:

`normal([loc, scale, size])`

where loc = mu, scale = standard deviation.

Step 3. Let's make a function to translate mu and v into s (standard deviation) and then use the function to create 25 samples from a normal distribution with the specified parameters. We're going to print out 20 just to get a sense of the data:

[20]:
```
mu = 32.5
s = 0.325
xs = np.random.normal( mu, s, 25)
print(xs[0:20])
```

```
[32.59396246 32.55047946 32.74616382 32.10882515 31.89728886 32.4131988
 32.22917489 32.37756642 32.04337818 32.81159913 33.21456123 32.96110348
```

```
32.62581775 32.97797022 32.71794508 32.53077859 32.58686424 32.28912607
32.54937358 32.18383262]
```

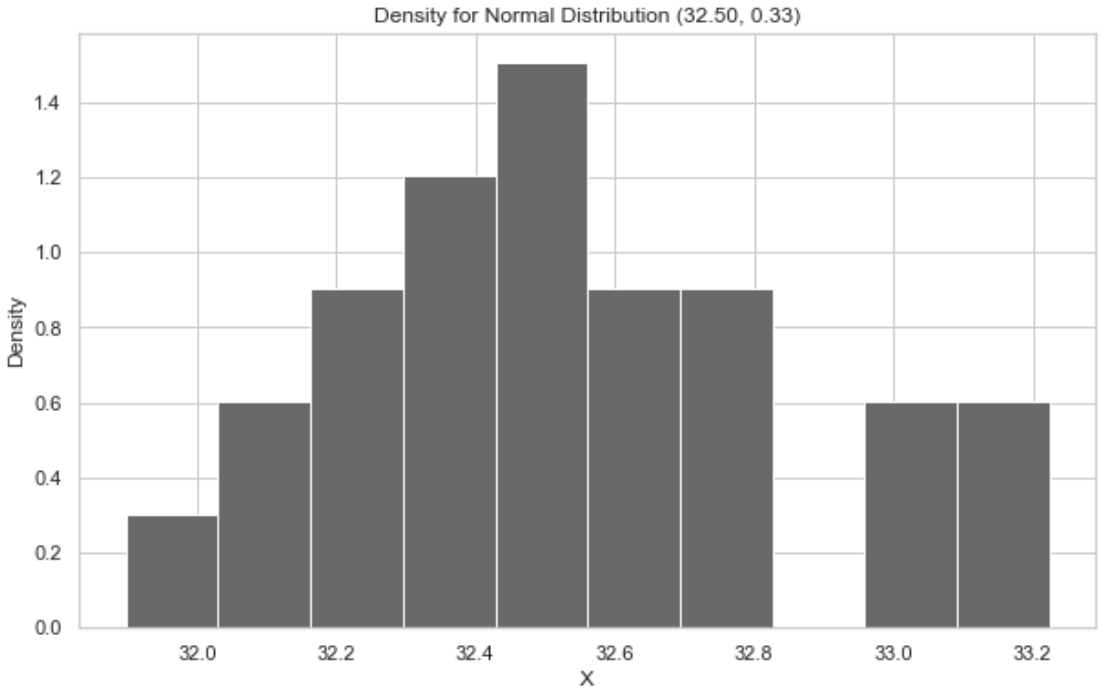
Step 4. Plot the data we just generated.

[21]:
```
figure = plt.figure(figsize=(10, 6)) # first element is width, second is height.

axes = figure.add_subplot(1, 1, 1)

axes.hist( xs, density=True, color="DimGray")
axes.set_ylabel( "Density")
axes.set_xlabel( "X")
axes.set_title( "Density for Normal Distribution (%0.2f, %0.2f)" % ( mu, s))

plt.show()
plt.close()
```



It looks normally distributed, kind of.

Step 5. Print out the first moment using `np.mean(xs)`:

[22]:
```
m1 = np.mean( xs)
print( "m1 =", m1)
```

```
m1 = 32.52947120021099
```

Step 6. Calculate the discrepancy. In this case, m_1 is the estimate of the mean:

[23]:
```
print( "discrepancy =", m1/mu)
```

```
discrepancy = 1.0009068061603381
```

The discrepancy isn't very much with the amount of variability we used (1% of the mean).

Now, one of the problems with this experiment is that we only run it once. We might run it and the mean will only be 1% off and we might run it again (with a different seed) and the results are 10% off.

3. Rerunning experiments

1. Set the random seed.
2. Write a function that will generate m samples of x from the Normal distribution, n times. This means the function will return a List of Lists. The outer List will have length n and the inner Lists will all have length m. Use the μ and σ from the previous exercise. Set $m = 25$ and $n = 100$ (you'll then have 100 data sets, each with 25 data points).
3. Calculate the first moment of each of the n data sets. You'll have 100 of these.
4. Plot a histogram of the data.
5. Calculate the low, high and first moment of the data and discuss.

Step 1. Set the random seed:

[24]:
```
np.random.seed([3841765259])
```

Step 2. Write a function that will run the previous experiment multiple times. mu is the mean, v is the coefficient of variation, m is the number of samples, n is the number of trials.

[25]:
```
def repeat_random_normal( mu, s, m, n,):
    result = []
    for i in range( n):
        xs = np.random.normal( mu, s, m)
        m1 = np.mean( xs)
        result.append( m1)
    return result
```

[26]:
```
mu = 32.5
s = 0.325
m = 25
n = 100

xs = repeat_random_normal( mu, s, 25, 100)
```

Notice that our xs are themselves calculations of first moments! We can apply these techniques to our estimates and just analyze them as data.

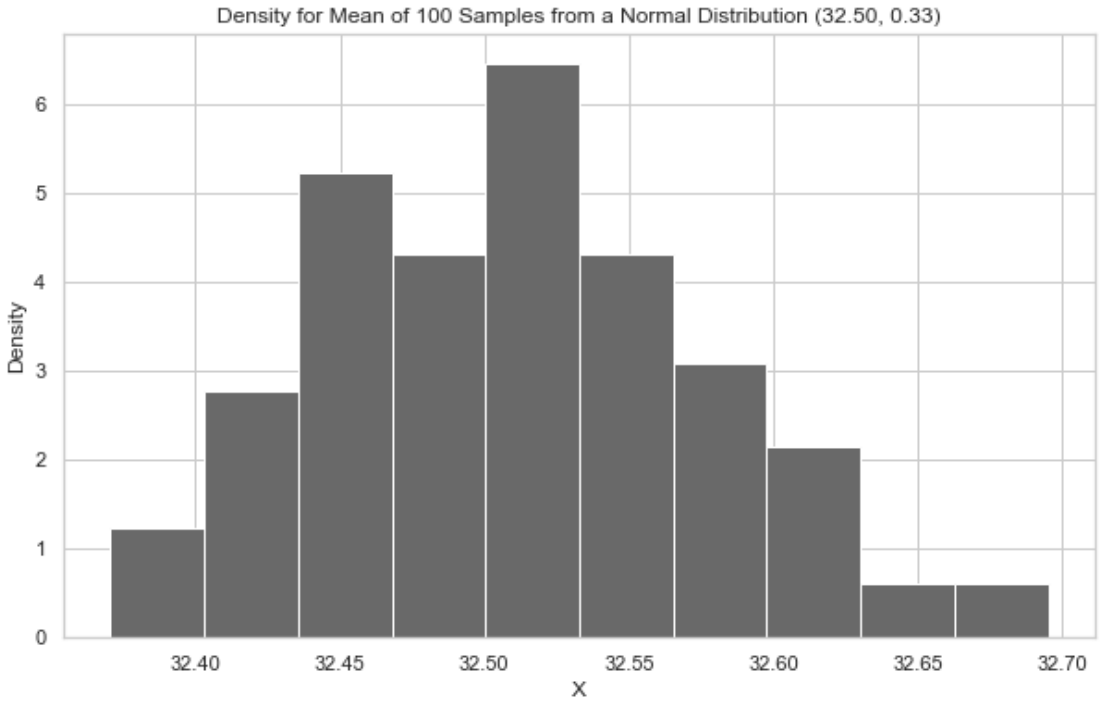
Step 3. Plot a histogram of the data:

[27]:
```
figure = plt.figure(figsize=(10, 6)) # first element is width, second is height.

axes = figure.add_subplot(1, 1, 1)
```

```
axes.hist( xs, density=True, color="DimGray")
axes.set_ylabel( "Density")
axes.set_xlabel( "X")
axes.set_title( "Density for Mean of %d Samples from a Normal Distribution (%0.
↪2f, %0.2f)" % ( n, mu, s))

plt.show()
plt.close()
```



Step 4. Calculate the low, high and mean of the data and discuss.

[28]:
```
print( "low =", np.min( xs))
print( "m1 =", np.mean( xs))
print( "high =", np.max( xs))
```

```
low = 32.37054290162743
m1 = 32.50949188252682
high = 32.69508466045785
```

Again we can see that the range of means calculated from the data match the actual mean (mu) very well. This depends mostly on the fact that the variability (dispersion) is so low.

1.2.2 Other Distributions

As we saw in the chapter "Mathematical Distributions", while the Normal distribution is parameterized by the mean and variance (standard deviation), other distributions are not. We would like to see what data from other distributions looks like whose first moment is about 32.5 (and whose second moment is about 0.325, if appropriate).

We can use the Method of Moments to move from our value of μ from Exercise 2 to the first moment m_1 and then the appropriate formulas to move to the parameters of whatever Mathematical distribution we want. Similarly, we can move from σ^2 to m_2 and then use that as the second moment if needed.

The Method of Moments formulas for various distributions are provided in the text. If you should need formulas for other distributions, you can either search for them or derive them yourself.

4. Exponentional Distribution

Based on this discussion, use $\mu = 32.5$ and $\sigma = 0.325$ to generate 25 samples from the Exponential distribution and repeat the same steps we did for the Normal distribution. Remember, you need to convert these into m_1 and m_2 and then m_1 and m_2 (if needed) into the parameter(s) you need.

Step 1. Set random seed:

[29]:
```
np.random.seed([13579])
```

** Step 2.** Find the function in the documentation:

```
exponential(scale=1.0, size=None)
```

We know from the text that the Exponential distribution is "officially" parameterized by λ, the rate. These kinds of mismatches often happen. Because $scale = \beta = \frac{1}{\lambda}$, we can use $\mu \rightarrow m_1 \rightarrow scale$ directly. m_2 is not used.

(Remember that if β, the scale, is 2 calls per hour then λ, the rate, is 30 minutes between calls.)

Step 3. Generate samples.

[30]:
```
mu = 32.5
xs = np.random.exponential( mu, 25)
```

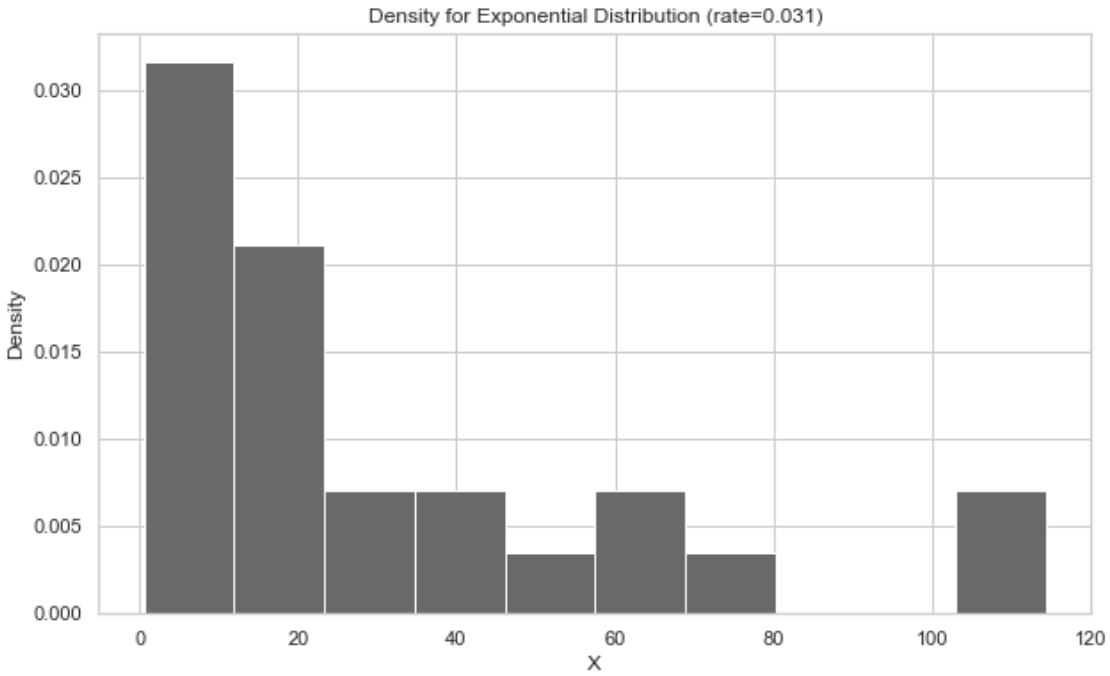
** Step 4.** Plot the data. Even though we used μ, we're going to put λ in the title:

[31]:
```
figure = plt.figure(figsize=(10, 6)) # first element is width, second is height.

axes = figure.add_subplot(1, 1, 1)

axes.hist( xs, density=True, color="DimGray")
axes.set_ylabel( "Density")
axes.set_xlabel( "X")
axes.set_title( "Density for Exponential Distribution (rate=%0.3f)" % (1/mu))

plt.show()
plt.close()
```



** Step 5.** Print out the mean (the first moment) although we're really interested in the rate parameter, λ:

[32]:
```
m1 = np.mean( xs)
rate = 1/m1

print( "m1 =", m1)
print( "rate =", rate)
```

```
m1 = 30.46480082943654
rate = 0.032824767363446944
```

Step 6. How far off is the mean you calculate from the μ you started off with (in percent)?

[33]:
```
print( "discrepancy =", m1/mu)
```

```
discrepancy = 0.9373784870595859
```

The discrepancy is about 10% which is quite a bit bigger than when we used the same number to parameterize a normal distribution.

5. Gamma Distribution

Based on the discussion above, use $\mu = 32.5$ and $\sigma = 0.325$ to generate 25 samples from the Gamma distribution and repeat the same steps we did for the Normal distribution. Remember, you need to convert these into m_1 and m_2 and then m_1 and m_2 (if needed) into the parameter(s) you need.

Step 1. Set random seed.

[34]:
```
np.random.seed([683920])
```

** Step 2.** Find the function in the documentation.

The function from the documentation is here:

`gamma(shape, scale=1.0, size=None)¶`

I found the formula for the Method of Moments estimators here. They are:

$$\gamma = (\frac{\mu}{\sigma})^2$$

$$\beta = \frac{\sigma^2}{\mu}$$

The description for the Gamma Distribution defines the parameters γ as the shape parameter and β as the scale parameter.

Step 3. Generate samples.

[35]:
```
mu = 32.5
s = 0.325
gamma = (mu/s)**2
beta = (s**2/mu)

xs = np.random.gamma( gamma, beta, 25)
print( xs[0:20])
```

```
[32.09316514 32.54489771 32.3690439  32.26440996 32.97763059 32.44208009
 32.92362969 32.39748714 32.80287017 33.1303675  32.56584326 32.36231413
 32.52895262 33.17594745 32.26200196 32.33554741 32.31893531 32.15532476
 32.69844167 32.19313578]
```

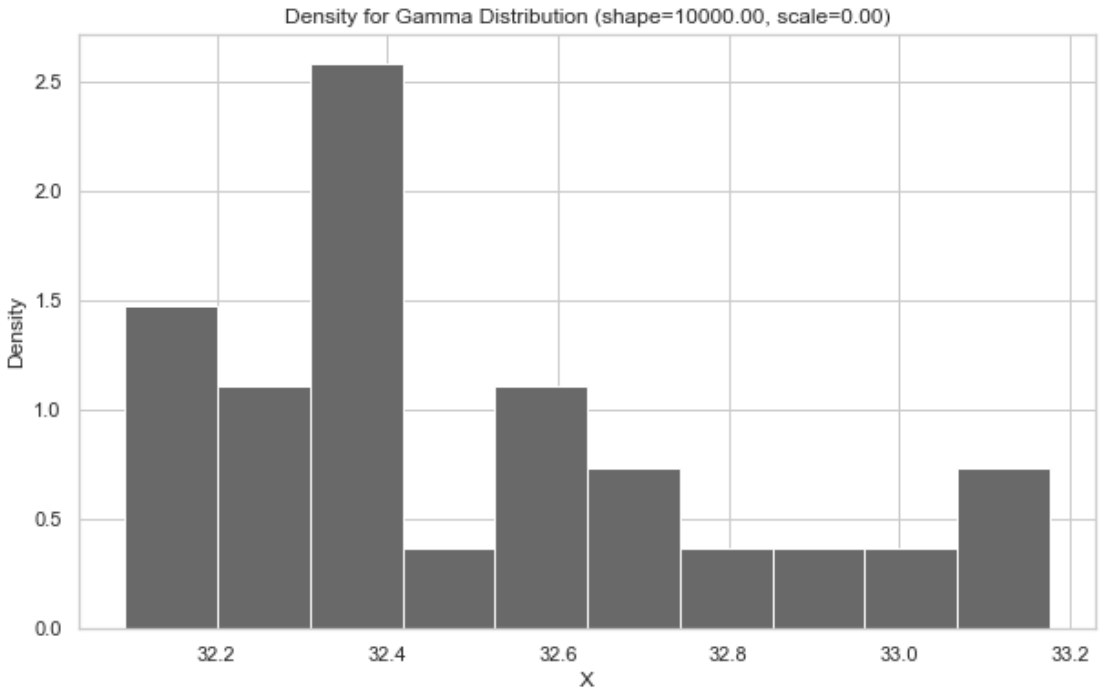
** Step 4.** Plot the data.

[36]:
```
figure = plt.figure(figsize=(10, 6)) # first element is width, second is height.

axes = figure.add_subplot(1, 1, 1)

axes.hist( xs, density=True, color="DimGray")
axes.set_ylabel( "Density")
axes.set_xlabel( "X")
axes.set_title( "Density for Gamma Distribution (shape=%0.2f, scale=%0.2f)" %␣
 ↪(gamma, beta))

plt.show()
plt.close()
```



** Step 5.** Print out the mean.

[37]:
```
m1 = np.mean( xs)
m2 = np.var( xs)

print("m1 =", m1)
print("m2 =", m2)

gamma_est = (m1**2)/m2
beta_est = m2/m1

print( "gamma =", gamma_est)
print( "beta =", beta_est)
```

```
m1 = 32.493214095942214
m2 = 0.0906100904367469
gamma = 11652.222806485079
beta = 0.0027885850309915137
```

Step 6. How far off is the mean you calculate from the μ you started off with (in percent)?

[38]:
```
print( "gamma discrepancy (est/actual) =", gamma_est/gamma)
print( "beta discrepancy (est/actual) =", beta_est/beta)
```

```
gamma discrepancy (est/actual) = 1.165222280648508
beta discrepancy (est/actual) = 0.8580261633820041
```

We're off by quite a bit. The gamma estimate is almost 17% larger and the beta is almost 15% smaller.

6. From a Systems/Complexity Theory perspective, how might we interpret the variability of a factor like x? What might it mean if the variability is low or high? (Why doesn't x just have one value...why does it vary at all?)

From Systems Theory (especially Causal Loop Diagrams), we know that we can build pretty complicated models of how the variables in our domain interact both positively and negatively and how they may balance or reinforce each other. We concede that most of the time, we simply cannot get all the relevant data for the domain we want to understand in order to solve whatever our problem is (remember the "CoNVO").

Now, each observation of x may either be an observation of the *single* system at different *times* (like looking at the economy each month) or observations of many copies of the same system at the same time (like looking at a bunch of people). In either case, the system will be in different states. In the first case because maybe the economy grew, there was a technological innovation (if we modeled it), or there was a natural disaster. In the second case, because all the different copies of the system have different histories and environments. Not to harp on height, but each person may be a different height because of genetics, gene expression, and access to food and types of food throughout their lifetimes so far, and then their age (because we–hopefully–grow as we grow up and–unfortunately–shrink when we age).

Now the *dispersion* may be due to a wide number of factors. We are trying to model a complex system with a single distribution. There may be many factors with small effects that cause the dispersion to be large or there may be a few factors with large effects that cause the dispersion to be large.

Sometimes this is all we can do (a simple Statistical Model) but sometimes we can build better models (which we'll do later in the semester).