CS109A Introduction to Data Science

Homework 0

Harvard University
Summer 2018
Instructors: Pavlos Protopapas and Kevin Rader

This is a homework which you must turn in.

This homework has the following intentions:

1. To get you familiar with the jupyter/python environment.
2. You should easily understand these questions and what is being asked. If you struggle, this may not be the right class for you.
3. You should be able to understand the intent (if not the exact syntax) of the code and be able to look up on Google and provide code that is asked of you. If you cannot, this may not be the right class for you.

```
In [1]:  ## RUN THIS CELL TO GET THE RIGHT FORMATTING
         from IPython.core.display import HTML
         def css_styling():
             styles = open("C:/Users/werkh/OneDrive/Documents/CS109_notebooks/HW0/cs109.cs
             return HTML(styles)
         css_styling()
```

Out[1]:

Basic Math and Probability/Statistics Calculations

We'll start you off with some basic math and statistics problems questions to make sure you have the appropriate background to be comfortable with concepts that will come up in CS 109a.

Question 1: Mathiness is What Brings Us Together Today

Matrix Operations

Complete the following matrix operations. ***Note: Do not do this numerically and show your work as a markdown/latex*** *notebook cell*

1.1. Let $A = \begin{pmatrix} 3 & 4 & 2 \\ 5 & 6 & 4 \\ 4 & 3 & 4 \end{pmatrix}$ and $B = \begin{pmatrix} 1 & 4 & 2 \\ 1 & 9 & 3 \\ 2 & 3 & 3 \end{pmatrix}$.

Compute $A \cdot B$.

1.2. Let $A = \begin{pmatrix} 0 & 12 & 8 \\ 1 & 15 & 0 \\ 0 & 6 & 3 \end{pmatrix}$.

Compute A^{-1}.

Calculus and Probability

Complete the following (show your work as a markdown/latex notebook cell)

1.3. From Wikipedia:

> In mathematical optimization, statistics, econometrics, decision theory, machine learning and computational neuroscience, a loss function or cost function is a function that maps an event or values of one or more variables onto a real number intuitively representing some "cost" associated with the event. An optimization problem seeks to minimize a loss function.

We've generated a cost function on parameters $x, y \in \mathcal{R}$ $L(x, y) = 3x^2y - y^3 - 3x^2 - 3y^2 + 2$. Find the critical points (optima) of $L(x, y)$.

1.4. A central aspect of call center operations is the per minute statistics of caller demographics. Because of the massive call volumes call centers achieve, these per minute statistics can often take on well-known distributions. In the CS109 Homework Helpdesk, X and Y are discrete random variables with X measuring the number of female callers per minute and Y the total number of callers per minute. We've determined historically the joint pmf of (X, Y) and found it to be

$$p_{X,Y}(x, y) = e^{-4}\frac{2^y}{x!(y-x)!}$$

where $y \in \mathbb{N}, x \in [0, y]$ (That is to say the total number of callers in a minute is a non-negative integer and the number of female callers naturally assumes a value between 0 and the total number of callers inclusive). Find the mean and variance of the marginal distribution of X.

Hints:

1. $x \in [0, y] \Rightarrow x < y$.
2. You may find the change of variable $z = y - x$ helpful.
3. Recall:

$$\sum_{z=0}^{\infty} \frac{2^z}{z!} = e^2$$

Basic Statistics

Complete the following: you can perform the calculations by hand (show your work) or using software (include the code and output...screenshots are fine if it is from another platform).

1.5. 37 of the 76 female CS concentrators have taken Data Science 1 (DS1) while 50 of the 133 male concentrators haven taken DS1. Perform a statistical test to determine if interest in Data Science (by taking DS1) ios related to sex. Be sure to state your conclusion.

Answers

1.1

Let $C = A \cdot B$

To compute C, we compute the dot product for each row of A with each column of B. For example, the first value of C:

$$C_{1,1} = x \cdot y \quad \text{where} \quad x = \begin{bmatrix} a_{1,1} \\ a_{1,2} \\ a_{1,3} \end{bmatrix} \text{ and } y = \begin{bmatrix} b_{1,1} & b_{2,1} & b_{3,1} \end{bmatrix}$$

$$x \cdot y = a_{1,1} b_{1,1} + a_{1,2} b_{2,1} + a_{1,3} b_{3,1} = 3 \times 1 + 4 \times 1 + 2 \times 2 = 11$$

Repeating the process of calculating the dot product of each row of A with each column of B yields the resulting 3x3 matrix

$$C = \begin{bmatrix} 11 & 54 & 24 \\ 19 & 86 & 40 \\ 15 & 55 & 29 \end{bmatrix}$$

We can also calculate the result programmatically with the following python code

```python
import numpy as np
A = np.array([[3, 4, 2], [5, 6, 4], [4, 3, 4]]);
B = np.array([[1, 4,  2],[1, 9, 3],[2, 3, 3]]);
C = np.matmul(A,B);
```

1.2

Let $B = A^{-1}$

We can calculate the inverse of A as

$$B = \frac{1}{determinant(A)} adjugate(a)$$

or we can reduce a matrix $[AI]$ to the form $[IB]$ where I is the identity matrix

$$[AI] = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & 1 & 0 & 0 \\ a_{2,1} & a_{2,2} & a_{2,3} & 0 & 1 & 0 \\ a_{3,1} & a_{3,2} & a_{3,3} & 0 & 0 & 1 \end{bmatrix}$$

substituting in the values of A, and switching R_2 and R_1 we get

$$[AI] = \begin{bmatrix} 1 & 15 & 0 & 0 & 1 & 0 \\ 0 & 12 & 8 & 1 & 0 & 0 \\ 0 & 6 & 3 & 0 & 0 & 1 \end{bmatrix}$$

$\frac{1}{12}R_2 \rightarrow$

$$\begin{bmatrix} 1 & 15 & 0 & 0 & 1 & 0 \\ 0 & 1 & \frac{8}{12} & \frac{1}{12} & 0 & 0 \\ 0 & 6 & 3 & 0 & 0 & 1 \end{bmatrix}$$

$-1R_3 - 6R_2 \rightarrow$

$$\begin{bmatrix} 1 & 15 & 0 & 0 & 1 & 0 \\ 0 & 1 & \frac{8}{12} & \frac{1}{12} & 0 & 0 \\ 0 & 0 & 1 & \frac{1}{2} & 0 & -1 \end{bmatrix}$$

$R_2 + \frac{-8}{12}R_3 \rightarrow$

$$\begin{bmatrix} 1 & 15 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & \frac{-3}{12} & 0 & \frac{8}{12} \\ 0 & 0 & 1 & \frac{1}{2} & 0 & -1 \end{bmatrix}$$

$R_1 - 15R_2 \rightarrow$

$$[IB] = \begin{bmatrix} 1 & 0 & 0 & \frac{45}{12} & 1 & -10 \\ 0 & 1 & 0 & \frac{-3}{12} & 0 & \frac{8}{12} \\ 0 & 0 & 1 & \frac{1}{2} & 0 & -1 \end{bmatrix}$$

with the left-hand side of the matrix equal to I, we can find A^{-1} on the right-hand side

$$B = \begin{bmatrix} \frac{45}{12} & 1 & -10 \\ \frac{-3}{12} & 0 & \frac{8}{12} \\ \frac{1}{2} & 0 & -1 \end{bmatrix}$$

We can also check out work with python

In [2]:
```python
import numpy as np
A = np.array([[0, 12, 8], [1, 15, 0], [0, 6, 3]]);
print(np.linalg.inv(A))
```

```
[[  3.75        1.         -10.        ]
 [ -0.25        0.          0.66666667]
 [  0.5        -0.         -1.        ]]
```

1.3

To find the critical points of $L(x, y)$, we have to find the points of the function where

$$\frac{\delta L}{\delta x} = 0 \quad \text{and} \quad \frac{\delta L}{\delta y} = 0$$

Let $L(x, y) = 3x^2 y - y^3 - 3x^2 - 3y^2 + 2$

$$\frac{\delta L}{\delta x} = 6xy - 6x = 6x(y - 1)$$

$$\frac{\delta L}{\delta x} = 0 \quad when \quad x = 0 \quad or \quad y = 1$$

Since we know that $\frac{\delta L}{\delta x} = 0$ when $x = 0$ or $y = 1$, we can evaluate $\frac{\delta L}{\delta y}$ at those values to simplify. Solving for $\frac{\delta L}{\delta y}$ we find

$$\frac{\delta L}{\delta y} = 3x^2 - 3y^2 - 6y$$

Evaluating $\frac{\delta L}{\delta y}$ at $x = 0$

$$\frac{\delta L}{\delta y} = 3(0)^2 - 3y^2 - 6y$$

$$\frac{\delta L}{\delta y} = -3y^2 - 6y$$

$$\frac{\delta L}{\delta y} = -3y(y + 2)$$

Therefor when $x = 0$

$$\frac{\delta L}{\delta y} = 0$$

$$at \quad y = 0 \quad and \quad y = -2$$

Evaluating $\frac{\delta L}{\delta y}$ at $y = 1$

$$\frac{\delta L}{\delta y} = 3x^2 - 3(1)^2 - 6(1)$$

$$\frac{\delta L}{\delta y} = 3x^2 - 9$$

$$\frac{\delta L}{\delta y} = -3(x^2 - 3)$$

Therefor when $y = 1$

$$\frac{\delta L}{\delta y} = 0$$

$$at \quad x = \pm\sqrt{3}$$

Therefor $L(x, y)$ has critical points at

x	y
0	0
0	-2
$\sqrt{3}$	1
$-\sqrt{3}$	1

1.4

Let $P_{xy}(x, y) = e^{-4} \frac{2^y}{x!(y-x)!}$

The marginal distribution of x is given by

$$P_x(x) = \sum_{y=0}^{\infty} P_{xy}(x, y)$$

We can simplify $P_{xy}(x, y)$ to $P_{xz}(x, z)$ substituting in the variable $z = y - x$

$$P_{xz}(x, z) = e^{-4} \frac{2^x 2^z}{x!z!}$$

Therefore

$$P_x(x) = \sum_{z=0}^{\infty} P_{xz}(x, y)$$

$$P_x(x) = \sum_{z=0}^{\infty} e^{-4} \frac{2^x 2^z}{x!z!}$$

$$P_x(x) = e^{-4} \frac{2^x}{x!} \sum_{z=0}^{\infty} \frac{2^z}{z!}$$

Recalling that

$$\sum_{z=0}^{\infty} \frac{2^z}{z!} = e^2$$

We can simplify $P_x(x)$

$$P_x(x) = e^{-4} \frac{2^x}{x!} e^2$$

$$P_x(x) = \frac{2^x e^{-2}}{x!}$$

Recognizing that looks the form of the poisson distribution

$$\frac{\lambda^k e^{-\lambda}}{k!}$$

and knowing that the mean and variance of the poisson distribution is the rate constant λ, the mean and variance of the marginal distribution of x is

$$\mu = 2, \quad \sigma^2 = 2$$

1.5

We can use a z-test to compare the null hypothesis $p_f = p_m$, to the alternative, $p_f \neq p_m$ where p_f is the probability of a female enrolling and p_m is the probability of a male enrolling.

The z test statistic is given by

$$z = \frac{\hat{p}_f - \hat{p}_m}{\sqrt{\hat{p}_j(1 - \hat{p}_j)(\frac{1}{n_f} + \frac{1}{n_m})}}$$

where the $\hat{p}_f$, $\hat{p}_m$, and $\hat{p}_j$ are the observed female, male, and joint probabilities respectively

$$\hat{p}_f = \frac{37}{76} = 0.487$$

$$\hat{p}_m = \frac{50}{133} = 0.376$$

$$\hat{p}_j = \frac{37 + 50}{76 + 133} = 0.416$$

using python to do the computation

```
In [3]:  import numpy as np
         pf = 0.487
         pm = 0.376
         pj = 0.416
         nf = 76
         nm = 133


         z = (pf-pm)/(np.sqrt(pj*(1-pj)*(1/nf + 1/nm)))
         print(z)
```

```
1.5661356065480148
```

This will be a two-tailed test since our probabilities could be either above or below the null hypothesis. We find the $z - score = 1.57$, which is well inside of $z = \pm 1.96$, the critical region values corresponding to a confidence level of 95%. Therefore we cannot conclude that the data shows a significant difference between male and female enrollment

---

```
In [4]:  ## RUN THIS CELL
         # The line %... is a jupyter "magic" command, and is not part of the Python langu
         # In this case we're just telling the plotting library to draw things on
         # the notebook, instead of on a separate window.
         %matplotlib inline
         # See the "import ... as ..." contructs below? They're just aliasing the package
         # That way we can call methods like plt.plot() instead of matplotlib.pyplot.plot(
         import numpy as np
         import scipy as sp
         import scipy.stats
         import matplotlib.pyplot as plt
```

Simulation of a Coin Throw

We'd like to do some experiments with coin flips, but we don't have a physical coin at the moment. So let's **simulate** the process of flipping a coin on a computer. To do this we will use a form of the **random number generator** built into `numpy` . In particular, we will use the function `np.random.choice` which picks items with uniform probability from a list. If we provide it a list ['H', 'T'], it will pick one of the two items in the list. We can also ask it to do this multiple times by specifying the parameter `size` .

```
In [5]:  def throw_a_coin(n_trials):
             return np.random.choice(['H','T'], size=n_trials)
```

`np.sum` is a function that returns the sum of items in an iterable (i.e. a list or an array). Because python coerces `True` to 1 and `False` to 0, the effect of calling `np.sum` on the array of `True` s and `False` s will be to return the number of of `True` s in the array (which can then effectively count

the number of heads).

Question 2: The 12 Labors of Bernoullis

Now that we know how to run our coin flip experiment, we're interested in knowing what happens as we choose larger and larger number of coin flips.

2.1. Run one experiment of flipping a coin 40 times storing the resulting sample in the variable `throws1` . What's the total proportion of heads?

2.2. **Replicate** the experiment in 2.1 storing the resulting sample in the variable `throws2` . What's the proportion of heads? How does this result compare to that you obtained in question 2.1?

2.3. Write a function called `run_trials` that takes as input a list, called `n_flips` , of integers representing different values for the number of coin flips in a trial. For each element in the input list, `run_trials` should run the coin flip experiment with that number of flips and calculate the proportion of heads. The output of `run_trials` should be the list of calculated proportions. Store the output of calling `run_trials` in a list called `proportions` .

2.4. Using the results in 2.3, reproduce the plot below.
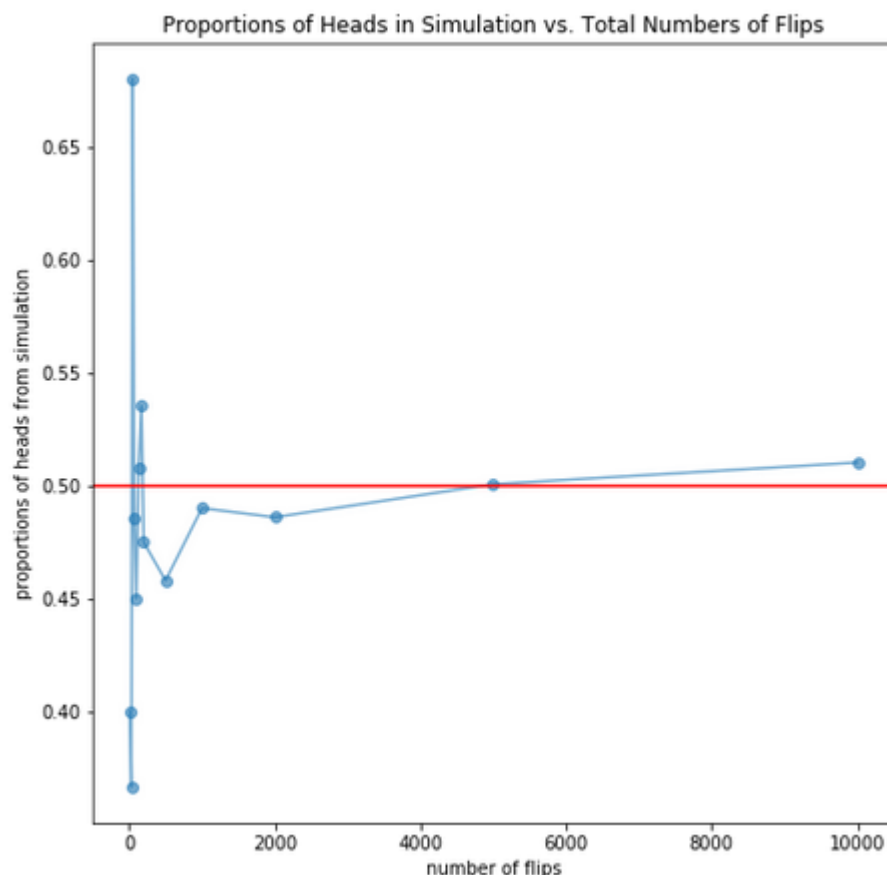
2.5. What's the appropriate observation about the result of running the coin flip experiment with larger and larger numbers of coin flips? Choose the appropriate one from the choices below and explain why.

A. Regardless of sample size the probability of in our experiment of observing heads is 0.5 so the proportion of heads observed in the coin-flip experiments will always be 0.5.

B. The proportions **fluctuate** about their long-run value of 0.5 (what you might expect if you tossed the coin an infinite amount of times), in accordance with the notion of a fair coin (which we encoded in our simulation by having `np.random.choice` choose between two possibilities with equal probability), with the fluctuations seeming to become much smaller as the number of trials increases.

C. The proportions **fluctuate** about their long-run value of 0.5 (what you might expect if you tossed the coin an infinite amount of times), in accordance with the notion of a fair coin (which we encoded in our simulation by having `np.random.choice` choose between two possibilities with equal probability), with the fluctuations constant regardless of the number of trials.

Answers

2.1

```
In [6]: nThrows = 40
        throws1 = throw_a_coin(nThrows)
        nHeads1 = 0
        for i in throws1:
            if i=='H':
                nHeads1 = nHeads1 + 1


        print(nHeads1/nThrows)
```

0.425

2.2

```
In [7]: nThrows = 40
        throws2 = throw_a_coin(nThrows)
        nHeads2 = 0
        for i in throws2:
            if i=='H':
                nHeads2 = nHeads2 + 1


        print(nHeads2/nThrows)
        print((nHeads1 + nHeads2)/(nThrows*2))
```

0.55
0.4875

The fraction of heads across the two trials are 0.53 and 0.45 respectively. The results are different.

2.3

```
In [8]: n_flips = [10, 30, 50, 70, 100, 130, 170, 200, 500, 1000, 2000, 5000, 10000]
```

```
In [9]: def run_trials(n_flips):

            proportions = []

            for i in range(len(n_flips)):
                nHeads = 0
                throws = throw_a_coin(n_flips[i])

                for j in throws:
                    if j=='H':
                        nHeads = nHeads+1

                proportions.append(nHeads/n_flips[i])

            return(proportions)
```
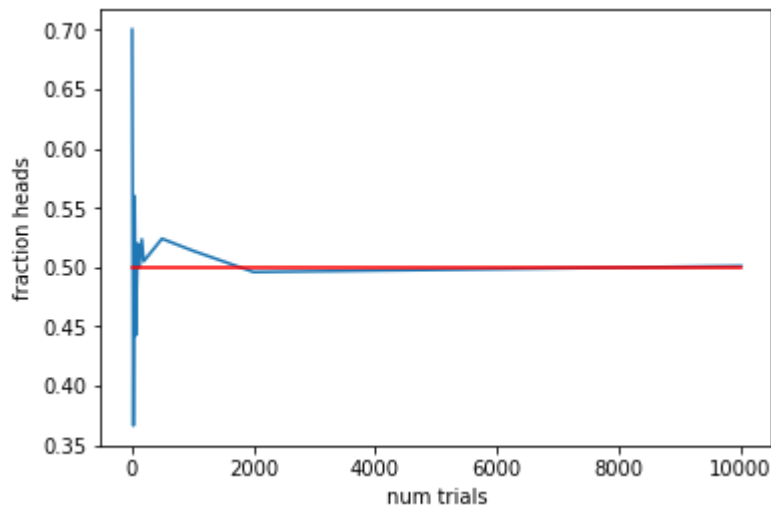
```
In [10]: proportions = run_trials(n_flips)
         print(proportions)
```

```
[0.7, 0.36666666666666664, 0.56, 0.44285714285714284, 0.52, 0.5, 0.523529411764
7059, 0.505, 0.524, 0.514, 0.496, 0.4978, 0.501]
```

2.4

In [11]:
```python
%matplotlib inline
import matplotlib.pyplot as plt

ph=plt.plot(n_flips,proportions)
plt.xlabel('num trials')
plt.ylabel('fraction heads')
plt.plot((0,n_flips[-1]),(0.5,0.5),'r')
```

Out[11]: [<matplotlib.lines.Line2D at 0x262bdc47470>]

2.5

What's the appropriate observation about the result of applying the coin flip experiment to larger and larger numbers of coin flips? Choose the appropriate one.

B. The fluctuation observed decreases as the sample size increases, as is consistent with the standard error of the mean. The standard error should go down as the square root of the number of trials.

# Multiple Replications of the Coin Flip Experiment

The coin flip experiment that we did above gave us some insight, but we don't have a good notion of how robust our results are under repetition as we've only run one experiment for each number of coin flips. Lets redo the coin flip experiment, but let's incorporate multiple repetitions of each number of coin flips. For each choice of the number of flips, n, in an experiment, we'll do M replications of the coin tossing experiment.

Question 3. So Many Replications

3.1. Write a function `make_throws` which takes as arguments the `n_replications` (M) and the `n_flips` (n), and returns a list (of size M) of proportions, with each proportion calculated by taking the ratio of heads to to total number of coin flips in each replication of n coin tosses.

n_flips should be a python parameter whose value should default to 20 if unspecified when make_throws is called.

3.2. Create the variables proportions_at_n_flips_100 and proportions_at_n_flips_1000 . Store in these variables the result of make_throws for n_flips equal to 100 and 1000 respectively while keeping n_replications at 200. Create a plot with the histograms of proportions_at_n_flips_100 and proportions_at_n_flips_1000 . Make sure to title your plot, label the x-axis and provide a legend.(See below for an example of what the plot may look like)
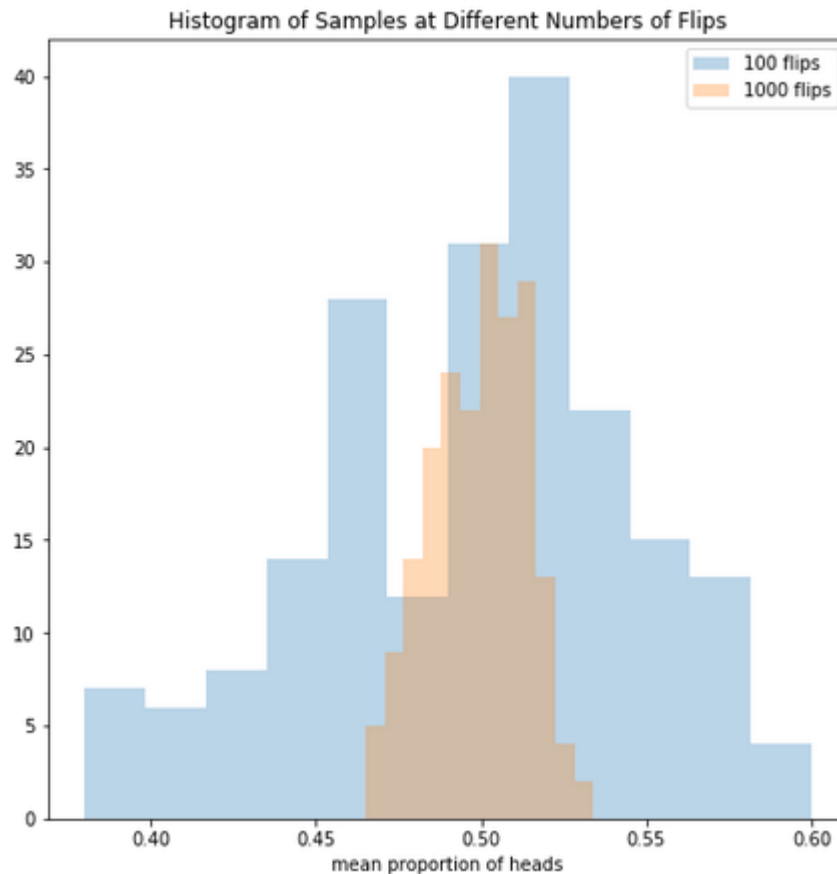
3.3. Calculate the mean and variance of the results in the each of the variables proportions_at_n_flips_100 and proportions_at_n_flips_1000 generated in 3.2.

3.4. Based upon the plots what would be your guess of what type of distribution is represented by histograms in 3.2? Explain the factors that influenced your choice.

> A. Gamma Distribution
>
> B. Beta Distribution
>
> C. Gaussian

3.5. Let's just assume for arguments sake that the answer to 3.4 is **C. Gaussian**. Plot a **normed histogram** of your results proportions_at_n_flips_1000 overlayed with your selection for the appropriate gaussian distribution to represent the experiment of flipping a coin 1000 times. (**Hint: What parameters should you use for your Gaussian?**)

3.1

```
In [12]: def make_throws(n_reps, n_flips=20):

             proportions = np.array([run_trials(n_flips) for _ in range(n_reps)])
             return(proportions)
```

3.2

```
In [13]: n_reps = 200
         proportions_n_flips_100 = make_throws(n_reps, n_flips = [100])
         proportions_n_flips_1000 = make_throws(n_reps, n_flips = [1000])
         print(proportions_n_flips_100.T)
         print(proportions_n_flips_1000.T)
```

```
[[0.49 0.58 0.46 0.48 0.55 0.47 0.56 0.5  0.45 0.5  0.54 0.52 0.41 0.51
  0.56 0.35 0.57 0.46 0.46 0.52 0.51 0.45 0.45 0.51 0.47 0.56 0.46 0.37
  0.49 0.52 0.53 0.47 0.51 0.45 0.49 0.6  0.5  0.45 0.5  0.51 0.52 0.6
  0.46 0.48 0.5  0.53 0.57 0.49 0.56 0.45 0.51 0.56 0.42 0.46 0.53 0.5
  0.54 0.5  0.49 0.45 0.55 0.46 0.56 0.51 0.45 0.51 0.44 0.5  0.43 0.54
  0.47 0.55 0.46 0.51 0.52 0.46 0.53 0.49 0.62 0.51 0.45 0.54 0.45 0.46
  0.55 0.56 0.53 0.55 0.46 0.48 0.45 0.57 0.49 0.38 0.55 0.55 0.56 0.51
  0.58 0.53 0.54 0.46 0.46 0.39 0.46 0.52 0.52 0.52 0.52 0.45 0.47 0.41
  0.38 0.58 0.47 0.58 0.58 0.53 0.46 0.48 0.46 0.39 0.5  0.47 0.43 0.52
  0.43 0.47 0.47 0.53 0.53 0.52 0.52 0.43 0.51 0.43 0.53 0.5  0.41 0.52
  0.54 0.59 0.49 0.51 0.5  0.57 0.5  0.46 0.55 0.53 0.53 0.52 0.55 0.46
  0.56 0.55 0.52 0.42 0.5  0.52 0.46 0.49 0.41 0.55 0.53 0.48 0.57 0.53
  0.44 0.59 0.35 0.47 0.49 0.52 0.51 0.45 0.44 0.35 0.44 0.45 0.51 0.38
  0.42 0.48 0.56 0.57 0.48 0.54 0.52 0.49 0.43 0.59 0.61 0.43 0.49 0.44
  0.48 0.48 0.5  0.48]]
[[0.494 0.48  0.482 0.478 0.472 0.514 0.522 0.481 0.498 0.487 0.522 0.515
  0.477 0.489 0.499 0.509 0.477 0.493 0.496 0.51  0.473 0.492 0.521 0.483
  0.485 0.515 0.51  0.494 0.489 0.515 0.503 0.506 0.497 0.489 0.507 0.501
  0.479 0.482 0.488 0.484 0.507 0.496 0.508 0.498 0.483 0.48  0.503 0.487
  0.511 0.485 0.478 0.49  0.514 0.505 0.509 0.519 0.483 0.519 0.496 0.496
  0.536 0.471 0.489 0.483 0.5   0.505 0.485 0.506 0.469 0.487 0.504 0.502
  0.499 0.483 0.486 0.523 0.483 0.491 0.5   0.486 0.503 0.513 0.5   0.499
  0.519 0.491 0.505 0.512 0.538 0.496 0.506 0.53  0.497 0.513 0.499 0.49
  0.495 0.518 0.497 0.527 0.496 0.51  0.487 0.492 0.513 0.526 0.5   0.518
  0.501 0.498 0.517 0.525 0.497 0.508 0.511 0.492 0.508 0.505 0.49  0.495
  0.487 0.509 0.502 0.471 0.52  0.47  0.507 0.51  0.525 0.485 0.465 0.458
  0.52  0.512 0.505 0.536 0.518 0.51  0.525 0.502 0.51  0.506 0.482 0.51
  0.475 0.477 0.513 0.499 0.487 0.517 0.522 0.496 0.528 0.485 0.526 0.527
  0.501 0.492 0.501 0.503 0.466 0.476 0.507 0.502 0.478 0.497 0.484 0.523
  0.497 0.504 0.469 0.471 0.498 0.471 0.518 0.508 0.489 0.507 0.521 0.502
  0.489 0.487 0.504 0.519 0.514 0.508 0.501 0.526 0.508 0.493 0.487 0.507
  0.493 0.525 0.489 0.515 0.506 0.5   0.494 0.491]]
```

```
In [14]: import seaborn as sns
         sns.distplot(proportions_n_flips_100, label='100 flips')
         sns.distplot(proportions_n_flips_1000, label='1000 flips')
         plt.xlabel('fraction heads')
         plt.title('histogram of coin flip repetitions')
         plt.legend()
```
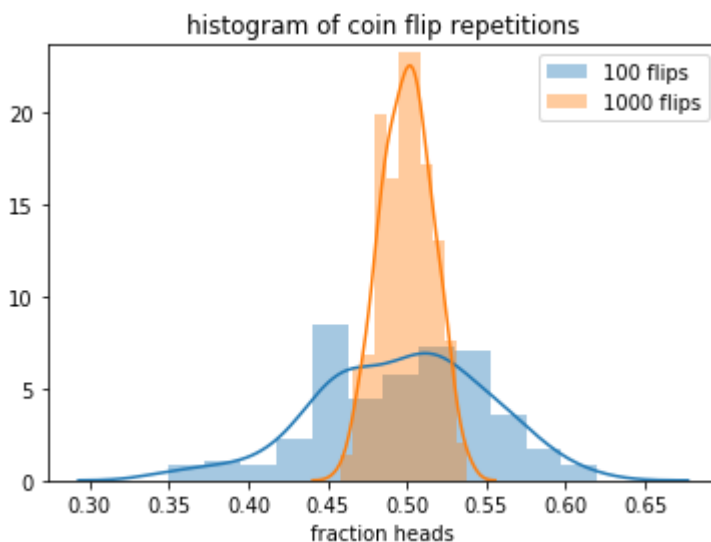
```
C:\Users\werkh\Anaconda3\lib\site-packages\matplotlib\axes\_axes.py:6462: UserW
arning: The 'normed' kwarg is deprecated, and has been replaced by the 'densit
y' kwarg.
  warnings.warn("The 'normed' kwarg is deprecated, and has been "
C:\Users\werkh\Anaconda3\lib\site-packages\matplotlib\axes\_axes.py:6462: UserW
arning: The 'normed' kwarg is deprecated, and has been replaced by the 'densit
y' kwarg.
  warnings.warn("The 'normed' kwarg is deprecated, and has been "
```

Out[14]: <matplotlib.legend.Legend at 0x262c02399b0>

3.3

```
In [15]: mu_100 = np.mean(proportions_n_flips_100)
         mu_1000 = np.mean(proportions_n_flips_1000)
         std_100 = np.std(proportions_n_flips_100)
         std_1000 = np.std(proportions_n_flips_1000)

         print('100 flips mean = %0.2f, std = %0.2f' %( mu_100, std_100))
         print('100 flips mean = %0.2f, std = %0.2f' %( mu_1000, std_1000))
```

```
100 flips mean = 0.50, std = 0.05
100 flips mean = 0.50, std = 0.02
```

3.4

The values appear to be normally distributed since the mean and shape of the distribution does not appear to change with increased sampling (apart from the dispersion)

3.5

Coin flips should be binomially distributed, with the probability of a heads $p_h = 0.5$

For a binomial distribution, $\mu = np_h$ and $\sigma = \sqrt{np_h(1 - p_h)}$

The binomial distributions approaches the normal distribution for large values of n, and since we are running 1000 flips for each repetition, it is reasonable to assume that it should approach the normal distribution. Since we are interested in the proportion, and not the total number of successes. Therefore, I'll plot

$$N(\mu, \sigma)$$

$$where$$

$$\mu = np_h = (1000)(0.5) = 500$$

$$\sigma = \sqrt{np_h(1 - p_h)} = \sqrt{(1000)(0.5)(0.5)} = 15.8$$

In [16]:
```python
# normalize throw data
normalized_flips_1000 = proportions_n_flips_1000/np.sum(proportions_n_flips_1000)

# gaussian parameters
mu = 0.5
sig = 0.0158
my_gauss = lambda x: (1/(sig*np.sqrt(2*np.pi)))*np.exp((-1*np.power((x-mu),2)/(2*
x = np.linspace(0,1,100)
gauss_dist = np.array([my_gauss(i) for i in x])
plt.plot(x, gauss_dist, LineWidth=3,label='gaussian')
plt.hist(proportions_n_flips_1000, normed=True, label='observed data')
plt.legend()
```

```
C:\Users\werkh\Anaconda3\lib\site-packages\matplotlib\axes\_axes.py:6462: UserW
arning: The 'normed' kwarg is deprecated, and has been replaced by the 'densit
y' kwarg.
  warnings.warn("The 'normed' kwarg is deprecated, and has been "
```

Out[16]: <matplotlib.legend.Legend at 0x262c137d4a8>
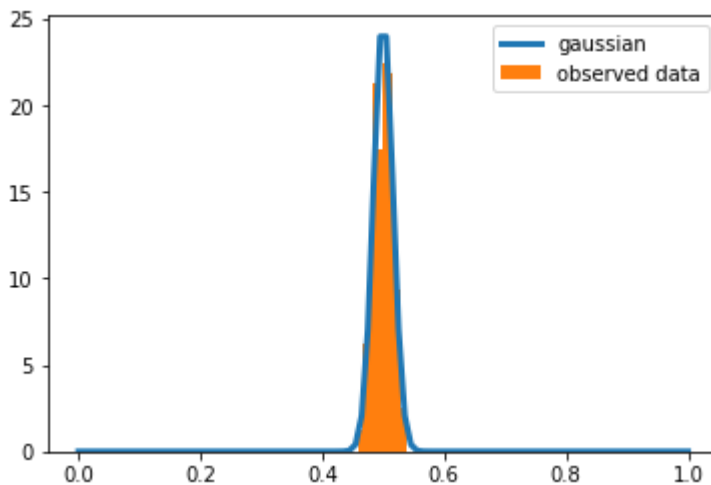
Working With Distributions in Numpy/Scipy

Earlier in this problem set we've been introduced to the Bernoulli "aka coin-flip" distribution and worked with it indirectly by using np.random.choice to make a random selection between two elements 'H' and 'T'. Let's see if we can create comparable results by taking advantage of the machinery for working with other probability distributions in python using numpy and scipy.

Question 4: My Normal Binomial

Let's use our coin-flipping machinery to do some experimentation with the binomial distribution. The binomial distribution, often represented by $k \sim Binomial(n, p)$ is often discribed the number of successes in n Bernoulli trials with each trial having a probability of success p . In other words, if you flip a coin n times, and each coin-flip has a probability p of landing heads, then the number of heads you observe is a sample from a binomial distribution.

4.1. Sample the binomial distribution with $p = 0.5$ using coin flips by writing a function `sample_binomial1` which takes in integer parameters `n` and `size`. The output of `sample_binomial1` should be a list of length `size` observations with each observation being the outcome of flipping a coin `n` times and counting the number of heads. By default `size` should be 1. Your code should take advantage of the `throw_a_coin` function we defined above.

4.2. Sample the binomial distribution directly using scipy.stats.binom.rvs by writing another function `sample_binomial2` that takes in integer parameters `n` and `size` as well as a float `p` parameter `p` where $p \in [0 \ldots 1]$. The output of `sample_binomial2` should be a list of length `size` observations with each observation a sample of $Binomial(n, p)$ (taking advantage of scipy.stats.binom). By default `size` should be 1 and `p` should be 0.5.

4.3. Run sample_binomial1 with 25 and 200 as values of the `n` and `size` parameters respectively and store the result in `binomial_trials1`. Run sample_binomial2 with 25, 200 and 0.5 as values of the `n`, `size` and `p` parameters respectively and store the results in `binomial_trials2`. Plot normed histograms of `binomial_trials1` and `binomial_trials2`. On both histograms, overlay a plot of the pdf of $Binomial(n = 25, p = 0.5)$

4.4. How do the plots in 4.3 compare?

4.5. Find the mean and variance of `binomial_trials1`. How do they compare to the true mean and varaince of a $Binomial(n = 25, p = 0.5)$ distribution?

Answers

4.1

```
In [17]:  def sample_binomial1(n, size=1):

              n_heads = np.zeros(size)

              for i in range(len(n_heads)):
                  flips = throw_a_coin(n)
                  for j in flips:
                      if j=='H':
                          n_heads[i]+=1

              return(n_heads)

          n_heads = sample_binomial1(40, size=20)

          print(n_heads)
```

```
[18. 27. 20. 16. 24. 21. 23. 23. 29. 20. 21. 18. 19. 14. 21. 21. 25. 27.
 20. 24.]
```

4.2

In [18]:
```python
from scipy.stats import binom

print(binom.rvs(30, p=0.5, size=20))
```

```
[19 16 14 18 13 12 20 15 14 17 11 15 15 18 12 13 16 16 16 17]
```

In [19]:
```python
from scipy.stats import binom

def sample_binomial2(n, size=1, p=0.5):

    return(binom.rvs(n, p=p, size=size))

n_successes = sample_binomial2(40, size=100, p=0.5)
print(n_successes)
```

```
[17 24 19 15 20 21 16 21 20 18 20 19 21 25 16 19 16 19 22 21 21 22 19 19
 20 13 23 14 23 23 17 18 21 19 19 17 17 18 18 17 15 24 16 22 19 20 15 15
 18 20 20 21 17 19 23 24 20 22 17 19 24 19 19 18 19 21 20 23 29 23 20 18
 23 20 22 23 20 20 14 18 19 24 23 20 15 18 16 18 21 20 15 20 21 15 21 16
 23 17 15 18]
```

4.3

In [20]:
```python
n=25
sz = 200

binomial_trials1 = sample_binomial1(n, size=sz)
binomial_trials2 = sample_binomial2(n, size=sz, p=p)

plt.hist(binomial_trials1, normed=True, alpha=0.5, label='sample_binom1')
plt.hist(binomial_trials2, normed=True, alpha=0.5, label='sample_binom2')
x = np.arange(binom.ppf(0.0001, n, p),binom.ppf(0.999999, n, p))
plt.plot(x, binom.pmf(x, n, p), 'b', ms=8, label='binom pmf')
plt.xlabel('num heads')
plt.legend()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-20-ae9653bdb7a1> in <module>()
      3
      4 binomial_trials1 = sample_binomial1(n, size=sz)
----> 5 binomial_trials2 = sample_binomial2(n, size=sz, p=p)
      6
      7 plt.hist(binomial_trials1, normed=True, alpha=0.5, label='sample_binom
1')

NameError: name 'p' is not defined
```

4.4

The distributions of the two random sampling methods appear qualitatively the same and are both show some error around the PDF, which is expected due to sampling.

4.5

```
In [ ]:  mu_100 = np.mean(binomial_trials1)
         std_100 = np.std(binomial_trials1)
         print(mu_100)
         print(std_100)
```

The observed mean and standard deviation are

$$\mu = 12.405, \quad \sigma = 2.38$$

The true mean and variance for $Binomial(n = 25, p = 0.5)$

$$\mu = np = (25)(0.5) = 12.5$$

$$\sigma = \sqrt{np(1 - p)} = \sqrt{(25)(0.5)(0.5)} = 2.5$$

Testing Your Python Code

In the following section we're going to do a brief introduction to unit testing. We do so not only because unit testing has become an increasingly important part of of the methodology of good software practices, but also because we plan on using unit tests as part of our own CS109 grading practices as a way of increasing rigor and repeatability decreasing complexity and manual workload in our evaluations of your code. We'll provide an example unit test at the end of this section.

Introduction to unit testing

```
In [ ]:  import ipytest
```

Unit testing is one of the most important software testing methodologies. Wikipedia describes unit testing as "a software testing method by which individual units of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures, are tested to determine whether they are fit for use."

There are many different python libraries that support software testing in general and unit testing in particular. PyTest is one of the most widely used and well-liked libraries for this purpose. We've chosen to adopt PyTest (and ipytest which allows pytest to be used in ipython notebooks) for our testing needs and we'll do a very brief introduction to Pytest here so that you can become familiar with it too.

If you recall the function that we provided you above `throw_a_coin`, which we'll reproduce here for convenience, it took a number and returned that many "coin tosses". We'll start by seeing what happens when we give it different sizes of N. If we give $N = 0$, we should get an empty array of "experiments".

```
In [ ]: def throw_a_coin(N):
            return np.random.choice(['H','T'], size=N)
```

```
In [ ]: throw_a_coin(0)
```

Great! If we give it positive values of N we should get that number of 'H's and 'T's.

```
In [ ]: throw_a_coin(5)
```

```
In [ ]: throw_a_coin(8)
```

Exactly what we expected!

What happens if the input isn't a positive integer though?

```
In [ ]: throw_a_coin(4.5)
```

or

```
In [ ]: throw_a_coin(-4)
```

It looks like for both real numbers and negative numbers, we get two kinds of errors a `TypeError` and a `ValueError`. We just engaged in one of the most rudimentary forms of testing, trial and error. We can use pytest to automate this process by writing some functions that will automatically (and potentially repeatedly) test individual units of our code methodology. These are called **unit tests**.

Before we write our tests, let's consider what we would think of as the approrpriate behavior for `throw_a_coin` under the conditions we considered above. If `throw_a_coin` receives positive integer input, we want it to behave exactly as it currently does -- returning an output consisting of a list of characters 'H' or 'T' with the length of the list equal to the positive integer input. For a positive floating point input, we want `throw_a_coin_properly` to treat the input as if it were rounded down to the nearest integer thus returning a list of 'H' or 'T' integers whose length is the same as the input rounded down to the next highest integer. For a any negative number input or an input of 0, we want `throw_a_coin_properly` to return an empty list.

We create pytest tests by writing functions that start or end with "test". We'll use the **convention** that our tests will start with "test".

We begin the code cell with ipytest's clean_tests function as a way to clear out the results of previous tests starting with "test_throw_a_coin" (the * is the standard wild card charater here).

In [ ]:

```
## the * after test_throw_a_coin tells this code cell to clean out the results
## of all tests starting with test_throw_a_coin
ipytest.clean_tests("test_throw_a_coin*")

## run throw_a_coin with a variety of positive integer inputs (all numbers betwee
## verify that the length of the output list (e.g ['H', 'H', 'T', 'H', 'T']) matc
def test_throw_a_coin_length_positive():
    for n in range(1,20):
        assert len(throw_a_coin(n)) == n

## verify that throw_a_coin produces an empty list (i.e. a list of length 0) if p
## of 0
def test_throw_a_coin_length_zero():
    ## should be the empty array
    assert len(throw_a_coin(0)) == 0


## verify that given a positive floating point input (i.e. 4.34344298547201), thr
## coin flips of length equal to highest integer less than the input
def test_throw_a_coin_float():
    for n in np.random.exponential(7, size=5):
        assert len(throw_a_coin(n)) == np.floor(n)


## verify that given any negative input (e.g. -323.4), throw_a_coin produces an e
def test_throw_a_coin_negative():
    for n in range(-7, 0):
        assert len(throw_a_coin(n)) == 0


ipytest.run_tests()
```

As you see, we were able to use pytest (and ipytest which allows us to run pytest tests in our ipython notebooks) to automate the tests that we constructed manually before and get the same errors and successes. Now time to fix our code and write our own test!

Question 5: You Better Test Yourself before You Wreck Yourself!

Now it's time to fix `throw_a_coin` so that it passes the tests we've written above as well as add our own test to the mix!

5.1. Write a new function called `throw_a_coin_properly` that will pass the tests that we saw above. For your convenience we'll provide a new jupyter notebook cell with the tests rewritten for the new function. All the tests should pass. For a positive floating point input, we want `throw_a_coin_properly` to treat the input as if it were rounded down to the nearest integer. For a any negative number input, we want `throw_a_coin_properly` to treat the input as if it were 0.

5.2. Write a new test for `throw_a_coin_properly` that verifies that all the elements of the resultant arrays are 'H' or 'T'.

Answers

5.1

```
In [ ]:  def throw_a_coin_properly(n_trials):
             n_trials = int(n_trials)
             if n_trials < 0:
                 n_trials = 0
             return np.random.choice(['H','T'], size=n_trials)
```

```
In [ ]:  ipytest.clean_tests("test_throw_a_coin*")

         def test_throw_a_coin_properly_length_positive():
             for n in range(1,20):
                 assert len(throw_a_coin_properly(n)) == n


         def test_throw_a_coin_properly_length_zero():
             ## should be the empty array
             assert len(throw_a_coin_properly(0)) == 0


         def test_throw_a_coin_properly_float():

             for n in np.random.exponential(7, size=5):
                 assert len(throw_a_coin_properly(n)) == np.floor(n)


         def test_throw_a_coin_properly_negative():

             for n in range(-7, 0):
                 assert len(throw_a_coin_properly(n)) == 0


         ipytest.run_tests()
```

5.2

```
In [ ]:  ipytest.clean_tests("test_throw_a_coin*")

         ## write a test that verifies you don't have any other elements except H's and T'
         def test_throw_a_coin_properly_verify_H_T():

             for n in range(1,20):
                 flips = throw_a_coin_properly(n)
                 ct = 0
                 for j in flips:
                     if j=='H' or j=='T':
                         ct+=1
                 assert ct == n


         ipytest.run_tests()
```

In [ ]:
```python
from IPython.core.display import HTML
def css_styling():
    styles = open("../../styles/cs109.css", "r").read()
    return HTML(styles)
css_styling()
```