EE131A Class Project

Alon Krauthammer

November 28, 2020

All the code written for this project is downloadable as python notebooks from my github:

https://github.com/krauthammera/ee131a˙finalproj

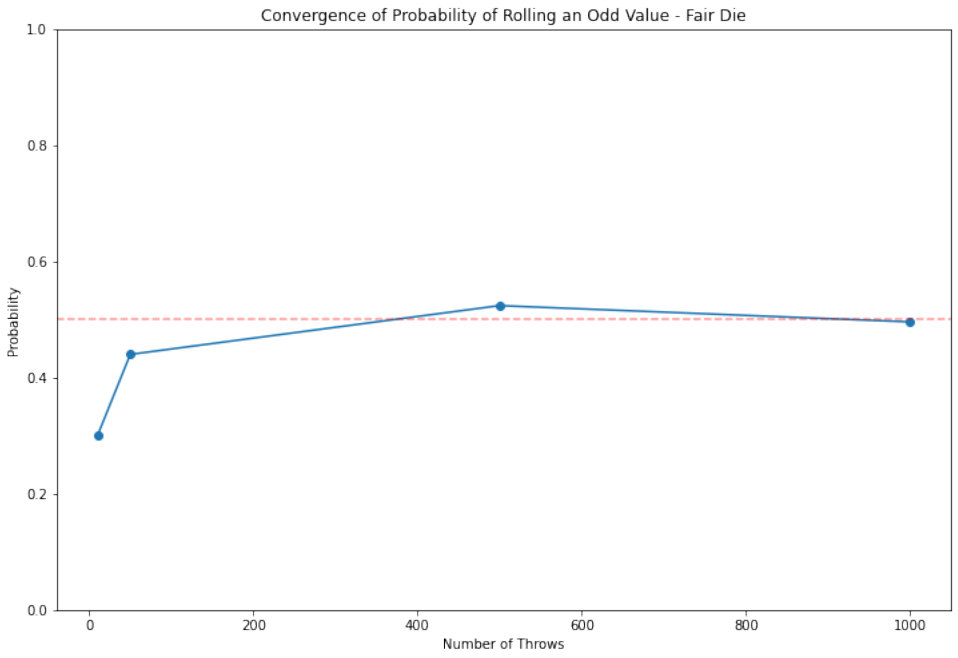
1. Tossing a fair and unfair die. Suppose you have a 4-sided die, with sides numbered 1, 2, 3, and 4.

   (a) Write a MATLAB program to simulate the tossing of a 4-sided fair die, for t=10, 50, 100, 500 and 1000 tosses. Based on the simulation, what is the estimated probability of obtaining an odd number?

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

#Tossing a fair and unfair die, 4 sides
die = np.arange(1,5)
probs_unbiased = [(1/4)]*4
probs_biased = [(1/6), (1/3)]*2
n_throws = [10, 50, 500, 1000]

odd_probs = []

for t in n_throws:
    #throw the fair die t times
    outcomes = np.random.choice(die, t, p=probs_unbiased)

    unique, counts = np.unique(outcomes, return_counts=True)
    counts_dict = dict(zip(unique, counts))

    #check if all values are in unique, if not, add zero count:
    for value in die:
        if value in unique:
            continue
      else:
            counts_dict[value] = 0

    counts = counts_dict.values()
    counts_series = pd.Series(counts)

    plt.figure(figsize=(12, 8))
    ax = counts_series.plot(kind='bar')
    ax.set_title('Fair Die Roll Frequency, t={}'.format(t))
    ax.set_xlabel('Value Rolled')
    ax.set_ylabel('Frequency')
    ax.set_xticklabels(die)
```

```
        #find P(odd)
        total_odds = counts_dict[1] + counts_dict[3]
        sim_prob_odd = total_odds/t
        odd_probs.append(sim_prob_odd)

    print("Simulation probability of rolling an odd value converges to
        {}".format(odd_probs[-1]))


    plt.figure(figsize=(12, 8))
    plt.plot(np.arange(1, len(n_throws) + 1), odd_probs, '-o')
    plt.ylim(0, 1)
    plt.axhline(y=0.5, color='r', linestyle='dashed', alpha = 0.4)
    plt.title("Convergence of Probability of Rolling an Odd Value - Fair Die")
```

The plot resulting from this simulation is below:



As can be seen in the plot, the simulation converged to a proportion of odd results of 0.496.

(b) Suppose X is a random variable denoting the outcome of a die toss. Based on the mathematical analysis, what is the probability that X has odd value?

The sample space of this experiment is

$$S = \{1, 2, 3, 4\}$$

Let event O be the event that the outcome of the die roll is an odd number. Then $O = 1, 3$. Because this is a fair die, each of the 4 outcomes is equiprobable, so $P[O] = \frac{1}{4} + \frac{1}{4} = \frac{1}{2}$.

(c) Refer back to part (a). Does it agree with the theoretical result in (b)?

Yes, this agrees with the simulation, because the simulated value is very close to the theoretical result.

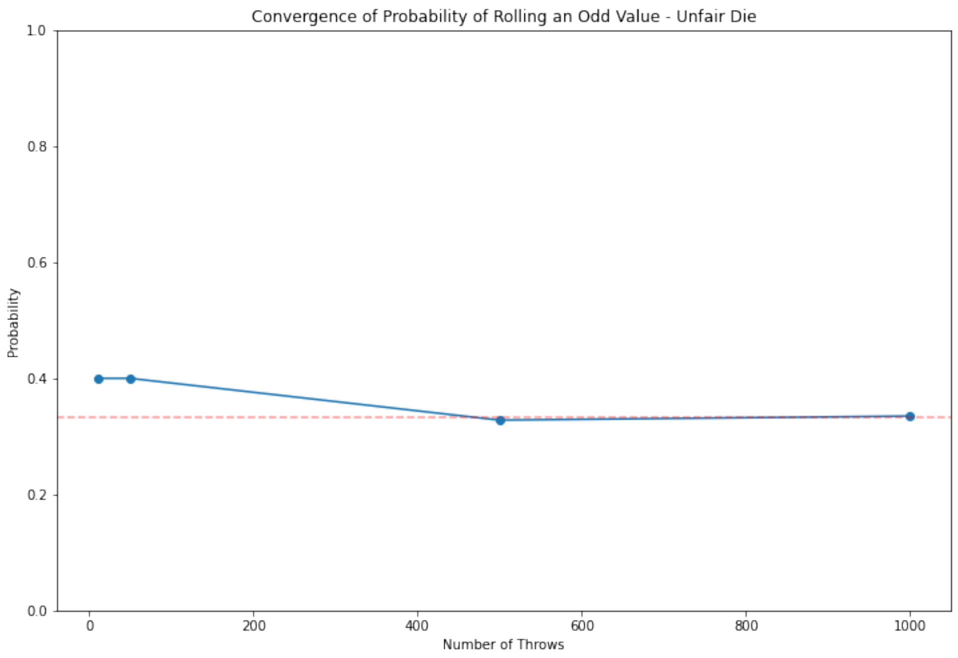
(d) Repeat parts (a), (b), and (c) if even sides are twice as likely as odd sides, and both even sides are equiprobable and both odd sides are equiprobable.

Whether the die is fair or unfair does not change the sample space. But now, the probability of the outcomes $2, 4$ are $\frac{1}{3}$ each, and the probability of the outcomes $1, 3$ are $\frac{1}{6}$ each. As a result, $P[O] = \frac{1}{6} + \frac{1}{6} = \frac{1}{3}$.

The solution to this part is the same as that of part (a), but I used the `probs_biased` array as an input to the `np.random.choice()` function, as seen below:

```
probs_biased = [(1/6), (1/3)]*2

for t in n_throws:
    #throw the unfair die t times
    outcomes = np.random.choice(die, t, p=probs_biased)
```

The below plot shows the proportion of rolls resulting in an odd value. For 1000 throws, this value converged to 0.335 in my simulation, which is very close to the theoretical value of $\frac{1}{3}$.



2. Consider the binary symmetric channel (BSC) and repetition code described in Lecture 6. The BSC has probability p to flip a bit sent through this channel, with $0 < p < 0.5$. To transmit information reliably through this channel, we use the Nrepetition code which repeats the bit we want to transmit N times (known as encoding), and then transmits this N-bit sequence through the BSC. Upon receiving this sequence, the receiver uses the majority rule to decide the encoded bit (this is decoding). For example, for $N = 5$, suppose we want to transmit a 0. We first encode it into the sequence *00000*, using the 5-repetition code. Suppose some bits got flipped during transmission over this BSC, so that the receiver observes the sequence *00011*. In this example, using majority vote, the receiver would determine that the encoded bit for transmission was a 0. In this question, assume that the bits encoded for transmission are equiprobable.

   (a) Write a program to simulate the BSC. Also write programs for encoding (repetition code) and decoding (majority rule). We choose $N = 1, 3, 5, 7$ for this problem. For a given N, simulate 20000 instances of sending a bit through the BSC using repetition code and record the fraction of error, i.e., empirical error probability, for $p = 0.01, 0.02, \ldots, 0.1$. Plot the empirical error probability against p for each N. You may want to use a semi-log for the y axis.

   First, I imported some packages and defined some constants:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from bitarray import bitarray
from collections import Counter
from scipy.special import comb

import seaborn as sns
sns.set(color_codes=True)
# settings for seaborn plot sizes
sns.set(rc={'figure.figsize':(20,10)})

p_errs = np.arange(0.01, 0.11, 0.01)
p_errs_2 = np.arange(0.1, 1.0, 0.05)
N_vals = [1, 3, 5, 7]
n_trials = 20000
```

   Next, I defined the functions required to perform this simulation:

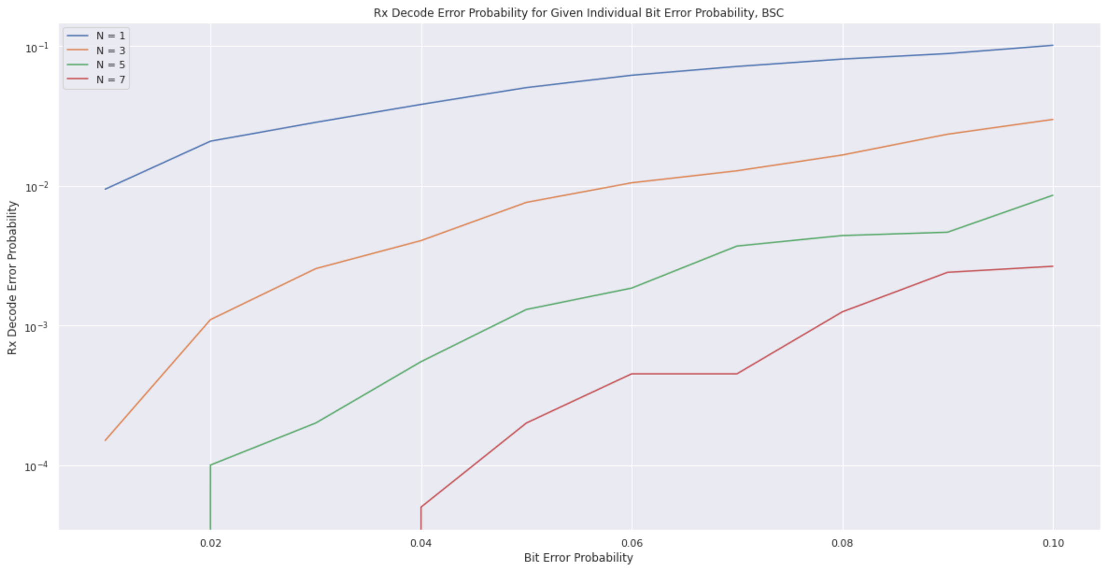
```
def get_bit():
   return np.random.choice(2)

def encode(bit, N):
   #bit will be either 0 or 1
   encoded = bitarray([bit]*N)
   return encoded

def channelize(tx_vector, p_err):
   tx_channelized = bitarray(len(tx_vector))
   for ix, bit in enumerate(tx_vector):
      bit = bit
      value = np.random.random_sample()
      if value < p_err:
         #flip bit
         tx_channelized[ix] = not bit
      else:
         tx_channelized[ix] = bit
   return tx_channelized
```

Finally, I carried out the BSC simulation as described. I've left in some debugging print statements as comments, which can be uncommented in order to see the structure of the program:

```
df_cols = ['N', 'Pe', 'Original', 'Channelized', 'Decision', 'Correct']

error_count = 0
rows = []

for N in N_vals:
   print("Computing for N = {}".format(N))
   for p_err in p_errs:
      print("   Computing for Pe = {}".format(p_err))
      for i in range(0, n_trials):
         #print("       trial #{}".format(i))
         bit = get_bit()
         rep_coded = encode(bit, N)
         #print("          Encoded bitstream: {}".format(rep_coded))
         to_tx = channelize(rep_coded, p_err)
         #print("          After applying channel: {}".format(to_tx))
         num0s = Counter(to_tx)[False]
         num1s = Counter(to_tx)[True]
         #print("          0s in Rx: {}".format(num0s))
         #print("          1s in Rx: {}".format(num1s))

         if (num0s > num1s):
            rx = 0
         else:
            rx = 1

         if (rx != bit):
            #print("ERROR")
            error_count += 1
            row = [N, p_err, rep_coded, to_tx, rx, False]
            #errs_for_given_p += 1
         else:
            row = [N, p_err, rep_coded, to_tx, rx, True]

         rows.append(row)


print("Total errors in experiment: {}".format(error_count))

#place in a pandas DF in order to calculate p_error for each run easily
df = pd.DataFrame(rows, columns=df_cols)

experiment_errors = []

#grab a separate pandas df for each N value
for N in N_vals:
   df_N = df[df["N"]==N]

   #initialize the errors list
   errors_for_N = []

   #for each p_err, calculate the p_err for the given N
   for p_err in p_errs:
      #get the total set of trials for a specific N and Pe
      df_N_perr = df_N[df_N["Pe"]==p_err]
```

```
        n_errs = df_N_perr[df_N_perr["Correct"] == False].shape[0]
        p_error = n_errs/df_N_perr.shape[0]
        errors_for_N.append(p_error)

    experiment_errors.append(errors_for_N)

#generate another DF for the final dataset, p_error for each value of Pe and N
exp_errs_df = pd.DataFrame(np.array(experiment_errors).T.tolist(),
     columns=[str(N) for N in N_vals])
exp_errs_df["P"] = p_errs

#finally, plot this thing
plt.figure()

for N in N_vals:
    sns.lineplot(data=exp_errs_df, x="P", y=str(N), label="N = {}".format(N))

plt.xlabel("Bit Error Probability")
plt.ylabel("Rx Decode Error Probability")
plt.title("Rx Decode Error Probability for Given Individual Bit Error
     Probability, BSC")
plt.yscale('log')
plt.legend()
```

This is the plot generated by the above program:



For $N = 5$ and $N = 7$, the p_{error} for small values of p results in zero RX decode errors for this simulation, which is why there is a discontinuity in those plot lines.

(b) Find the theoretical error probability of sending a bit through BSC using repeti-tion code, parameterized by N and p. For $N = 1, 3, 5, 7$ and $p = 0.01, 0.02, \dots, 0.1$, generate plots similar to (a) using your theoretical results.

For a given N, a majority of the bits must be erroneous for there to be a decode error. As a result, we must calculate the number of ways we can have $floor(\frac{N}{2}) + 1, \dots, N$ bit errors and sum these terms. This can be expressed as:

$$\sum_{k=floor(\frac{N}{2})+1}^{N} \binom{N}{k} P_e^k (1 - P_e)^{N-k}$$

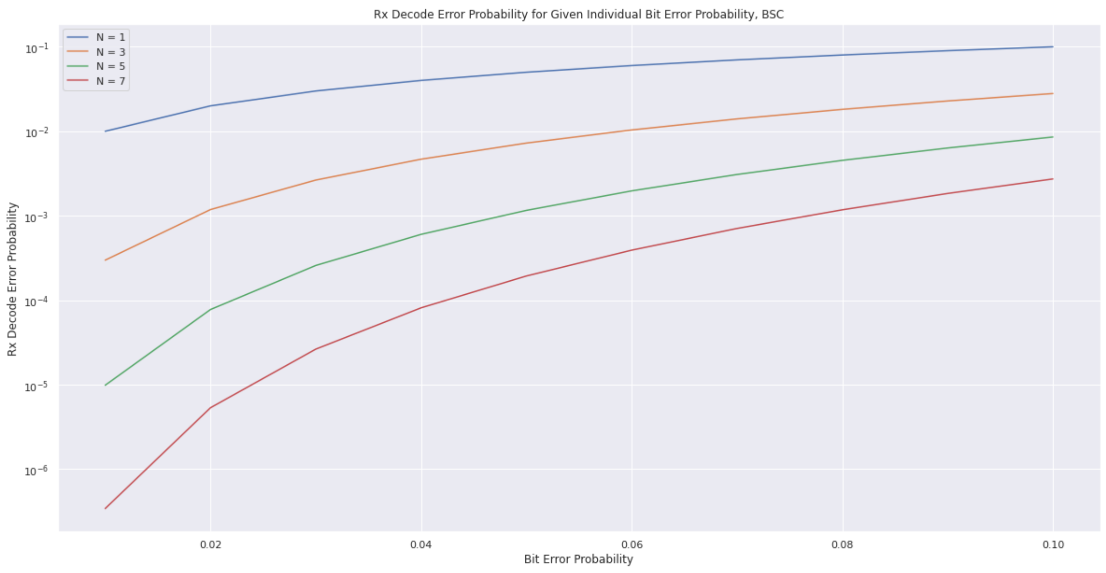
Below is the code which implements the above expression:

```
experiment = {}

for N in N_vals:

   p_errors = []

   for Pe in p_errs:

      #print("   {}".format(Pe))
      summation = 0

      for k in range(int(N/2)+1, N+1):
         #print("      {}".format(k))
         term = comb(N, k)*(Pe**k)*((1-Pe)**(N-k))
         summation += term

      p_errors.append(summation)

   experiment[str(N)] = p_errors

   #convert dictionary into dataframe
   theoretical_df = pd.DataFrame(experiment)
   theoretical_df["P"] = p_errs
```

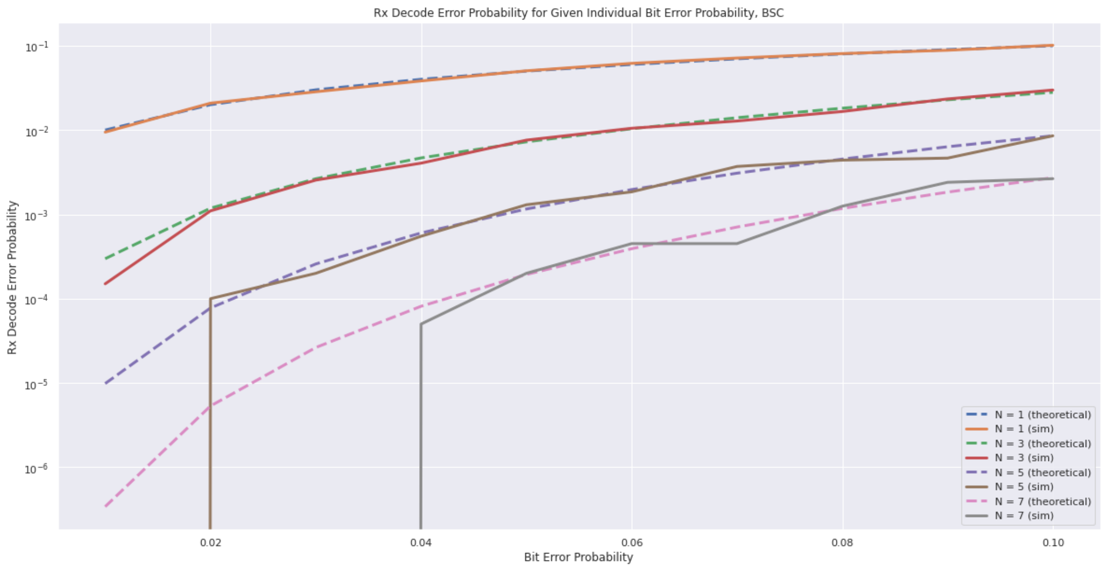
Here is the code used to plot the generated theoretical values:

```
plt.figure()

for N in N_vals:
   sns.lineplot(data=theoretical_df, x="P", y=str(N), label="N = {}".format(N))

plt.xlabel("Bit Error Probability")
plt.ylabel("Rx Decode Error Probability")
plt.title("Rx Decode Error Probability for Given Individual Bit Error
    Probability, BSC")
plt.yscale('log')
plt.legend()
```

Below is the plot:

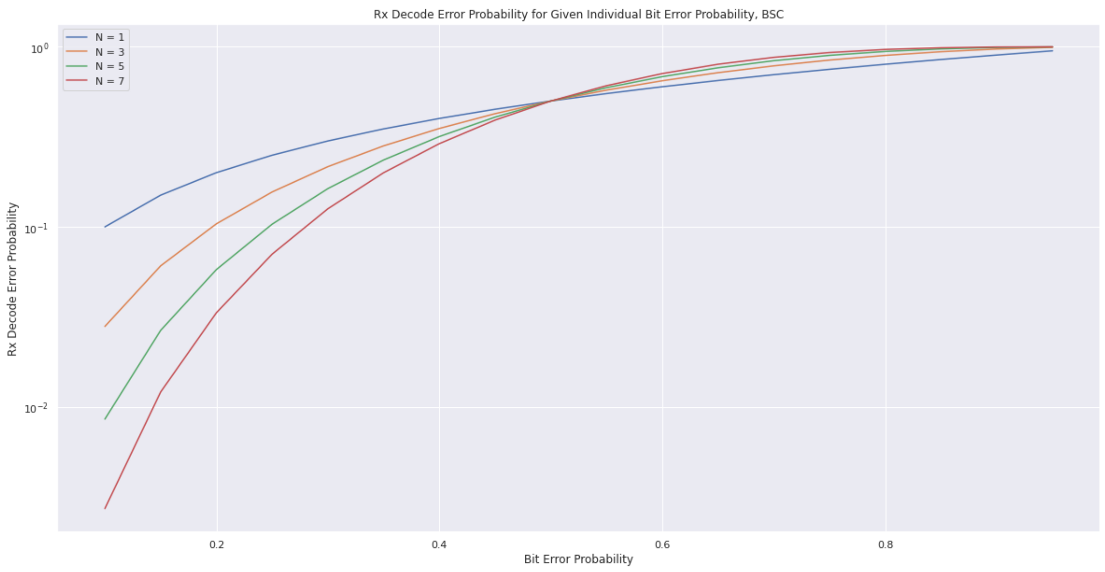


Also, here is a plot combining the simulation results from (a) with the theoretical results:



(c) What kind of channel would be a BSC with $p = 0.5$? What change, if any, would you employ in your system design if you knew that $0.5 < p < 1$?

Below is a plot showing what happens as p approaches (and passes) 0.5:



This shows that for $p = 0.5$, it doesn't matter how much repetition coding is used - the resulting channel makes the receiver generate a random value from the set $[0, 1]$.

It can also be seen that the error rate is actually greater than 0.5 with $p > 0.5$. As a result, at the receiver, it would be beneficial to take the opposite of the majority decoding result - "minority decoding."

3. *Naive Bayes Classifier.* On April 15, 1912, the largest passenger liner ever made collided with an iceberg during her maiden voyage. When the Titanic sank it killed 1502 out of 2224 passengers and crew. This sensational tragedy shocked the international community and led to better safety regulations for ships. One of the reasons that the shipwreck resulted in such a loss of life was that there were not enough lifeboats for the passengers and crew. Although there was some element of luck involved in surviving the sinking, some groups of people were more likely to survive than others.

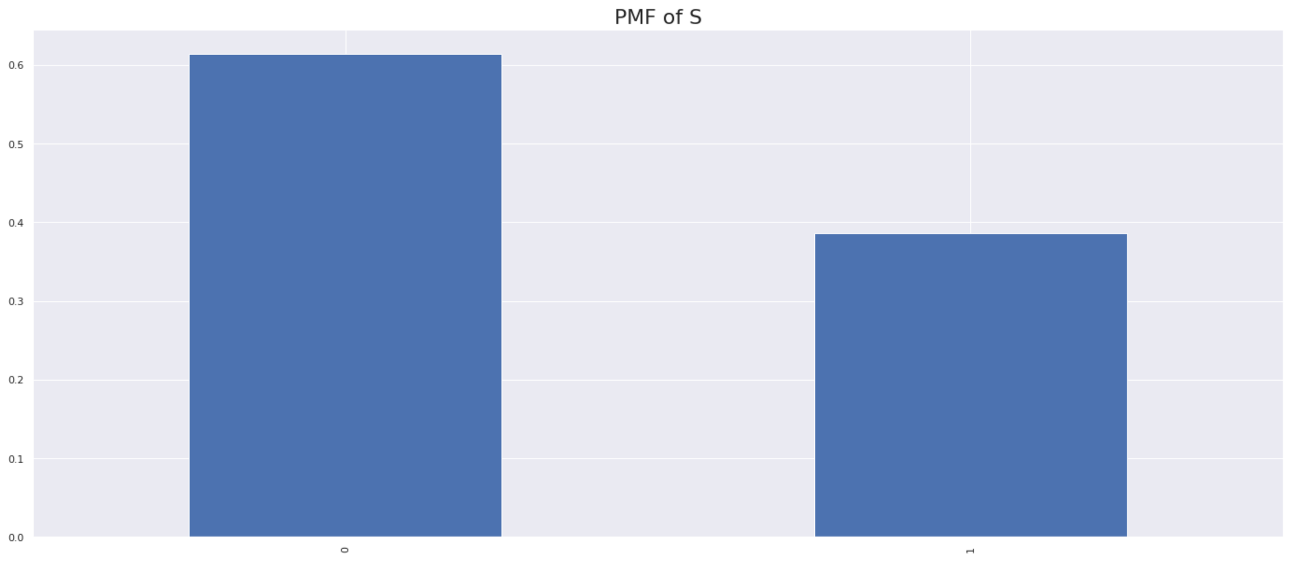
   Some demographic data are collected for 887 passengers and are provided in *titanic.csv*.The price class, gender (1 for Male and 0 for Female) and age are recorded for each survived (1) or killed (0) passenger. We use random variables S, C, G, and A for survival status, price class, gender and age, respectively. In this problem, we are going to build a popular Naive Bayes Classifier to predict S given C, G, and A.

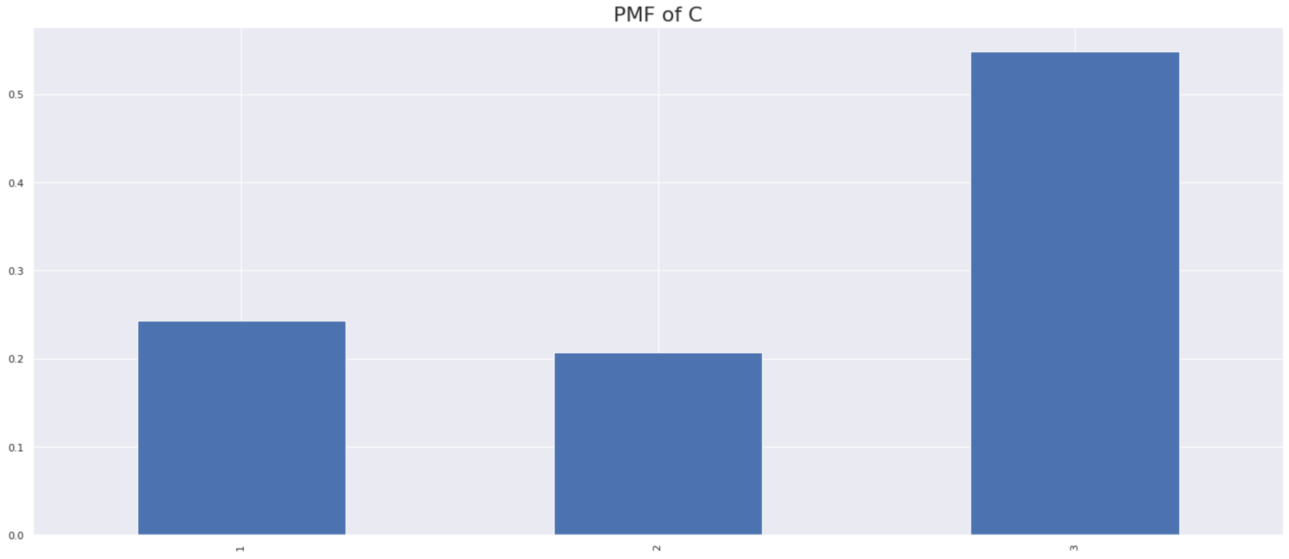
   (a) Estimate the PMFs for S, C, G, and A by finding the fraction of each realization of these random variables among all data. Plot these PMFs.

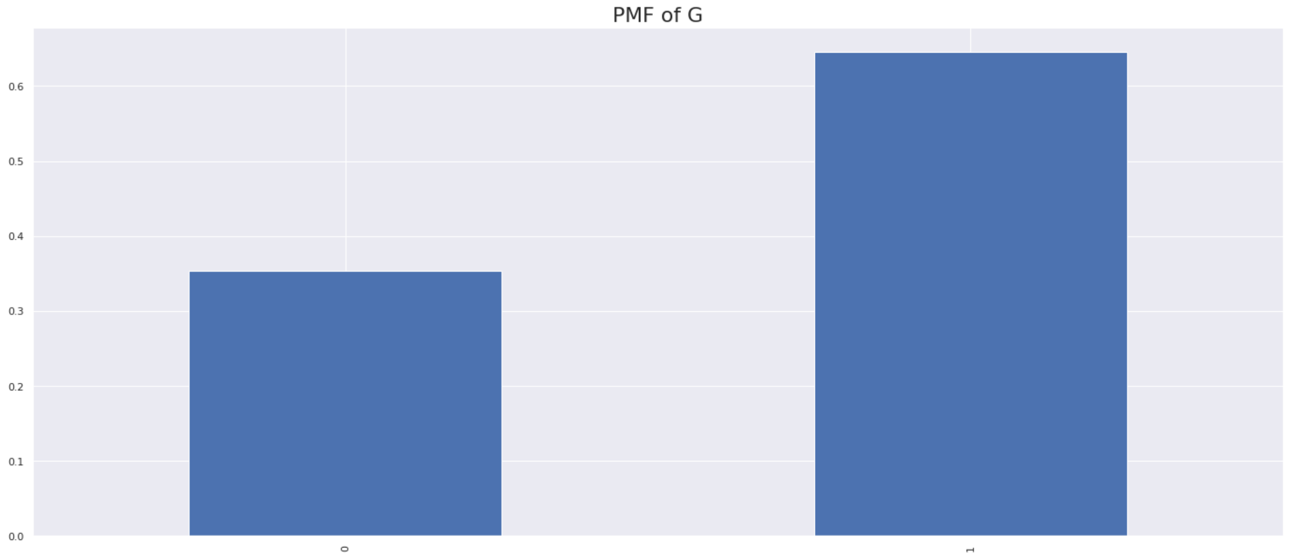
   Below is the code used to generate these PMFs:

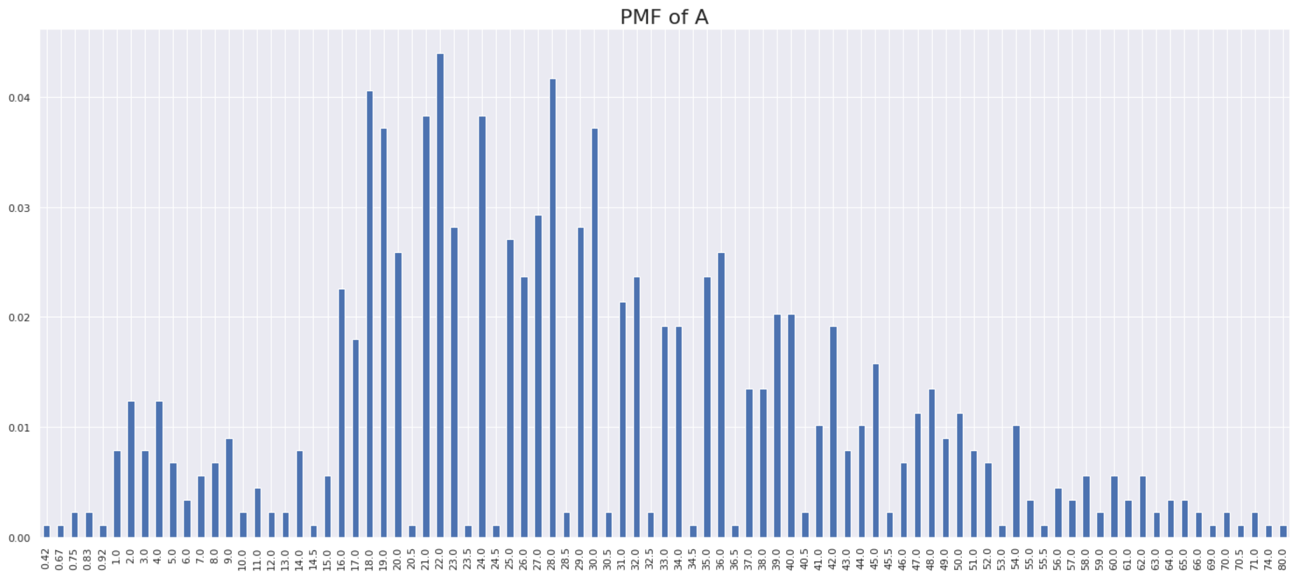
```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

import seaborn as sns
sns.set(color_codes=True)
# settings for seaborn plot sizes
sns.set(rc={'figure.figsize':(24,10)})

filename = 'titanic.csv'
rv_names = ['S', 'C', 'G', 'A']

df = pd.read_csv('titanic.csv')

for i in range(0, df.shape[1]):
   col = df.iloc[:, i]
   pmf = col.value_counts().sort_index() / len(col)
   fig = plt.figure()
   #fig.suptitle("PMF of {}".format(rv_names[i]), fontsize=24)
   ax = fig.add_subplot(111)
   ax.set_title("PMF of {}".format(rv_names[i]), fontdict={'fontsize': 22,
       'fontweight': 'medium'})
   pmf.plot(kind="bar")
```

   Below are the plots generated by the above code:


PMF of S
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0
1


PMF of C
0.0
0.1
0.2
0.3
0.4
0.5
1
2
3


PMF of G
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0
1



(b) We are interested in how C, G, and A affect S, in the context of the *Naive Bayes Classifier*; the first step is to estimate the conditional PMF conditioning on the outcome of interest, i.e., survival or not. Estimate and plot the conditional PMFs for C, G, and A conditioned on S.

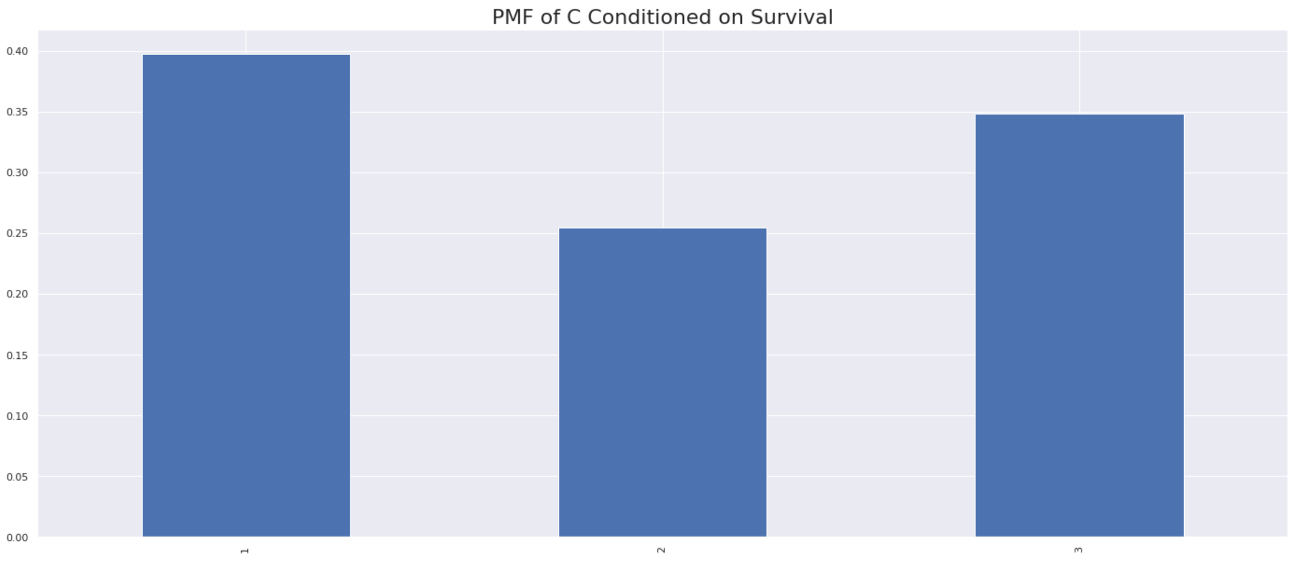
In order to do this part, I generated two separate dataframes for survival and death:

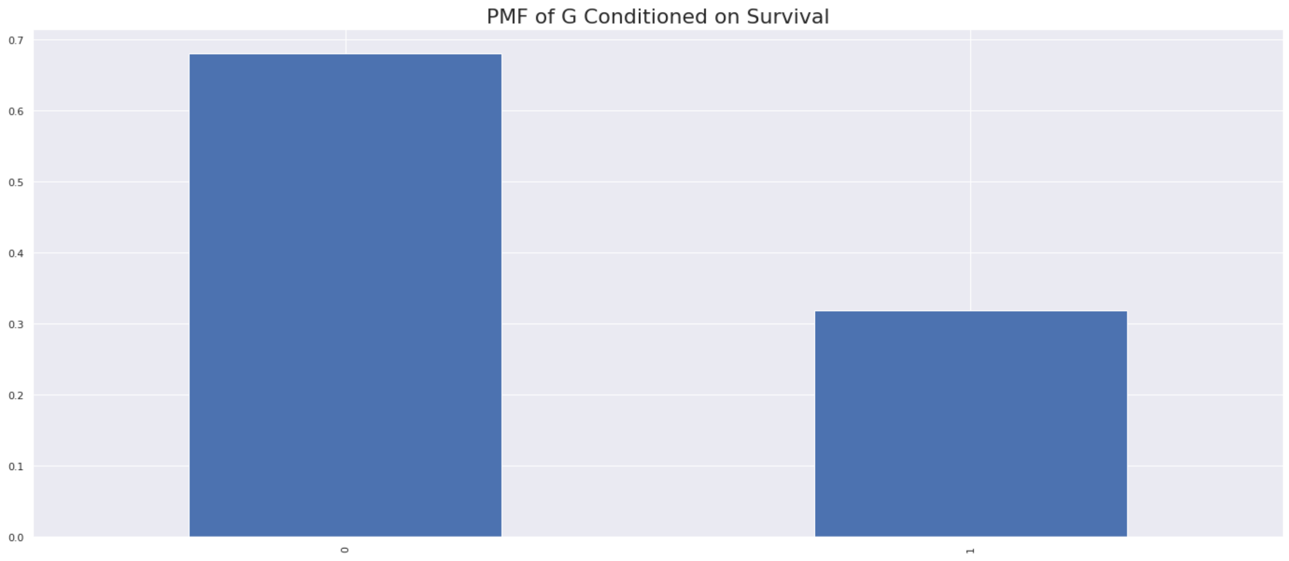
```
df_s1 = df[df["Survived"] == 1]
df_s0 = df[df["Survived"] == 0]
```

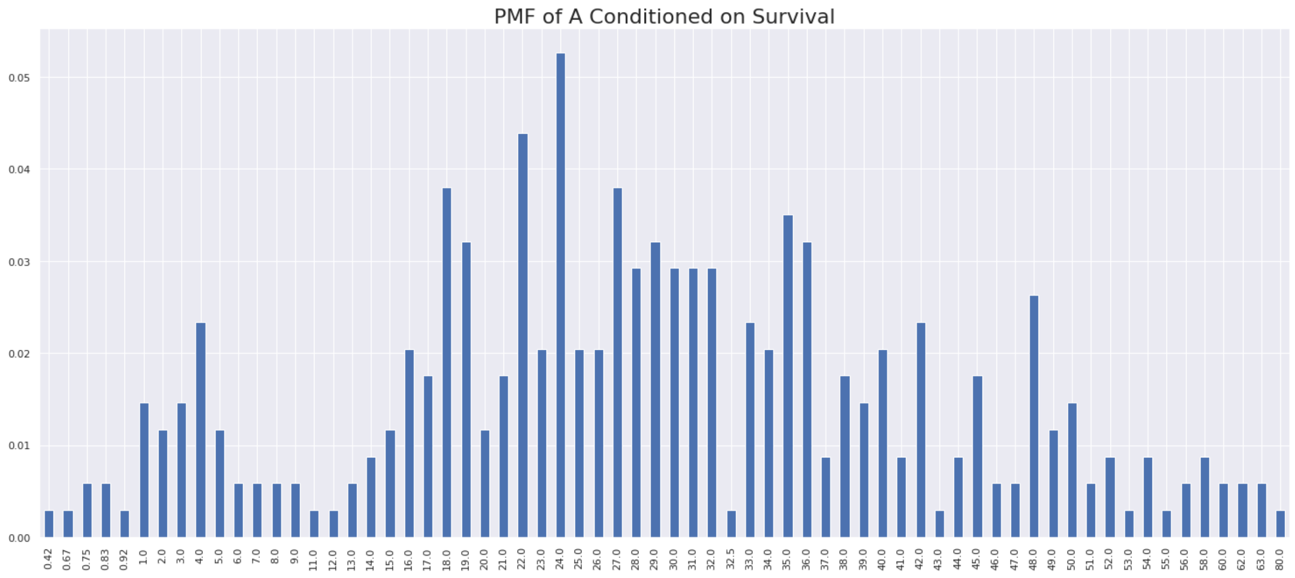
Once I had done that, I simply generated a plot for each of the 6 conditional PMFs, similar to the process in (a):

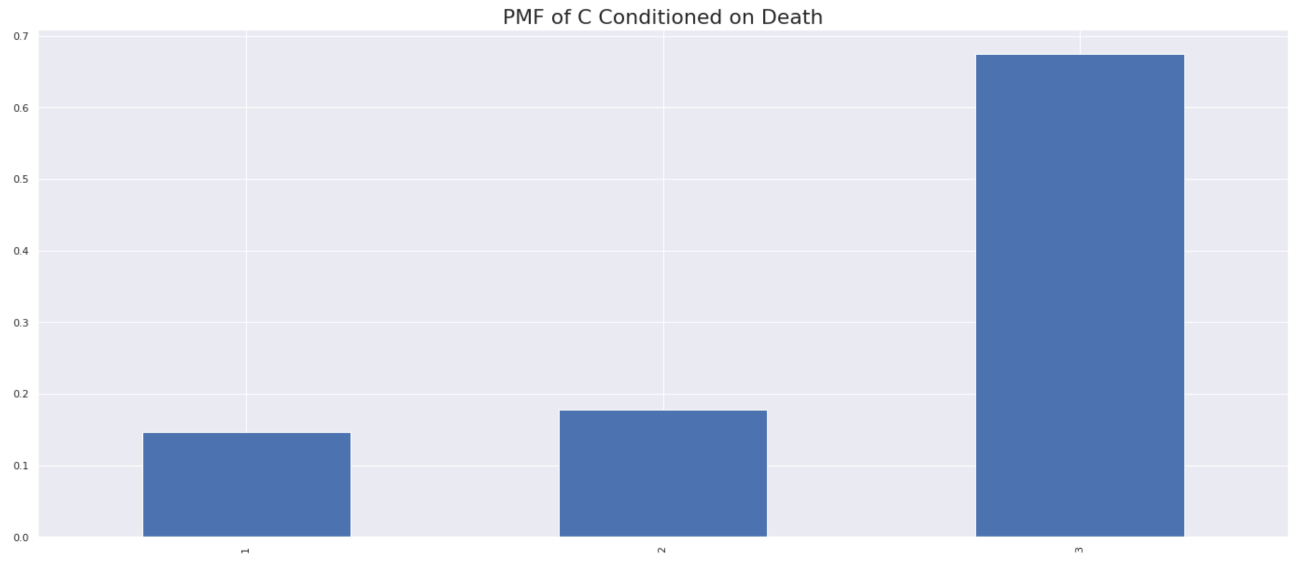
```
for ix, col_name in enumerate(["Pclass", "Sex", "Age"]):

   #get the individual data
   col = df_s1[col_name]
   col2 = df_s0[col_name]

   #get the conditonal PMFs for RV
   pmf_cond = col.value_counts().sort_index() / len(col)
   pmf_cond2 = col2.value_counts().sort_index() / len(col2)

   #figure for PMF of RV conditioned on survival
   fig = plt.figure()
   ax = fig.add_subplot(111)
   ax.set_title("PMF of {} Conditioned on Survival".format(rv_names[ix+1]),
       fontdict={'fontsize': 22, 'fontweight': 'medium'})
   pmf_cond.plot(kind="bar")

   #figure for PMF of RV conditioned on death
   fig2 = plt.figure()
   ax2 = fig2.add_subplot(111)
   ax2.set_title("PMF of {} Conditioned on Death".format(rv_names[ix+1]),
       fontdict={'fontsize': 22, 'fontweight': 'medium'})
   pmf_cond2.plot(kind="bar")
```

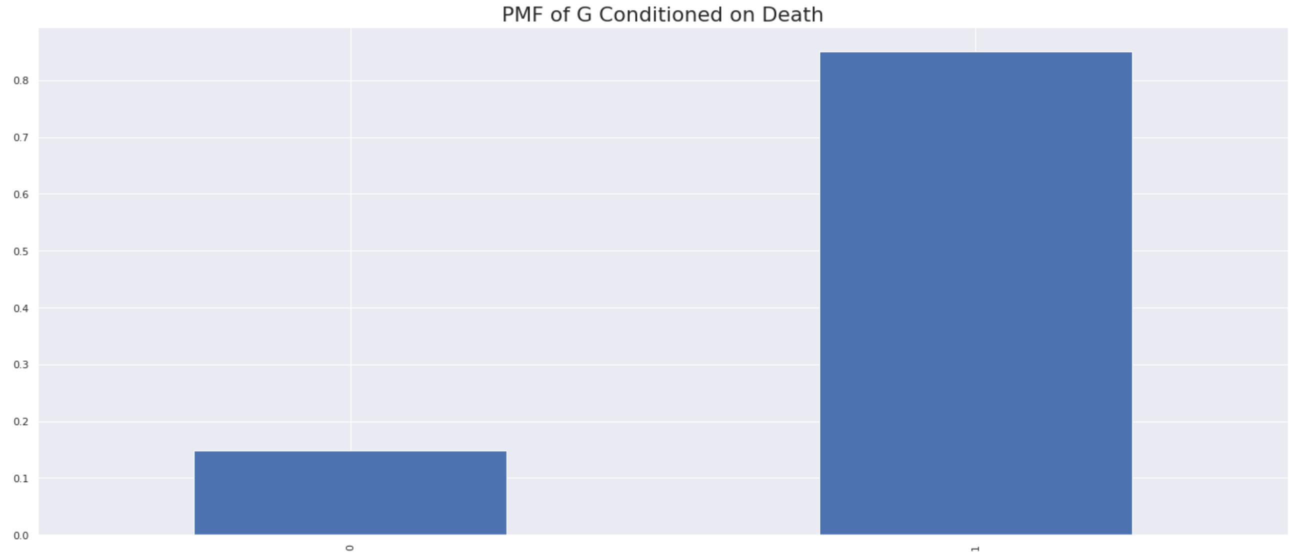
Below are the plots generated by this notebook cell:

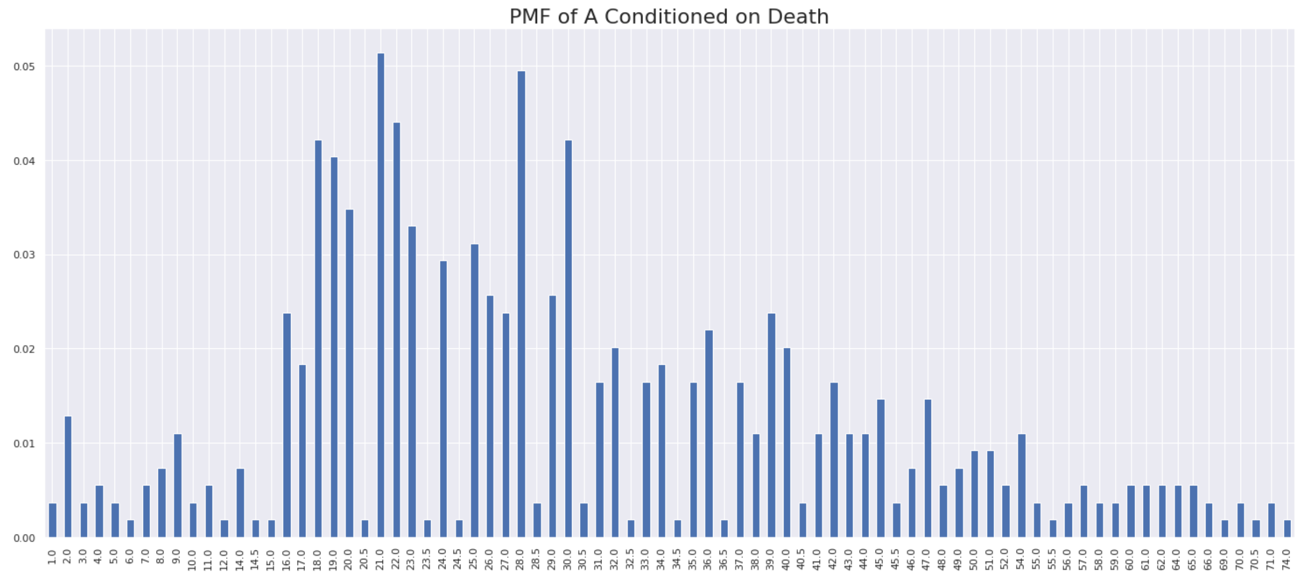

PMF of C Conditioned on Survival
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
0.40
1
2
3


PMF of G Conditioned on Survival
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0
1


PMF of A Conditioned on Survival
0.00
0.01
0.02
0.03
0.04
0.05


PMF of C Conditioned on Death
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
1
2
3


PMF of G Conditioned on Death
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0
1


PMF of A Conditioned on Death
0.00
0.01
0.02
0.03
0.04
0.05

(c) In the *Naive Bayes Classifier*, we use the conditional independence assumption. For example, if C, G, and A are conditionally independent on S, then

$$P(C, G, A|S = 0) = (C|S = 0)P(G|S = 0)P(A|S = 0) \textit{ and } P(C, G, A|S = 1) = (C|S = 1)P(G|S = 1)P(A|S = 1)$$

Using this assumption, compute

$$P(S = 0, C = 1, G = 0, A \leq 40) \textit{ and } P(S = 1, C = 1, G = 0, A \leq 40)$$

If we treat $C = 1, G = 0, A \leq 40$ as one event and $S = 0$ or $S = 1$ as another event, we can use the definition of conditional probability to write, for example:

$$P(S = 0, C = 1, G = 0, A \leq 40) = P(S = 0)P(C = 1, G = 0, A \leq 40|S = 0) \tag{1}$$

Then, taking the right side of equation (1) and using the the conditional independence assumption, we get the following relationships:

$$\begin{aligned} P(S = 0, C = 1, G = 0, A \leq 40) &= P(S = 0)P(C = 1, G = 0, A \leq 40|S = 0) \\ &= P(S = 0)[P(C = 1|S = 0)P(G = 0|S = 0)P(A \leq 40|S = 0)] \end{aligned} \tag{2}$$

$$\begin{aligned} P(S = 1, C = 1, G = 0, A \leq 40) &= P(S = 1)P(C = 1, G = 0, A \leq 40|S = 1) \\ &= P(S = 1)[P(C = 1|S = 1)P(G = 0|S = 1)P(A \leq 40|S = 1)] \end{aligned} \tag{3}$$

With these expressions, the PMFs of S computed in (a), and the conditional PMFs of the other RVs computed in (b), we can implement the calculations in Python below:

```python
pmf_survival = df["Survived"].value_counts().sort_index() /
    len(df["Survived"])

pS0 = pmf_survival[0]
pS1 = pmf_survival[1]

pmf_s1_class = df_s1["Pclass"].value_counts().sort_index() /
    len(df_s1["Pclass"])
pmf_s0_class = df_s0["Pclass"].value_counts().sort_index() /
    len(df_s0["Pclass"])

pC1S1 = pmf_s1_class[1]
pC1S0 = pmf_s0_class[1]

pmf_s1_gender = df_s1["Sex"].value_counts().sort_index() / len(df_s1["Sex"])
pmf_s0_gender = df_s0["Sex"].value_counts().sort_index() / len(df_s0["Sex"])

pG0S1 = pmf_s1_gender[0]
pG0S0 = pmf_s0_gender[0]

pmf_s1_age = df_s1["Age"].value_counts().sort_index() / len(df_s1["Age"])
pmf_s0_age = df_s0["Age"].value_counts().sort_index() / len(df_s0["Age"])

pA40S1 = pmf_s1_age.loc[pmf_s1_age.index <= 40.0].sum()
pA40S0 = pmf_s0_age.loc[pmf_s0_age.index <= 40.0].sum()

#P(C, G, A|S = 0) = P(C|S = 0)P(G|S = 0)P(A|S = 0)
pC1G0A40gS0 = pC1S0*pG0S0*pA40S0
pS0C1G0A40 = pS0*pC1G0A40gS0
print(pS0C1G0A40)
print(pC1G0A40gS0)

#P(C, G, A|S = 1) = P(C|S = 1)P(G|S = 1)P(A|S = 1)
pC1G0A40gS1 = pC1S1*pG0S1*pA40S1
pS1C1G0A40 = pS1*pC1G0A40gS1
print(pS1C1G0A40)
print(pC1G0A40gS1)
```

Following the above, we obtain the following:

$$P(S = 0, C = 1, G = 0, A \leq 40) = 0.010625322687488965$$

$$P(S = 1, C = 1, G = 0, A \leq 40) = 0.08460553314142814$$

(d) Based on your result in (c), compute

$$P(S = 0|C = 1, G = 0, A \leq 40) \; and \; P(S = 1|C = 1, G = 0, A \leq 40)$$

Predict whether a female whose age is under 40 and who is in first class will survive or not. Using Bayes's Rule, we can obtain an expression for the above:

$$P(S = 0|C = 1, G = 0, A \leq 40) = \frac{P(S = 0)P(C = 1, G = 0, A \leq 40|S = 0)}{P(C = 1, G = 0, A \leq 40)} \tag{4}$$

$$P(S = 1|C = 1, G = 0, A \leq 40) = \frac{P(S = 1)P(C = 1, G = 0, A \leq 40|S = 1)}{P(C = 1, G = 0, A \leq 40)} \tag{5}$$

And then recognizing that the right side of each of these expressions is just the corresponding result from (c), we can make the substitution and obtain:

$$P(S = 0|C = 1, G = 0, A \leq 40) = \frac{P(S = 0, C = 1, G = 0, A \leq 40)}{P(C = 1, G = 0, A \leq 40)} \tag{6}$$

$$P(S = 1|C = 1, G = 0, A \leq 40) = \frac{P(S = 1, C = 1, G = 0, A \leq 40)}{P(C = 1, G = 0, A \leq 40)} \tag{7}$$

Now, all we need is a way to compute $P(C = 1, G = 0, A \leq 40)$. For this, we can use the Law of Total Probability, recognizing that $S = 0$ and $S = 1$ form a partition over S, and simply sum the two results from (c):

$$P(C = 1, G = 0, A \leq 40) = P(S = 0, C = 1, G = 0, A \leq 40) + P(S = 1, C = 1, G = 0, A \leq 40) \tag{8}$$

Substituting (8) into (6) and (7), we now have an expression for the desired result in terms of known quantities. Below is the code I used to compute these quantities:

```
df_first_class = df[df["Pclass"]==1]
df_first_class_female = df_first_class[df_first_class["Sex"]==0]
df_first_class_female_u40 =
    df_first_class_female[df_first_class_female["Age"]<=40]

#Use total probability law to estimate P(C=1,G=0,A<=40)
pC1G0A40 = pS0C1G0A40 + pS1C1G0A40
```

```
pS1gC1G0A40 = pS1C1G0A40/pC1G0A40
print("Estimate for survival given person is a first class woman under 40 on
    titanic: {}".format(pS1gC1G0A40))

pS0gC1G0A40 = pS0C1G0A40/pC1G0A40
print("Estimate for death given person is a first class woman under 40 on
    titanic: {}".format(pS0gC1G0A40))
```

The resulting values for $P(S=0|C=1,G=0,A\leq 40)$ and $P(S=1|C=1,G=0,A\leq 40)$, respectively, were 0.112 and 0.888. As a result, it was more likely for a female whose age is under 40 and who is in first class to survive than to die aboard the Titanic.

4. *Central Limit Theorem.* Let $X_1, X_2, X_3, \ldots$ be a sequence of i.i.d. random variables with finite mean μ and finite variance σ^2, and let Z_n be the sum of the first n random variables in the sequence:

$$Z_n = X_1 + X_2 + \ldots + X_n.$$

(a) Let X_n for $i = 1, 2, \ldots$ be a uniform continuous random variable taking values in the interval (1,4). Write a MATLAB program to plot the pdf of Z_n. Consider $n = 1, 3, 10, 30$ and compare your results across different n's.

The below python notebook cell will plot separate PDFs for each value of n:

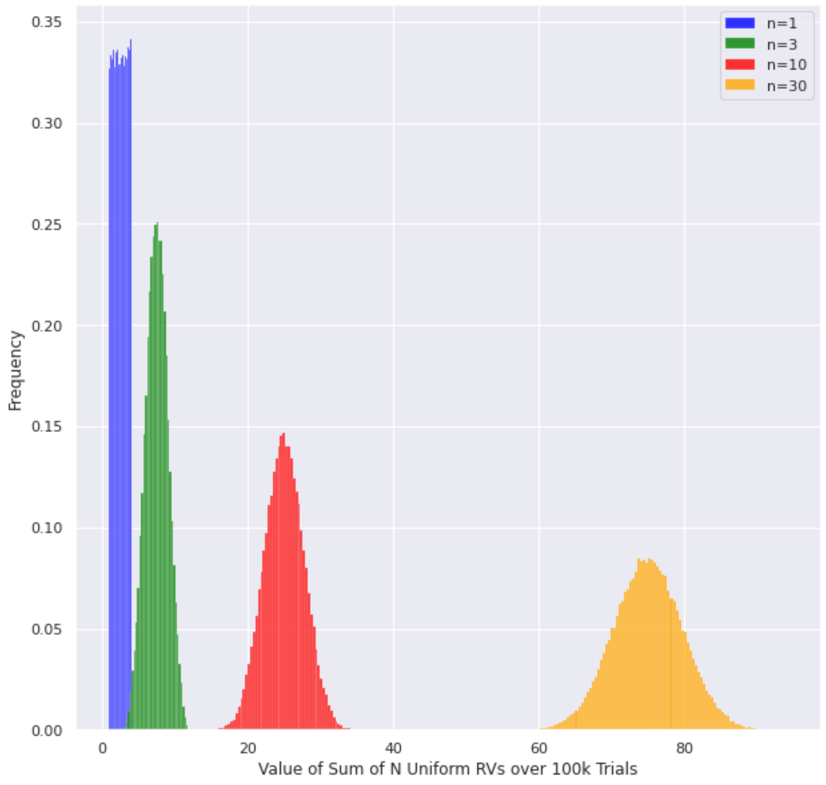
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

import seaborn as sns
sns.set(color_codes=True)
# settings for seaborn plot sizes
sns.set(rc={'figure.figsize':(10,10)})

n_vec = [1, 3, 10, 30]
n_trials = 100000

colors = ['blue', 'green', 'red', 'orange']

def get_xi():
   return np.random.uniform(1, 4)

for n in n_vec:
   sums = []

   for i in range(0, n_trials):

      values = []

      for j in range(0, n):
         value = get_xi()
         values.append(value)

      sums.append(np.sum(values)/n)

   plt.figure()

   ax = sns.distplot(sums,
              bins=100,
              label="n={}".format(n),
              kde=False,
              color=colors[n_vec.index(n)],
              hist_kws={"linewidth": 1,'alpha':0.8})
              ax.set(xlabel='Value of Sum of {} Uniform RVs in {}
                  Trials'.format(n, n_trials), ylabel='Frequency')
```

I'm not going to include the individual plots in the interest of brevity. Below is the code to generate a comparison plot of the PDFs of Z_n for different values of n:

```
sums_list = []

for n in n_vec:
```

```
sums = []

for i in range(0, n_trials):

   values = []

   for j in range(0, n):
      value = get_xi()
      values.append(value)

   sums.append(np.sum(values)/n)


ax = sns.distplot(sums,
             bins=100,
             label="n={}".format(n),
                kde=False,
             color=colors[n_vec.index(n)],
             hist_kws={"linewidth": 1,'alpha':0.8})
             ax.set(xlabel='Value of Sum of {} Uniform RVs in {}
                Trials'.format(n, n_trials), ylabel='Frequency')
```

Below is the plot:



(b) Calculate analytically the mean and the variance of X_i and Z_n in part (a).

$$E[Z_n] = E[X_1 + X_2 + \ldots + X_n] = E[X_1] + E[X_2] + \ldots + E[X_n] = n\mu$$

$$
\begin{aligned}
VAR[Z_n] = E[(Z_n - E[Z_n])^2] \\
&= E[((X_1 + X_2 + \ldots + X_n) - n\mu)^2] \\
&= E[(X_1 - \mu)^2 + (X_2 - \mu)^2 + \ldots + (X_n - \mu)^2] \qquad (9) \\
&= VAR[X_1] + VAR[X_2] + \ldots + VAR[X_n] \\
&= n\sigma^2
\end{aligned}
$$

Where, in (9), the result is supported by Bienayme's formula.

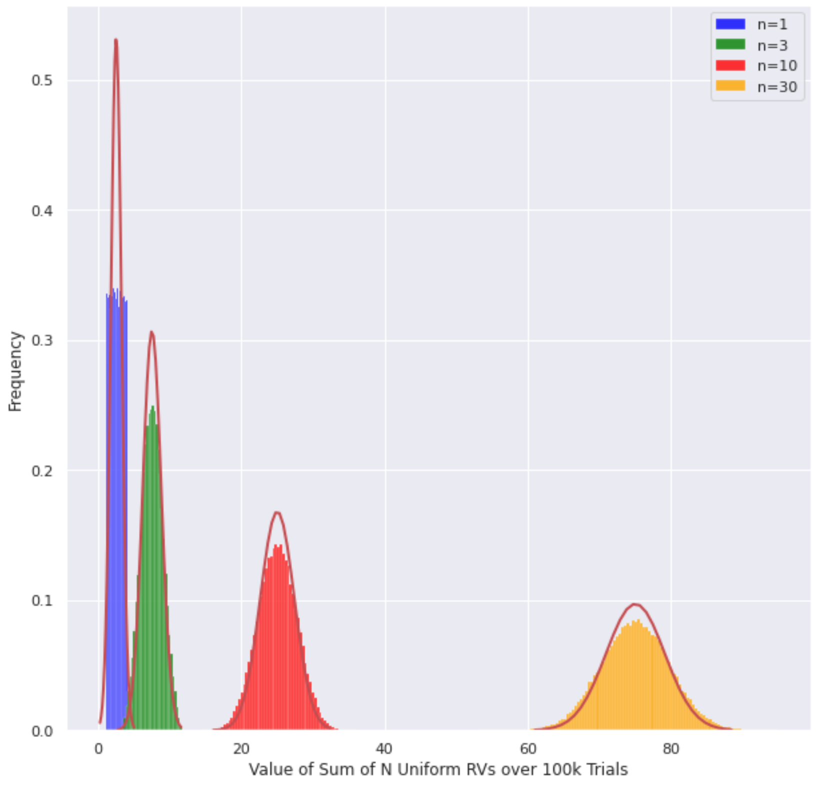
(c) Write a MATLAB program to generate a Gaussian random variable with the same mean and variance as Z_n. Superimpose its pdf on the plots from part (a).

The mean of a continuous uniform RV in the range [1, 4] is 2.5 and the variance is $\frac{3}{4}$. Following the results in (b), we can implement some code to plot a Gaussian with mean $2.5n$ and standard deviation $0.75n$, as below:

```
## Part c: superimposing a gaussian:

n_vec = [1, 3, 10, 30]
n_trials = 100000

colors = ['blue', 'green', 'red', 'orange']

def get_xi():
   return np.random.uniform(1, 4)

sums_list = []


plt.figure()

for n in n_vec:
   sums = []

   for i in range(0, n_trials):

      values = []

      for j in range(0, n):
         value = get_xi()
         values.append(value)


      sums.append(np.sum(values))
      #Y = (np.sum(values)/np.sqrt(n)) + 2.5*(1-np.sqrt(n))
      #sums.append(Y)

   sums_list.append(sums)

   ax = sns.distplot(sums,
            bins=int(20*np.sqrt(n)),
            hist=True,
            label="n={}".format(n),
            kde=False,
            color=colors[n_vec.index(n)],
            hist_kws={"linewidth": 0.25,'alpha': 0.8, "density": True})
```

```
ax.set(xlabel='Value of Sum of N Uniform RVs over {}k
    Trials'.format(int(n_trials/1000)), ylabel='Frequency')
ax.legend()

#generate the gaussian
mu, sigma = 2.5*n, 0.75*np.sqrt(n) # mean and standard deviation
s = np.random.normal(mu, sigma, 1000)

hist, bins = np.histogram(s, 30, density=True)
plt.plot(bins, 1/(sigma * np.sqrt(2 * np.pi)) * np.exp( - (bins - mu)**2 /
    (2 * sigma**2) ), linewidth=2, color='r')
```

And below is the plot:

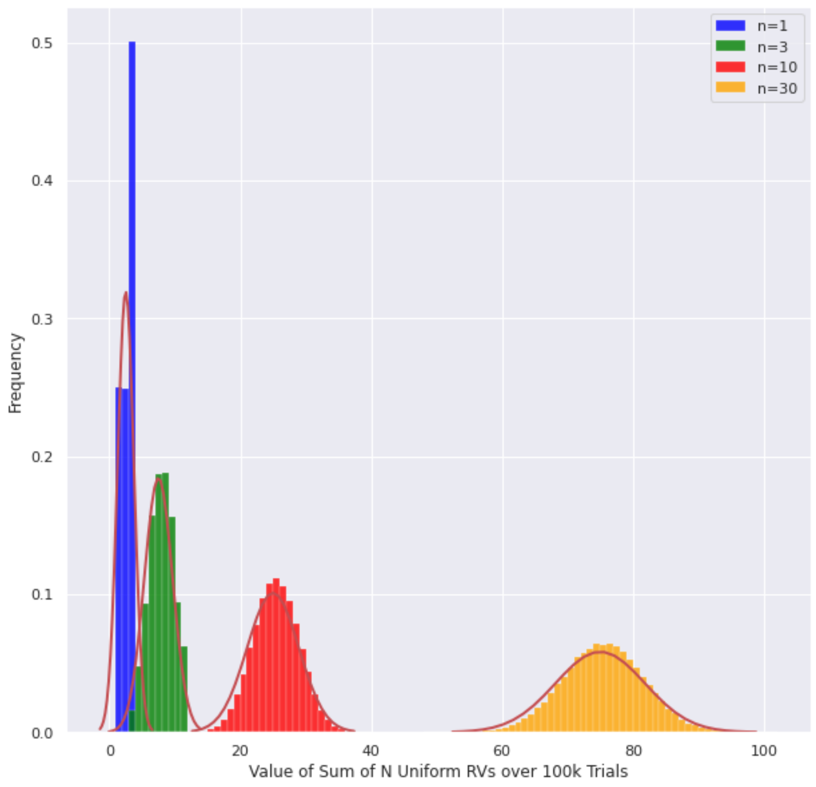


(d) Repeat parts (a), (b), and (c) with X_i representing a toss of a fair 4-sided die (see Problem 1). Note that X_i and Z_n are discrete in this case.
The Python notebook cell containing the code to do this is reproduced below:

```
n_vec = [1, 3, 10, 30]
n_trials = 100000

die = np.arange(1,5)
probs_unbiased = [(1/4)]*4

colors = ['blue', 'green', 'red', 'orange']

def get_xi():
   return np.random.choice(die, p=probs_unbiased) #throw the fair die t times


plt.figure()
```

```
for n in n_vec:
   sums = []

   for i in range(0, n_trials):

      values = []

      for j in range(0, n):
         value = get_xi()
         values.append(value)


      sums.append(np.sum(values))

   sums_list.append(sums)

   ax = sns.distplot(sums,
         bins=int(20*np.sqrt(n)),
         hist=True,
         label="n={}".format(n),
         kde=False,
         color=colors[n_vec.index(n)],
         hist_kws={"linewidth": 0.25,'alpha': 0.8, "density": True})
   ax.set(xlabel='Value of Sum of N Uniform RVs over {}k
       Trials'.format(int(n_trials/1000)), ylabel='Frequency')
   ax.legend()

   #generate the gaussian
   mu, sigma = 2.5*n, 0.75*np.sqrt(n) # mean and standard deviation
   s = np.random.normal(mu, sigma, 1000)

   hist, bins = np.histogram(s, 30, density=True)
   plt.plot(bins, 1/(sigma * np.sqrt(2 * np.pi)) * np.exp( - (bins - mu)**2 /
       (2 * sigma**2) ), linewidth=2, color='r')
```

And the generated plot appears below:



For the repetition of part (b), nothing really changes due to the fact that the X_is are discrete rather than continuous. That is a commonly stated property of CLT.