Simulations Homework

In [1]:
```
import numpy as np
import seaborn as sns
%matplotlib inline
```

```
C:\Users\furqa\anaconda3\lib\site-packages\scipy\__init__.py:146: UserWarnin
g: A NumPy version >=1.16.5 and <1.23.0 is required for this version of SciP
y (detected version 1.23.5
  warnings.warn(f"A NumPy version >={np_minversion} and <{np_maxversion}"
```

Reset Generator Function

If you didn't do so in class, write a function to reseed the numpy random number generator. It should default to setting the seed to 42, but be able to set it to whatever you want.

In [2]:
```
# reset the random seed
def reset_rng():
    rng = np.random.default_rng(42)
    return rng
```

Reset the generator using your function.

In [3]:
```
rng = reset_rng()
```

"Accept Cookies" Simulation

Assuming the base rate for hitting the "Accept Cookies" button when landing on a website is 85%, do a set of 10,000 simulations of 20 people visiting a given website and accepting cookies.

In [4]:
```
# set the variables
num_people = 20
num_simulations = 10000
acc_prob = 0.85
```

In [5]:
```
# run the function
n_accept = rng.binomial(num_people, acc_prob, num_simulations)
```

Based on your simulation, what is the probability of getting exactly 15 accepts?

In [6]:
```
n_accept_15 = n_accept[n_accept == 15]
```

In [7]:
```
len(n_accept_15) / 10000
```

Out[7]:
```
0.1
```

What is the probability of getting *at least* 15 accepts?

In [8]:
```
n_accept_15greater =  n_accept[n_accept >= 15]
prob_great_15 = len(n_accept_15greater) / 10000
prob_great_15
```

Out[8]:
```
0.9339
```

What is the probability of getting fewer than 15 accepts?

In [9]:
```
n_accept_15smaller =  n_accept[n_accept < 15]
prob_less_15 = len(n_accept_15smaller) / 10000
prob_less_15
```

Out[9]:
```
0.0661
```

Confirm that the last two probabilities computed sum to 1.0.

In [10]:
```
prob_great_15 + prob_less_15
```

Out[10]:
```
1.0
```

What Is and Isn't Binomial?

Check the binomial approximation for the election simulations from the in-class notebook for the cases in which we did and didn't account for the poll-to-poll variability arising from a single poll.

What is the expected standard deviation for our distribution of election outcomes based on the normal approximation?

The expected standard deviation would be around 1 because it is a normal distribution.

What was the empirical standard deviation of the distribution of election outcomes when we only used a single probability? ("*single poll, many elections*")

(You can just copy and paste the code from the in-class notebook to regenerate the simulated election outcomes.)

In [11]:
```
# reset the seed
rng = reset_rng()
# create variables

prob = 0.51
num_voters = 100000
num_simulations = 20000

# run the simulations

elections = rng.binomial(num_voters, prob, num_simulations) # run the simulat
```

In [12]:
```
elections.std()
```

Out[12]:
```
158.93412097331398
```

What was the empirical standard deviation of the distribution of election outcomes when we accounted for random variation in poll outcomes in our simulation? ("*simulate poll -> simulate election*")

In [13]:
```
# reset the seed
rng = reset_rng()

  # best guess of "true" probability
  # poll sample size
  # number of simulations to run
true_prob = 0.51
pol_size = 2000
num_simulations = 20000

    # get the polling results

polls = rng.binomial(pol_size, true_prob, num_simulations)


  # convert to probabilities
polls_prob = (polls / 2000)

 # medium city - expect around 100k voter turnout
expected_voters = 100000

rng = reset_rng()
simulated_elections = rng.binomial(expected_voters, polls_prob, num_simulatio
```

In [14]:
```
simulated_elections.std()
```

Out[14]:
```
1135.2591791690554
```

Effect of Poll Sample Size

As you have probably realized, these distributions of outcomes from many experiments we've been generating are, by definition, ***sampling distributions***! One firm law about sampling distributions is that their width depends strongly on sample size. As such, we would expect our simulated election outcomes to be affected by the size of the poll on which they are based.

In the cell below, run the *simulate poll -> simulate elections* code for poll sample sizes of 50, 100, 500, 1000, 2000 and 5000. For each sample size, record the obtained standard deviation of the distribution of outcomes. (pro tip: make a new code cell below and put them in a Python list)

In [15]:
```
 # reset generator
rng = reset_rng()
```

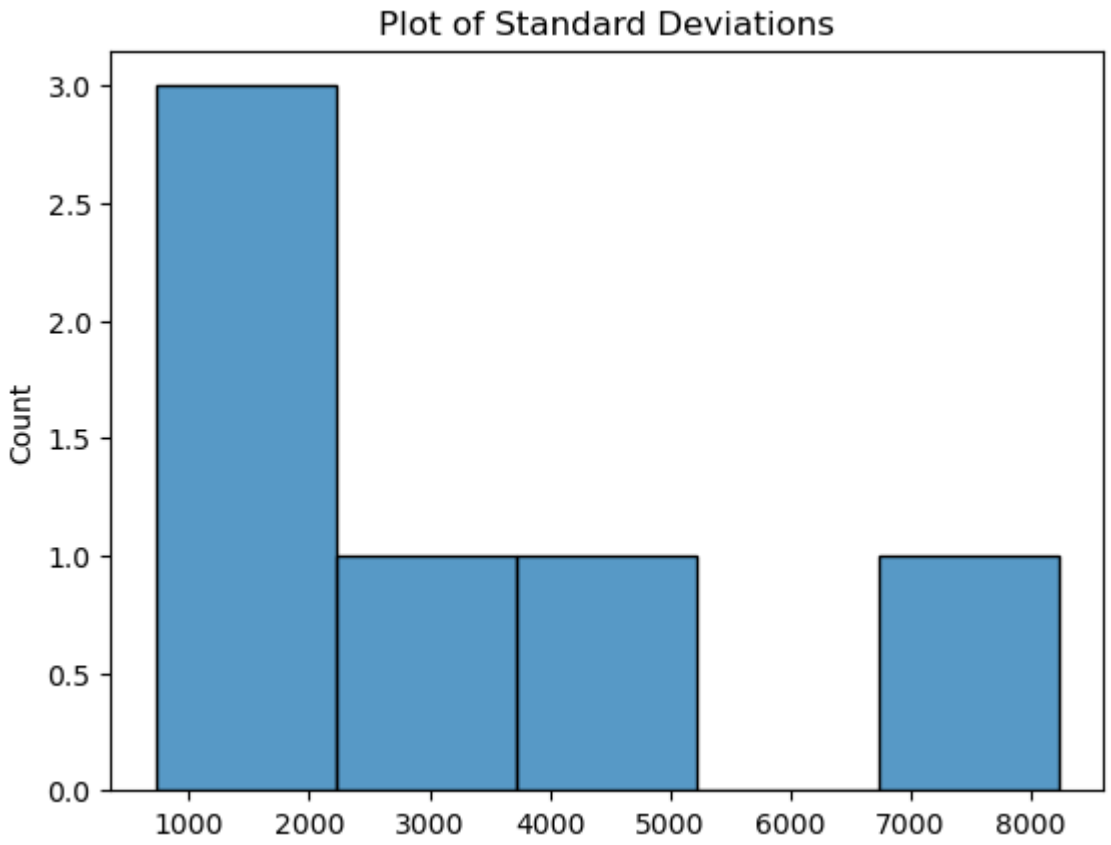
In [16]:
```
 # simulate
samp_sizes = [50, 100, 500, 1000, 2000, 5000]

  # best guess of "true" probability
  # poll sample size
  # number of simulations to run
std_list = []

for sample in samp_sizes:
    print(sample)
    # reset the seed
    rng = reset_rng()

      # best guess of "true" probability
      # poll sample size
      # number of simulations to run
    true_prob = 0.51
    pol_size = sample
    num_simulations = 20000

        # get the polling results

    polls = rng.binomial(pol_size, true_prob, num_simulations)

      # convert to probabilities
    polls_prob = (polls / pol_size)

     # medium city - expect around 100k voter turnout
    expected_voters = 100000

    rng = reset_rng()
    simulated_elections = rng.binomial(expected_voters, polls_prob, num_simul

    # store the list
    std_list.append(simulated_elections.std())

std_list
```

```
50
100
500
1000
2000
5000
```

Out[16]:
```
[7087.552661416986,
 5009.274160159059,
 2238.5560487091443,
 1591.4317128767848,
 1135.2591791690554,
 728.0110718700918]
```

In [17]:
```
 # plot
sns.histplot(std_list, binwidth =1500).set(title = "Plot of Standard Deviatio
 # compute std
```

Out[17]:
```
[Text(0.5, 1.0, 'Plot of Standard Deviations')]
```



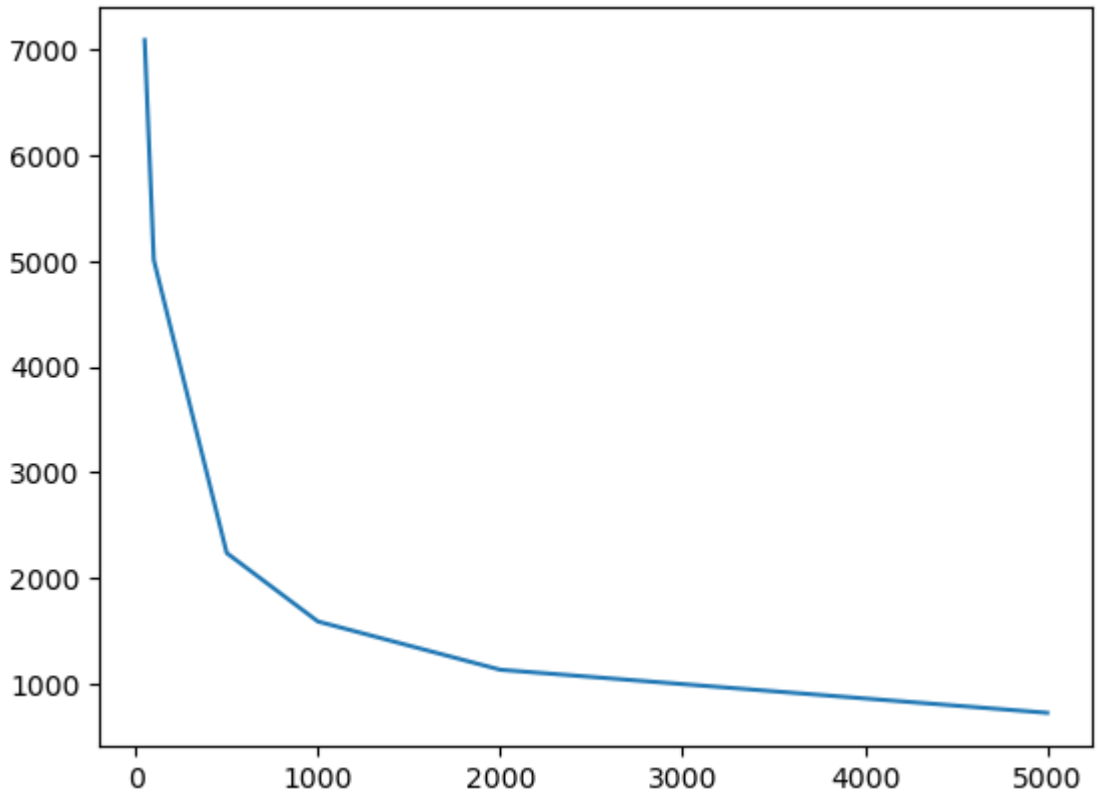
In [18]:
```
# compute the standard deviation
total_std = np.array(std_list).std()
total_std
```

Out[18]:
```
2307.2703524800104
```

Make a plot of standard deviation of outcomes (y-axis) vs. poll sample size (x-axis).

In [19]:
```
sns.lineplot(x = samp_sizes, y = std_list)
```

Out[19]:
```
<AxesSubplot:>
```



Based on this plot, why do you think almost all polls sample around 2000 people?

2000 people is where the drop in standard deviation happens. Basically, the standard deviation becomes more stable and the sample size increases, which reduces the difference between each data point.

Re-Write the Multi-Poll Code

The code for combining three polls using a weighted average works, but it is awkward. Changing it to handle a different number of polls would involve lots of copying and pasting and mistake-prone editing.

Make the code "Pythonic" so that all you have to do is provide a list (or tuple) of poll results and another for poll weights, and your code will do the rest.

Your code can be just code in a code cell. But if you're feeling spicy, you could make it a function!

In [27]:

```
def combine_weights(poll_list, poll_weights):
    rng = reset_rng()
    sum_of_w = np.sum(poll_weights)
    samp_sz = 2000                          # poll sample size
    n_sims = 20000
    n_voters = 100000


    poll_results = []
    poll_probs = []

    # conduct the polls
    for polls in poll_list:
        poll_results.append(rng.binomial(samp_sz, polls, n_sims))
    # convert poll total to probablities
    for poll in poll_results:
        poll_probs.append(poll/samp_sz)

    # set the number of voters


    elec_results = []

    for i in range(len(poll_list)):
        elec_results.append(rng.binomial(n_voters, poll_probs[i], n_sims))
    # compute weighted average
    print(elec_results)

    weighted_sum = 0

    for i in range(len(poll_list)):
        weighted_sum += (poll_weights[i] * elec_results[i])
    weigthed_average = weighted_sum / sum_of_w

    return weigthed_average
```

Weight polls by sample size

Use your new code to compute predicted election outcomes based on 5 polls weighted by the sample sizes of the polls (or their square root, if you prefer – wink wink, nudge nudge). The polls are as follows:

```
poll_ests = [.51, .55, .53, .49, 0.50]

poll_samp_szs = [2000, 1000, 1500, 1200, 1142]
```

In [28]:
```
swer = combine_weights([.51, .55, .53, .49, 0.50],[2000, 1000, 1500, 1200, 114
swer
```

```
[array([50538, 52696, 51666, ..., 48278, 51861, 50988], dtype=int64), array
([56390, 55248, 54766, ..., 54227, 55658, 55062], dtype=int64), array([5142
8, 55314, 52912, ..., 52700, 53187, 50939], dtype=int64), array([49710, 4903
8, 49229, ..., 48181, 49845, 49992], dtype=int64), array([48536, 49320, 5063
5, ..., 49160, 49609, 49988], dtype=int64)]
```

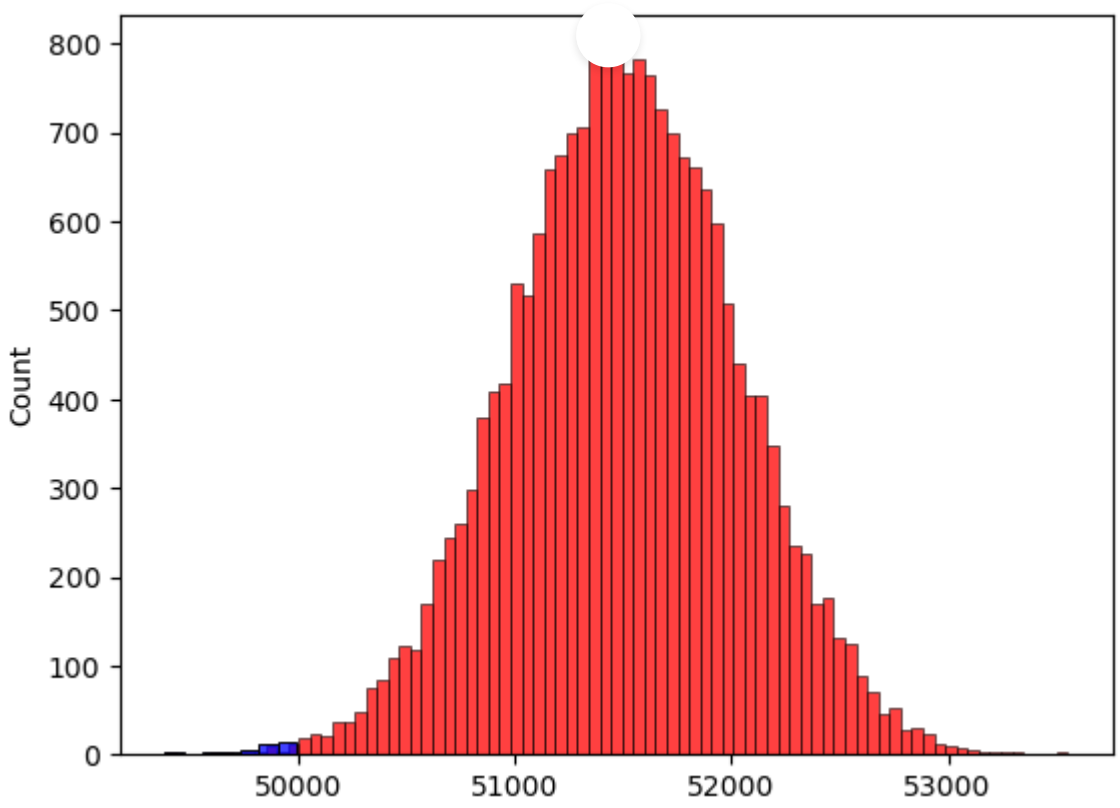
Out[28]:
```
array([51109.04881614, 52437.88950599, 51792.74627302, ...,
       50247.13826367, 51977.19643379, 51231.10143233])
```

Make a plot of the distribution of simulated outcomes, with the area representing the underdog winning highlighted.

In [29]:
```
the_answer[the_answer < 50000]
```

Out[29]:
```
array([49881.47033031, 49991.79421222, 49986.03215434, 49814.71821105,
       49889.43759135, 49985.58783981, 49942.20228004, 49931.3165741 ,
       49983.645133  , 49624.8368898 , 49387.18269512, 49927.77755042,
       49467.45308389, 49798.1368021 , 49690.17421806, 49955.61297866,
       49974.34989769, 49768.98655364, 49962.11897106, 49887.43788366,
       49679.57614733, 49725.0958784 , 49957.27886583, 49882.29611225,
       49952.83835136, 49803.75211926, 49406.10669395, 49820.88424437,
       49890.39432914, 49974.66237942, 49550.2516808 , 49598.7027185 ,
       49855.115171  , 49785.43992985, 49976.2253727 , 49793.9687226 ,
       49870.29815843, 49860.39842151, 49835.3861444 , 49885.34083601,
       49941.34960538, 49871.01578486])
```

In [33]:
```
sns.histplot(the_answer, color = 'red')
sns.histplot(the_answer[the_answer < 50000], color = 'blue');
```



Bonus (totally optional): Write your own function, `my_binom()` that does the same thing as `rng.binomial()`. The function should use `rng.random()` internally. To the user, it should behave just like `rng.binomial()`!

In [ ]:
```
```