S-109A 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 but it won't be graded.

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.

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 (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}.

In [41]:

```
import numpy as np
#1.1

#define matrices
A = np.matrix([[3,4,2],[5,6,4],[4,3,4]])
B = np.matrix([[1,4,2],[1,9,3],[2,3,3]])

#calculate A*B
print(A*B)
```

```
[[11 54 24]
 [19 86 40]
 [15 55 29]]
```

In [42]:

```
import numpy as np
#1.2

#define matrix

A = np.matrix([[0,12,8],[1,15,0],[0,6,3]])
print(A)

#Inverse

Inverse_A = np.linalg.inv(A)
print(Inverse_A)
```

```
[[ 0 12  8]
 [ 1 15  0]
 [ 0  6  3]]
[[  3.75         1.         -10.        ]
 [ -0.25         0.           0.66666667]
 [  0.5         -0.          -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. Two central aspects of call center operations are the busy-ness of the customer service reps and the caller demographics. Because of historical data, the distribution of these measures can often take on well-known distributions. In the CS109 Homework Helpdesk, X and Y are discrete random variables with Y measuring the total number of help lines that are busy with callers (there are three lines available), and X measuring the number of help lines that are busy with female callers. We've determined historically the joint pmf of (X, Y) and found it to be:

$$p_{X,Y}(x, y) = \binom{y}{x}\left(\frac{1}{3}\right)^x\left(\frac{2}{3}\right)^{y-x}\frac{(1+y)}{10}$$

where $y \in 0, 1, 2, 3, x \in [0, y]$(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). Note: by definition, $\binom{0}{0} = 1$.

(i) What is the conditional distribution of $X|Y$ (it is a well-known family)?

(ii) Find the mean and variance of the marginal distribution of X.

1.3 solution

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

The derivatives of L(x,y) are

$L_x(x, y) = 6xy - 6x$

$L_y(x, y) = 3x^2 - 3y^2 - 6y$

Setting each equation eqaul to zero, we have critical points $(0, 0), (0, -2), (0, \pm\sqrt{3})$.

Then, we will find which critical points are saddle points, do second derivatives.

$L_{xx}(x, y) = 6y - 6$

$L_{yy}(x, y) = 6x$

$L_{yy}(x, y) = -6y - 6$

$D(x, y) = L_{xx}(x, y) * L_{yy}(x, y) - (L_{yy}(x, y))^2$

$= -36(x^2 + y^2 - 1)$

So, put critical points. (0,0) = 36. It seems Local Maximum at this point.

(0,-2) = -108

$(0, \pm\sqrt{3}))$ = -108 Two of them are Saddle points.

1.4 solution

i)

First, find X|Y

$X|Y = P_{X,Y}(X,Y)/P_Y(Y)$
$P_Y(Y) = \sum_{x=0}^{y} P_{X,Y}(X,Y)$
$= \sum_{x=0}^{y} \binom{y}{x}\left(\frac{1}{3}\right)^x\left(\frac{2}{3}\right)^{y-x}\left(\frac{1+y}{10}\right)$
$= \left(\frac{1+y}{10}\right)\sum_{x=0}^{y} \binom{y}{x}\left(\frac{1}{3}\right)^x\left(\frac{2}{3}\right)^{y-x}$
$= \left(\frac{1+y}{10}\right)\left\{\frac{1}{3}+\frac{2}{3}\right\}^y$
$= \frac{1+y}{10}$
$\therefore P(X|Y) = \binom{y}{x}\left(\frac{1}{3}\right)^x\left(\frac{2}{3}\right)^{y-x}$

This is binomial distribution.

ii)

we need to find the mean and variance of X, numbers of female callers.

mean = $\sum_{x=0}^{3} x * P_X(X)$

There can be 0, 1, 2 and 3 female callers

$$\sum_{x=0}^{3} x * P_X(X)$$
$$= 0 * P_X(0) + 1 * P_X(1) + 2 * P_X(2) + 3 * P_X(3)$$
$$0 * P_X(0)$$

can be happen in 3 ways.

when y=1,2 and 3.

$$P_x(1) = P_{x,y}(1,1) + P_{x,y}(1,2) + P_{x,y}(1,3)$$

$$= \frac{1}{15} + \frac{2}{15} + \frac{8}{45}$$

$$= \frac{17}{45}$$

similarily,

$$P_x(2) = P_{x,y}(2,2) + P_{x,y}(2,3)$$

$$= \frac{1}{30} + \frac{4}{45}$$

$$= \frac{11}{90}$$

$$P_x(3) = P_{x,y}(3,3)$$

$$= \frac{2}{135}$$

$$\therefore mean = \frac{17}{45} + 2 * \frac{11}{90} + 3 * \frac{2}{135}$$

$$= 2/3$$

$$variance = \sum x^2 P_x(x) - (\sum x P_x(x))^2$$

$$= 1^2 \frac{17}{45} + 2^2 \frac{11}{90} + 3^2 \frac{2}{135} - (\sum x P_x(x))^2$$

$$= 1 - \frac{2}{3}^2$$

$$= \frac{5}{9}$$

In [43]:
```
# The line %... is a jupyter "magic" command, and is not part of the Python la
nguage.
# 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 packa
ge names.
# That way we can call methods like plt.plot() instead of matplotlib.pyplot.pl
ot().
import numpy as np
import scipy as sp
import scipy.stats
import matplotlib.pyplot as plt
```

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) is related to sex. Be sure to state your conclusion.

In [44]:
```
# 1.5 solution
# use Chi-squared test for whether each values(gender) is independent or not
# two-way chi-squared test

import numpy as np
from scipy.stats import chi2_contingency

observation = np.array([[37,39],[50,83]])
print(chi2_contingency(observation))
```

```
(2.0128746736116723, 0.15596953904839861, 1, array([[ 31.63636364,  44.363636
36],
       [ 55.36363636,  77.63636364]]))
```

explanation

(2.0128746736116723, 0.15596953904839861, 1, array([[31.63636364, 44.36363636], [55.36363636, 77.63636364]]))

```
is return value of chi-square two way independent test on above.
```

2.0128746736116723 = the test statistic. 0.15596953904839861 = p-value 1 = degree of freedom (because it is 2x2 table, (2-1)(2-1) = 1

array([[31.63636364, 44.36363636], [55.36363636, 77.63636364]]) = expected values.

Therefore, p-value is about 0.16 which is > 0.05 level.

It is failed to reject the null of they are unrelated, independent at a = 0.05 level since p-value >0.05.

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 [45]:
```
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.

> 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 [46]:
```
## Your code here

import numpy as np

def throw_a_coin(n_trials):
    return np.random.choice([1,0], size=n_trials)
# Head is 1 , Tail is 0
throw1 = np.sum(throw_a_coin(40))/40

#To find proportion of Head
#Divide by total,40.

print(throw1)
```

```
0.5
```

2.2

In [47]:
```
## Your code here

#replicate it in same way

throw2 = np.sum(throw_a_coin(40))/40

#compare with throw1

print(throw1)
print(throw2)

#it probabaly different because random function make new value every time.
#although it is possibly same when they make same value.
```

```
0.5
0.475
```

2.3

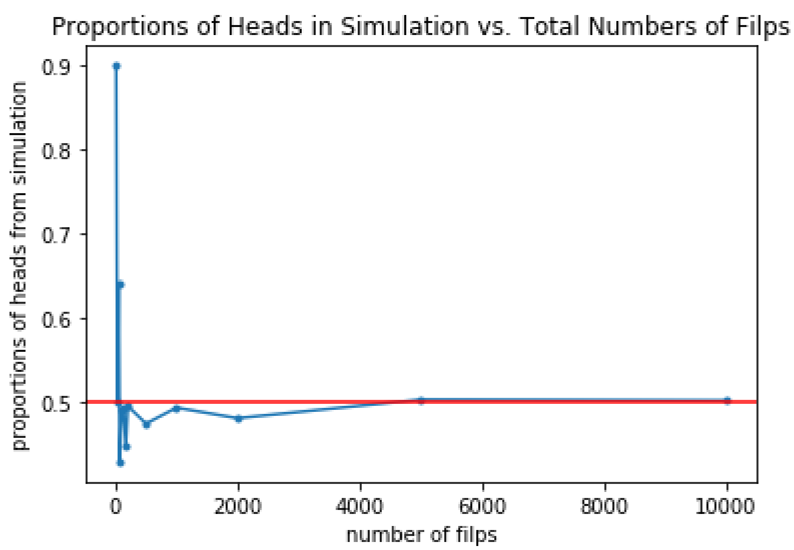
In [48]:
```
n_flips = [10, 30, 50, 70, 100, 130, 170, 200, 500, 1000, 2000, 5000, 10000]
```

In [49]:
```
## Your code here

n_flips = [10, 30, 50, 70, 100, 130, 170, 200, 500, 1000, 2000, 5000, 10000]
def run_trials(List):
    A = np.array([])
    for n in List:
        A = np.append(A,np.sum(throw_a_coin(n))/n)
    return A

proportions = run_trials(n_flips)

print(proportions)
```

```
[ 0.9         0.5         0.64        0.42857143  0.49        0.49230769
  0.44705882  0.495       0.474       0.493       0.4805      0.5026
  0.5016    ]
```

2.4

In [50]:
```
## your code here
plt.plot(n_flips,proportions, marker ='.')
plt.xlabel('number of filps')
plt.ylabel('proportions of heads from simulation')
plt.title('Proportions of Heads in Simulation vs. Total Numbers of Filps')
plt.axhline(y=0.5, color='red')
plt.show()
```



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.

your choice here

B

Because as you can see, it gets smaller when the trials increases.

And if we change x axis and y axis, you can see that the graph is normal distribution.

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.

Answers

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

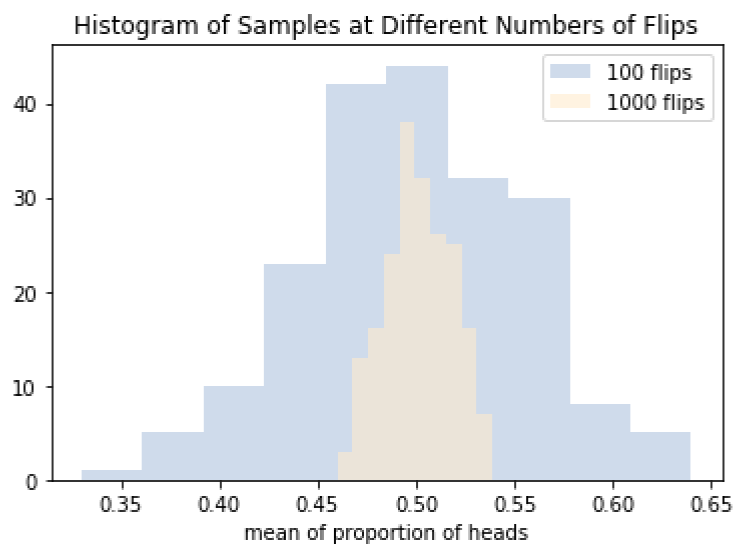
In [51]:
```
# your code here

def make_throws(n_replications,n_flips=20):
    list_proportion = np.array([])
    for i in range(n_replications):
        list_proportion = np.append(list_proportion,np.sum(throw_a_coin(n_flip
s))/n_flips)
    return list_proportion

print(make_throws(10))
```

```
[ 0.45  0.4   0.45  0.65  0.45  0.2   0.35  0.35  0.55  0.5 ]
```

3.2

In [52]:
```
# your code here
import matplotlib.pyplot as plt

_at_n_flips_100 = make_throws(200,100)
_at_n_flips_1000 = make_throws(200,1000)
```

In [53]:
```
# code for your plot here


plt.hist(_at_n_flips_100, label="100 flips",alpha=0.6, color='lightsteelblue')
plt.hist(_at_n_flips_1000, label="1000 flips", alpha=0.6,color='blanchedalmon
d')
plt.xlabel('mean of proportion of heads')
plt.title('Histogram of Samples at Different Numbers of Flips')
plt.legend()
plt.show()
```



3.3

In [54]:
```
# your code here
print(np.mean(_at_n_flips_100),np.var(_at_n_flips_100))
print(np.mean(_at_n_flips_1000),np.var(_at_n_flips_1000))
```

```
0.49985 0.0028844775
0.501455 0.000299437975
```

3.4

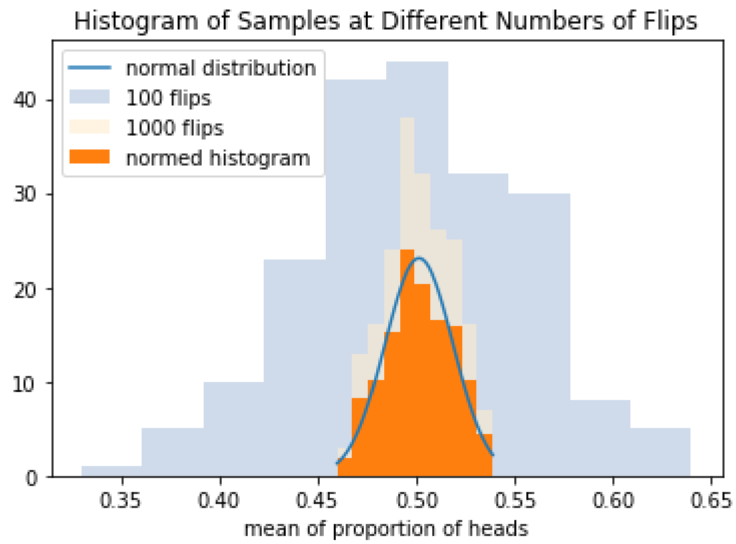
Your choice and explanation here

Answer is C

because we have get mean and variance in above, it is explanation that the graph is distributed with roughly 0.5(mean) and 0.26(variance). Also if we generate _at_n_flips_1000000 , for instance, we can see the graph are more nomalizing when the values increase.

3.5

In [55]:
```
# your code here
import matplotlib.mlab as mlab

# define the values, and make a plot.
mu = np.mean(_at_n_flips_1000)
sigma = np.std(_at_n_flips_1000)
x = np.linspace(min(_at_n_flips_1000),max(_at_n_flips_1000))
plt.plot(x, mlab.normpdf(x,mu,sigma), label="normal distribution")
plt.hist(_at_n_flips_100, label="100 flips",alpha=0.6, color='lightsteelblue')
plt.hist(_at_n_flips_1000, label="1000 flips", alpha=0.6,color='blanchedalmon
d')
plt.hist(_at_n_flips_1000,normed=True , label="normed histogram")
plt.xlabel('mean of proportion of heads')
plt.title('Histogram of Samples at Different Numbers of Flips')
plt.legend()
plt.show()
```



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 [56]:
```
# your code here

import numpy as np

def throw_a_coin(n_trials):
    return np.random.choice([1,0], size=n_trials)


def sample_binomial1(n,size=1):
    A = np.array([])
    for i in range(size):
        A = np.append(A,np.sum(throw_a_coin(n)))
    return A

print(sample_binomial1(10))

print(sample_binomial1(50,2))
```

```
[ 5.]
[ 30.  23.]
```

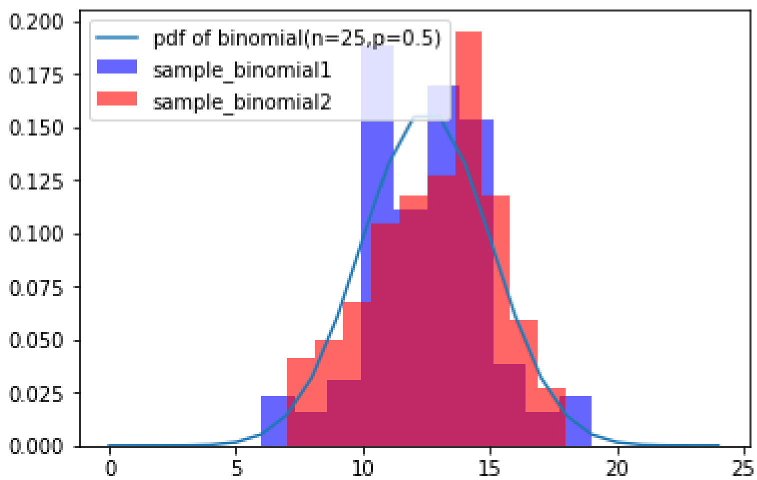
4.2

In [57]:
```
# your code

import scipy.stats as stats


def sample_binomial2(n,p=0.5,size=1):
    return (stats.binom.rvs(n,p,size=size))

print(sample_binomial2(50,0.5,2))
```

```
[30 28]
```

4.3

In [58]:
```
# your code here

x1=sample_binomial1(25,200)

x2=sample_binomial2(25,0.5,200)

x=range(25)
y=stats.binom.pmf(x,25,0.5)

plt.plot(x,y,label="pdf of binomial(n=25,p=0.5)")
plt.hist(x1, label="sample_binomial1", color="blue" ,alpha =0.6, normed=True)
plt.hist(x2, label="sample_binomial2" , color="red" ,alpha =0.6, normed=True)
plt.legend()
plt.show()
```



4.4

Your explanation here

Answer

First of all, the plot of the pdf of binomial is apparently normal distribution.

Sample_binomial 1 and sample_binomial 2 also look like normally distributted.

If n is increased, the graph will be more normalized.

And Sample_binomial 1 and sample_binomial 2 look different , although the basis of calcuation is same.

Because it has a different answer due to randome varibles in two function. Unless putting some seed number, it would get

diverged answers. However it is technically same idea and working similarily.

4.5

In [59]:
```
# your code here
print(np.mean(x1),np.var(x1))
print(np.mean(y),np.var(y))
```

```
12.445 5.856975
0.0399999988079 0.00289100700174
```

Your explanation here

Why this answer is coming out is that sample_binomial1 is based on counting number of heads when filpping occurs.

But binomial(n,p) is just about pdf of binomial formula. That is why it has pretty significant different.

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 [60]:
```
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 [61]:
```
def throw_a_coin(N):
    return np.random.choice(['H','T'], size=N)
```

In [62]:
```
throw_a_coin(0)
```

Out[62]:
```
array([],
      dtype='<U1')
```

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

In [63]:
```
throw_a_coin(5)
```

Out[63]:
```
array(['H', 'T', 'H', 'T', 'T'],
      dtype='<U1')
```

In [64]:
```
throw_a_coin(8)
```

Out[64]:
```
array(['H', 'H', 'H', 'T', 'H', 'T', 'H', 'H'],
      dtype='<U1')
```

Exactly what we expected!

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

In [65]:
```
throw_a_coin(4.5)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-65-0a0318674769> in <module>()
----> 1 throw_a_coin(4.5)

<ipython-input-61-54a1ec744eb3> in throw_a_coin(N)
      1 def throw_a_coin(N):
----> 2     return np.random.choice(['H','T'], size=N)

mtrand.pyx in mtrand.RandomState.choice()

mtrand.pyx in mtrand.RandomState.randint()

mtrand.pyx in mtrand.RandomState.randint()

randint_helpers.pxi in mtrand._rand_int32()

TypeError: 'float' object cannot be interpreted as an integer
```

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 bet
ween 1 and 20) and
## verify that the length of the output list (e.g ['H', 'H', 'T', 'H', 'T']) m
atches the input integer
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) i
f provide with an input
## 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),
 throw_a_coin produces a list of
## 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 a
n empty
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 [ ]:
```
import numpy as np
def throw_a_coin_properly(n_trials):
    if n_trials <= 0: #negative values
        return np.random.choice(['H','T'], size=0)
    else: #float values
        return np.random.choice(['H','T'], size=int(n_trials))

print(throw_a_coin_properly(-5.5))
```

In [67]:
```
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()
```

```
unittest.case.FunctionTestCase (test_throw_a_coin_properly_float) ... ok
unittest.case.FunctionTestCase (test_throw_a_coin_properly_length_positive)
... ok
unittest.case.FunctionTestCase (test_throw_a_coin_properly_length_zero) ... o
k
unittest.case.FunctionTestCase (test_throw_a_coin_properly_negative) ... ok

----------------------------------------------------------------------
Ran 4 tests in 0.007s

OK
```

5.2

In [66]:
```
ipytest.clean_tests("test_throw_a_coin*")

## write a test that verifies you don't have any other elements except H's and
 T's
def test_throw_a_coin_properly_verify_H_T():
    for n in range(1,20):
        assert all([0 if x!='H' and x!='T' else x for x in throw_a_coin_proper
ly(n)])
# your code here


ipytest.run_tests()
```

```
unittest.case.FunctionTestCase (test_throw_a_coin_properly_verify_H_T) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.004s

OK
```