tu16_HW

April 11, 2023

1 Simulations Homework

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

1.0.1 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.

```
[54]: def reset(seed = 42):
          rng = np.random.default_rng(seed)

          return rng
```

Reset the generator using your function.

```
[55]: rng = reset()
```

1.0.2 "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.

```
[36]: cookies = rng.binomial(n = 20, p = 0.85, size = 10000)
```

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

```
[37]: sum(cookies == 15)/10000
```

```
[37]: 0.1
```

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

```
[38]: sum(cookies >= 15)/10000
```

[38]: 0.9339

What is the probability of getting fewer than 15 accepts?

```
[39]: sum(cookies < 15)/10000
```

[39]: 0.0661

Confirm that the last two probabilities computed sum to 1.0.

```
[40]: 0.9321 + 0.0679
```

[40]: 1.0

1.0.3 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?

```
[41]: np.sqrt(100000*0.51*(1-0.51))
```

[41]: 158.082257068907

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.)

```
[42]: vote_a = rng.binomial(n = 100000, p = 0.51, size = 20000)
```

```
[43]: vote_a.std()
```

[43]: 158.09579697100583

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*")

```
[44]: rng = reset()
      polls = rng.binomial(100000, 0.51, 20000)
      prob = polls/100000
      voters = 100000
      med_polls = rng.binomial(voters, prob)
```

```
[45]: med_polls.std()
```

```
[45]: 223.8773119239866
```

1.0.4 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)

```
[46]: rng = reset() # reset generator
```

```
[47]: samples = [50,100,500,1000,2000,5000] # simulate
```

```
[48]: # compute std
      results = []

      for i in samples: # simulation data
          n_sample = i
          n_sim = 20000
          n_prob = 0.51

          poll = rng.binomial(n_sample, n_prob, n_sim) # run simulation

          rng = np.random.default_rng(42) # reset seed
          prob = poll / n_sample # probability

          voters = 100000
          poll_2 = rng.binomial(n_sample, prob, n_sim)

          results.append(poll_2.std()) # standard deviation

      results
```
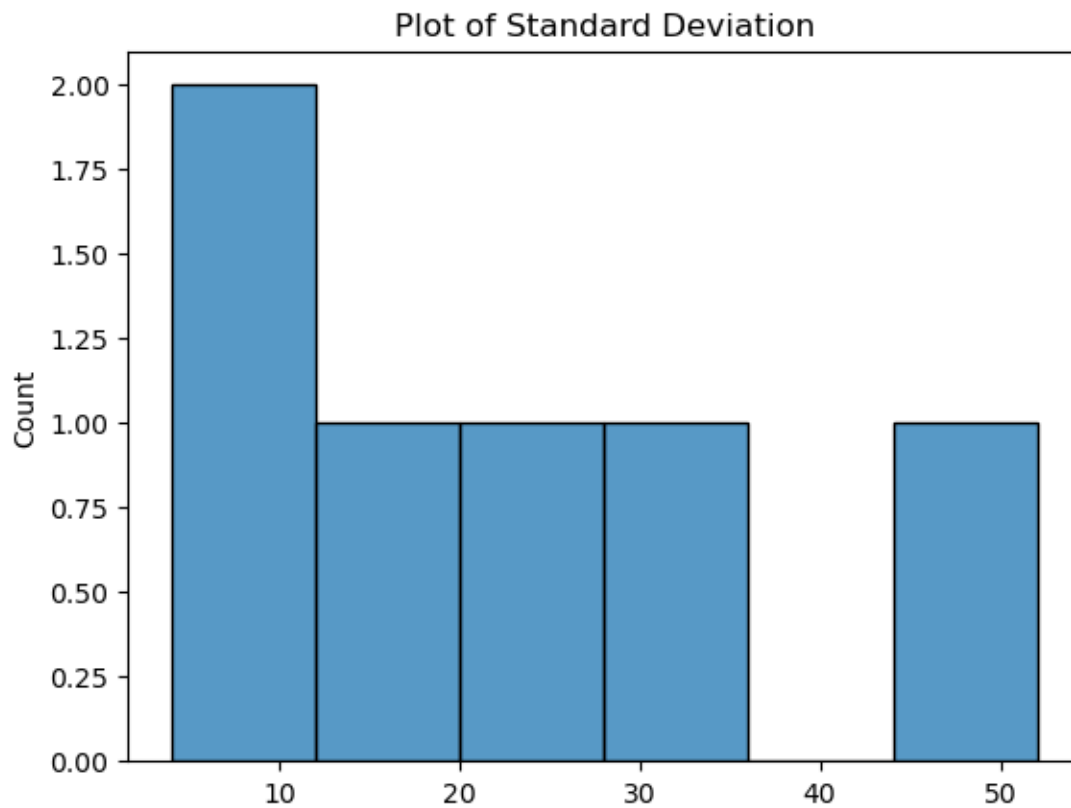
```
[48]: [4.037207538633604,
       7.050011715593102,
       15.807371441118857,
       22.33544430803874,
       31.69433799557107,
       49.79240385939908]
```

```
[49]: # plot
      sns.histplot(results, binwidth = 8).set(title = "Plot of Standard Deviation")
```
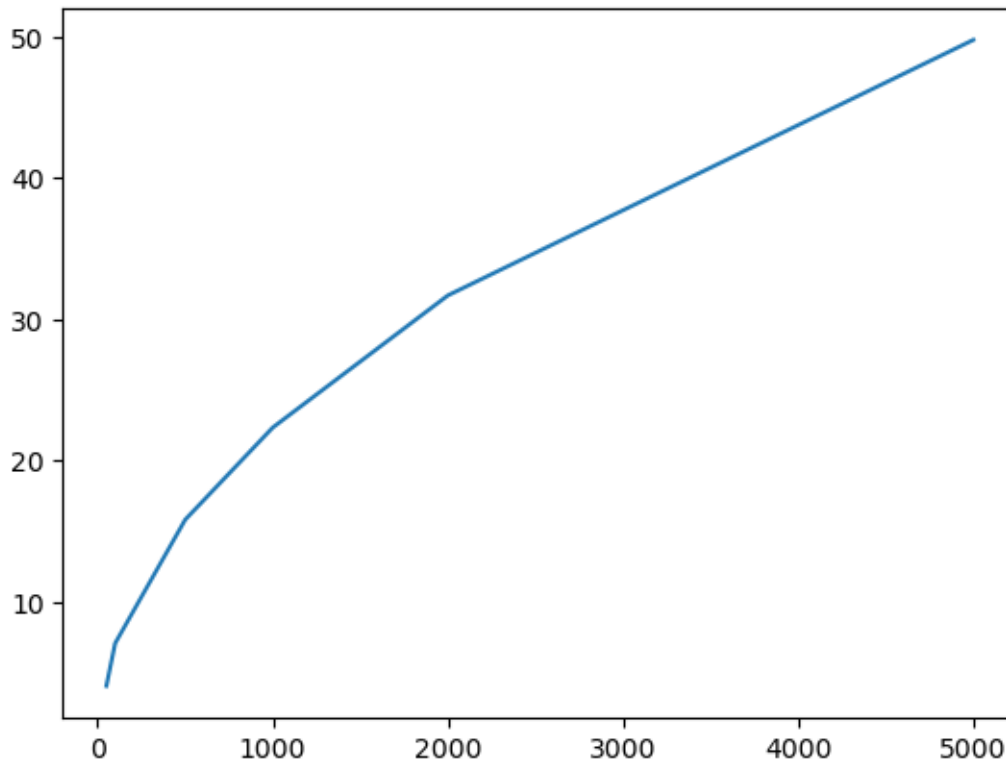
[49]: [Text(0.5, 1.0, 'Plot of Standard Deviation')]

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

```
[50]: sns.lineplot(x = samples, y = results)
```

[50]: <AxesSubplot:>

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

Because once we past a 2000 sample size the standard deviation starts increasing meaning there is a lot more variation making it difficult to make predictions from it.

1.0.5 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!

```python
[66]: def combine_weights(poll_list, poll_weights):
          rng = reset()
          sum_of_w = np.sum(poll_weights)
          samp_sz = 2000 # poll sample size
          n_sims = 20000
          n_voters = 100000
          poll_results = []
```

```python
    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])
        weighted_average = weighted_sum / sum_of_w

    return weighted_average
```

1.0.6 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]

```python
[57]: poll_ests = [.51, .55, .53, .49, 0.50]
      poll_samp_szs = [2000, 1000, 1500, 1200, 1142]
```

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

```python
[70]: sim_outcome = combine_weights(poll_ests, poll_samp_szs)
      sim_outcome
```

```
[array([50538, 52696, 51666, …, 48278, 51861, 50988]), array([56390, 55248,
54766, …, 54227, 55658, 55062]), array([51428, 55314, 52912, …, 52700,
53187, 50939]), array([49710, 49038, 49229, …, 48181, 49845, 49992]),
array([48536, 49320, 50635, …, 49160, 49609, 49988])]
```

[70]:
```
array([51109.04881614, 52437.88950599, 51792.74627302, …,
       50247.13826367, 51977.19643379, 51231.10143233])
```

[73]:
```python
sns.histplot(sim_outcome, color = 'red')
sns.histplot(sim_outcome[sim_outcome < 50000], color = 'blue');
```
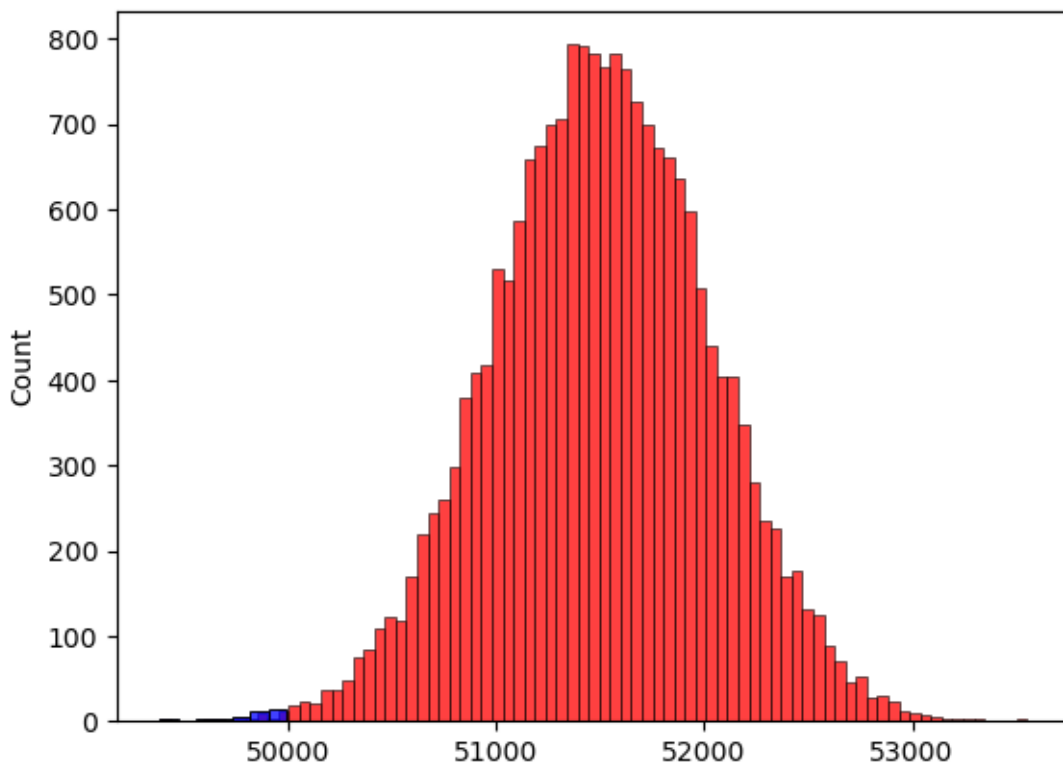
Bonus (totally optional): Write your own function, `my_binom()` that does the same thing as `rng.binomial()`. The function should use `rng.random()` internally. To the user, it should behave just like `rng.binomial()`!

[74]:
```python
# sorry this is already late I'm not going to attempt this
```