Lab Eight: Graphing Continuous Distributions, Verifying the CLT, Sampling Theory, Confidence Intervals

In this lab we will experiment with generating random variates according to the normal distribution and see how to display the results in a meaningful way. Then we will confirm the results predicted by the Central Limit Theorem and perform experiments in which we interpret the results using confidence intervals.

```python
In [1]:
# Jupyter notebook specific
from IPython.display import Image
from IPython.core.display import HTML
from IPython.display import display_html
from IPython.display import display
from IPython.display import Math
from IPython.display import Latex
from IPython.display import HTML

# General useful imports
import numpy as np
from numpy import arange, linspace, round, unique, concatenate, sort
import matplotlib.pyplot as plt
from numpy.random import random, randint, uniform, choice, binomial, geometric, poisson, normal, exponential
from scipy.stats import norm
import math
from collections import Counter
import pandas as pd
%matplotlib inline
import scipy as sc
import scipy.stats as sct
sc.version.full_version # 0.15.1

#a. Find f(x) for normal with mean loc and std dev scale
sct.norm.pdf(x=60, loc=60, scale=40) # 0.0099735570100358169

#b. Find P(X<50)
sct.norm.cdf(x=50, loc=60, scale=40) # 0.4012936743170763

#c. Find P(X>50)
sct.norm.sf(x=50, loc=60, scale=40) # 0.5987063256829237

#d. Find P(60<X<80)
sct.norm.cdf(x=80, loc=60, scale=40) - sct.norm.cdf(x=60, loc=60, scale=40)

#e. how much top most 5% expensive house cost at least? or find x where P(X>x) = 0.05
sct.norm.isf(q=0.05, loc=60, scale=40)

#f. how much top most 5% cheapest house cost at least? or find x where P(X<x) = 0.05
sct.norm.ppf(q=0.05, loc=60, scale=40)

# Round to 4 decimal places
def round4(x):
    return round(float(x)+0.0000000001,4)
```
Generating Variates from a Normal Distribution

Samples from a given distribution are often called "random variates" or just "variates" for short; to generate size random variates from a normal distribution with mean $\mu$ and standard deviation $\sigma$ we can use the numpy function

$$X = \text{normal}(\text{loc}=0,\text{scale}=1,\text{size}=1)$$

(Note that in this case, the normal is defined in terms of the standard deviation, and not the variance.)

Run the next cell several times to get a sense for how this function works

In [2]:

```python
In [2]: X = normal() # default is a standard normal with mean 0 and standard deviation 1
print(X)
Out[2]:

In [2]: X = normal(10,2) # defined by mean and standard deviation (NOT the variance)
print(X)
Out[2]:

In [2]: X = normal(66,3,size=10)
print(X)
Out[2]:
```

Graphing Normal Variates: A Problem

So generating normal variates is easy! What we are going to concern ourselves with in this first problem is now to graph a collection of such normal numbers. Here is the problem: since each value occurs (with high probability) only once, we can’t just create a histogram and convert it into a frequency distribution. Here is what happens if we do this, and graph it as a scatter plot against the theoretical distribution, as we did in previous labs:
In [3]: def display_normal_samples(mu, sigma, num_trials):
    fig, ax = plt.subplots(1,1,figsize=(12,6))
    plt.title('Analytical and Experimental Distributions for N(' +str(mu)+','+str(sigma)+')')
    plt.ylabel("P(X=k)")
    plt.xlabel("k in Range(X)"")
    # use normal(...) to generate random samples
    X = sorted(normal(mu,sigma,num_trials))
    # Now convert frequency counts into probabilities
    D = Counter(X)
    P = [D[k]/num_trials for k in X]
    plt.scatter(X,P)
    # Now generate the theoretical normal with the same mean and
    X2 = np.linspace(mu-sigma*3,mu+sigma*3,100)
    Y = [norm.pdf(x,mu,sigma) for x in X2]
    plt.plot(X2,Y)
    plt.show()

    # try setting the last number - the number of samples generated to 100 and 1000.
    display_normal_samples(66,3,10)

Graphing Normal Variates with Bins

You see the problem: since each floating point number (an approximation of a real number) is generated with high probability at most once, we can’t see the accumulation of samples that would indicate the probability. What to do? Well, you know that probabilities can only be calculated in continuous distributions using intervals, so we will create intervals by rounding our floating-point samples to put the variates into various-width bins, and then plot the probabilities of each bin.

So we must specify the width of each bin that we want to use as an interval to collect together our samples from the continuous distribution. Then we must “slot” each variate into its appropriate bin.

It makes sense to define the bin boundaries in terms of standard deviations from the mean, since we will be dealing with an unknown range of data; this bins can be made as wide or as narrow as we want, but will represent an interval defined in terms of the standard deviation sigma of the distribution.

We will graph the distribution in a range of at least 4 standard deviations of the mean, ignoring the rare occurrence of a variate outside this range.
In [4]: # Define the boundaries of bins with the specified width around the mean, 
   # to plus/minus at least 4 * sigma 
   
   # bin_width is in units of sigma, so bin_width = 0.1 means sigma/10 
   def makeBins(mu,sigma,bin_width):
       numBins = math.ceil(4/bin_width)
       bins = [mu+sigma*bin_width*x for x in range(-numBins,numBins+1)]
       return bins
   
   # Change the parameters several times to see the effect of this 
   print(makeBins(0,1,.5))

[-4.0, -3.5, -3.0, -2.5, -2.0, -1.5, -1.0, -0.5, 0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0]

Problem One: Generating and Graphing Normal Variates

How does the number of trials affect the fit of the data to the normal distribution?

Now let's do our previous experiment but trying various values for bin_width...
In [5]: def display_normal_samples_binned(mu, sigma, num_trials, bin_widths):
    
    fig, ax = plt.subplots(1, 1, figsize=(12, 6))
    plt.title('Analytical and Experimental Distributions for N(' + str(mu) + ',' + str(sigma) + ')')
    plt.ylabel(f(k)"
    plt.xlabel("k in Range(X)"
    # use normal(...) to generate random samples
    X = normal(mu, sigma, num_trials)
    plt.hist(X, bins=makeBins(mu, sigma, bin_widths), normed=True, edgecolor='k', alpha=0.5) # bins
    are of width 1/10**decimals
    # Now generate the theoretical normal with the same mean and
    X2 = np.linspace(mu-sigma*3, mu+sigma*3, 100)
    Y = [norm.pdf(x, mu, sigma) for x in X2]
    plt.plot(X2, Y)
    plt.show()

#try each of these and observe the effects
N = 10  # try 100, 1000, and 100000
display_normal_samples_binned(66, 3, N, 0.1)

![Analytical and Experimental Distributions for N(66,3)](image)

Clearly the data seems to fit the normal better when the number of trials increases...

**Affect of the bin width**

But now let’s think about the issue of precision, i.e., the width of the bins. Again, try each of the following and see what happens. You can see
that too-wide bins don’t give much information, but too-narrow bins don’t show how the data fits the normal distribution. There is a
relationship between the number of data points and the width of the bins.
In [6]: bin_width = 0.5  # try changing this to 0.5, 0.2, 0.1, 0.05, and 0.01
display_normal_samples_binned(66,3,1000,bin_width)

Problem 1 (a)

Clearly the “fit” with the normal curve depends on the width of the bins! For the following three examples, find a value for the indicated parameter which gives a good correspondence between the normal curve and the data.

In [7]: # Problem 1(a)
    : bin_width = 0.2  # experiment with this value 0.01, 0.05, 0.1, 0.15, etc. -- find the
    : # largest number which still gives a good fit
    : display_normal_samples_binned(0,10,3000,bin_width)  # don’t change this line

    #: Solution: Around 0.1 - 0.5 is probably right

Problem 1 (b)
In [8]:  # Problem 1(b)
        bin_width = 0.05  # experiment with this value -- find the largest number which still gives a good fit
        display_normal_samples_binned(0, 0.1, 10000, bin_width)  # don't change this line
        # Solution: 0.1 - 0.05 is probably about right

In [9]:  # Problem 1(c)
        bin_width = 0.05  # experiment with this value -- find the largest number which still gives a good fit
        display_normal_samples_binned(1000, 100, 10000, bin_width)  # don't change this line
        # Solution: 0.1 - 0.05 is probably right
Problem 1 (d)

What do you think is a good value for bin_width in general, assuming that num_trials is sufficient to give a reasonable approximation of the normal distribution

Solution: Probably about 0.1

Problem 2 (Verifying the CLT)

In this problem you will reproduce a result I showed in class, showing how the sample mean of samples from an exponential distribution produce a normal distribution following the prediction of the CLT.

The result is shown below (we have left the charts produced by the solution so you can see what you are aiming for).

Follow the instructions given in the comments in the code cell below, using code and the bin_width you investigated in Problem 1.
In [10]: # Use the numpy function exponential(1/lam) to generate random exponential variates with # rate parameter lam; this function simply generates n of them and returns the mean
def sampleMeanExponential(lam,n):
    return sum([exponential(1/lam) for i in range(n)]) / n

# Now display the result of generating num_trials values of the sample mean using # the above function, and graph the result, adapting the code from Problem 1 and using # an appropriate bin_width to demonstrate the results most clearly
def display_sample_mean_exponential(lam,n,num_trials,bin_widths):
    fig, ax = plt.subplots(1,1,figsize=(12,6))
    plt.title('Exp('+str(lam)+'): Distribution of Sample Mean with n = '+str(n))
    plt.ylabel("f(x)")
    plt.xlabel("k in Range(X)"
    ax.set_xlim(-5,25)
    ax.set_ylim(0,0.4)
    mu = 1/lam
    sigma = (1/lam) / (n**0.5)

    # use exponential to generate random samples
    X = [sampleMeanExponential(lam,n) for i in range(num_trials)]
    plt.hist(X,bins=makeBins(mu,sigma,bin_widths), normed=True,edgecolor='k',alpha=0.5) # bins are of width 1/10**decimals

    # Now generate the theoretical normal for sample mean with std dev sigma/sqrt(n)
    X2 = np.linspace(mu-sigma*3,mu+sigma*3,100)
    Y = [norm.pdf(x,mu,sigma) for x in X2]
    plt.plot(X2,Y)
    plt.show()

lam = 0.1
num_trials = 10000
display_sample_mean_exponential(lam,1,num_trials,0.1)
display_sample_mean_exponential(lam,2,num_trials,0.1)
display_sample_mean_exponential(lam,5,num_trials,0.1)
display_sample_mean_exponential(lam,10,num_trials,0.1)
display_sample_mean_exponential(lam,30,num_trials,0.1)
display_sample_mean_exponential(lam,100,num_trials,0.1)
Exp(0.1): Distribution of Sample Mean with n = 1

Exp(0.1): Distribution of Sample Mean with n = 2

Exp(0.1): Distribution of Sample Mean with n = 5
Exp(0.1): Distribution of Sample Mean with n = 10

Exp(0.1): Distribution of Sample Mean with n = 30

Exp(0.1): Distribution of Sample Mean with n = 100
Problem 3

The GPAs for 4897 individuals from an institution of higher education in the northeastern United States is given in StudentGPADatadata.csv (the first column gives the Gender, and the second the GPA—you only need the GPA data). For your information, the histogram shows that it is approximately normal, except for the limit at 4.0:

![Histogram of GPAs](image)

We will use this data (just the list of GPAs) for exploring the various ideas presented in lecture about sampling theory. You must submit your Python code as part of the homework submission, and also do the following parts of the problem.

(a) Calculate the mean and standard deviation of this population and provide them in your solution to 4 decimal places (use the function round4(.) provided above).

(b) Write a function getSample(n) to generate ONE random sample of n = 30 samples, using random.choice(L,n), which takes a list L and returns n random values, and calculate the sample mean with a function Xbar(n). Using the techniques developed in lecture, report on your estimation of the population mean using a confidence interval for 95% (NOT 95.14....%) confidence. Use your results from (a) to calculate the confidence interval.

(c) Now run an experiment, generating least $10^5$ sample means using Xbar(30), and determine how many of them were within the error margin calculated in (b) around the true mean determined in (a); print this out as a probability (i.e., the percentage that were within the error margin).

This may vary but you should usually get a probability close to 95%. Running more trials will improve the accuracy.....

```python
# If BUData.csv is in the same directory as this notebook, this will read in
# the gpa data into a list
studs = pd.read_csv('BUData.csv')
gpaList = studs['GPA'].tolist()

# Part (a)
(a) mean = 3.1729    sigma = 0.4934

# Part (b) and (c)
standardError = sigma / (30**0.5)
(b) Result is 2.991 +/- 0.1765
    or: [ 2.8145 .. 3.1675]
(c) Probability = 0.94942
```
Problem 4

In this problem we will test "Bessel's Correction" for the sample variance (using n-1 rather then n in the denominator), using our GPA data from problem 2 above. This concept will be discussed on Tuesday 11/7 in lecture.

(a) Write a function sampleVar(n) which will generate a random sample of size n from the gpa data in Problem 2, and return the sample variance (which will of course involve calculating the sample mean), using n in the denominator (so we are returning the "population variance"). Then write a function testPopulationVariance(M,n) which will generate M sample variances using sampleVar(n), and return of the mean of these M values. Run this for M = 1000 and n = 2, 3, ..., 50 and store it in a list PV.

(b) Do the same as (a), but with a function testSampleVariance(M,n) which uses a denominator of n-1 and store it in a list SV.

Note: (a) and (b) require no output, just writing the code.

(c) Graph these results with the actual variance calculated directly from the entire data set (i.e., the x axis is the values 2, 3, ..., 50, and the three curves are the (constant) value of the true population variance (a list [v, v, ...., v] where v is the actual population variance calculated in problem 2 (a)), and the y values in PV and SV). How do these two estimators compare as n approaches 50?

In [33]:

```python
def getSample(n,list):
    return np.random.choice(list,n)

print("Problem 4")
```

Problem 4
(a) & (b) require no output
(c)