FDS I final project

Starter notebook to simulate election results for an 11 state country that uses an electoral college system.

Preliminary stuff

Import the things.

In [21]:
```
import numpy as np
import numpy.random as rnd
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

Load the data:

In [22]:
```
state_pop = np.loadtxt('statePopulations.csv', delimiter=',')
poll_quality = np.loadtxt('poll_quality.csv', delimiter=',')
state_polls = np.loadtxt('state_polls.csv', delimiter=',')
```

Explore the data

Data just got dropped in our lap. We need to explore the data a little bit to see what we're dealing with!

Before we ever start doing an "analysis" on data, we always – always – explore the data so we know what we're dealing with, and what further analysis can or should be done.

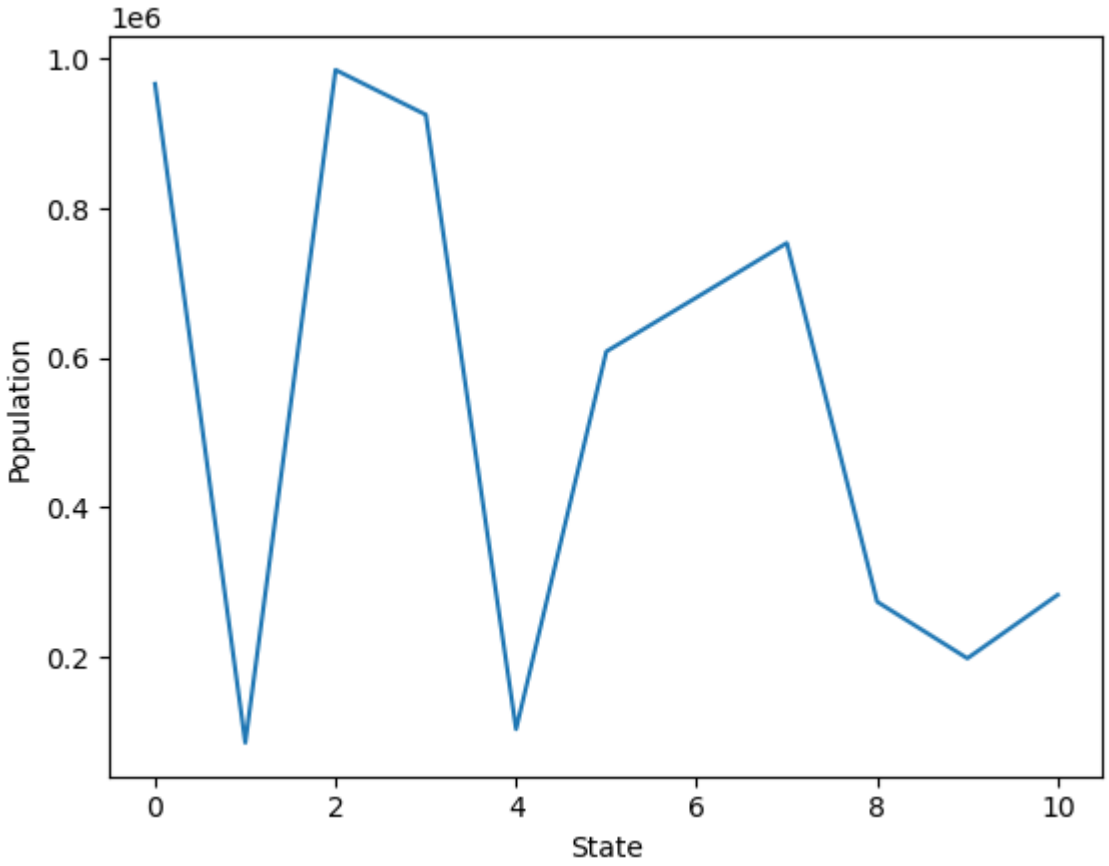
In [23]:
```
print(f"""pop vector shape is {state_pop.shape},
and poll quality shape is {poll_quality.shape},
and state polls shape is {state_polls.shape}""")
```

```
pop vector shape is (11,),
and poll quality shape is (7,),
and state polls shape is (11, 7)
```

This all makes sense as we were told that there were 11 states, 7 pollsters, and there should thus be 11x7 poll results (one poll for each state from each pollster).

Let's look at the populations.

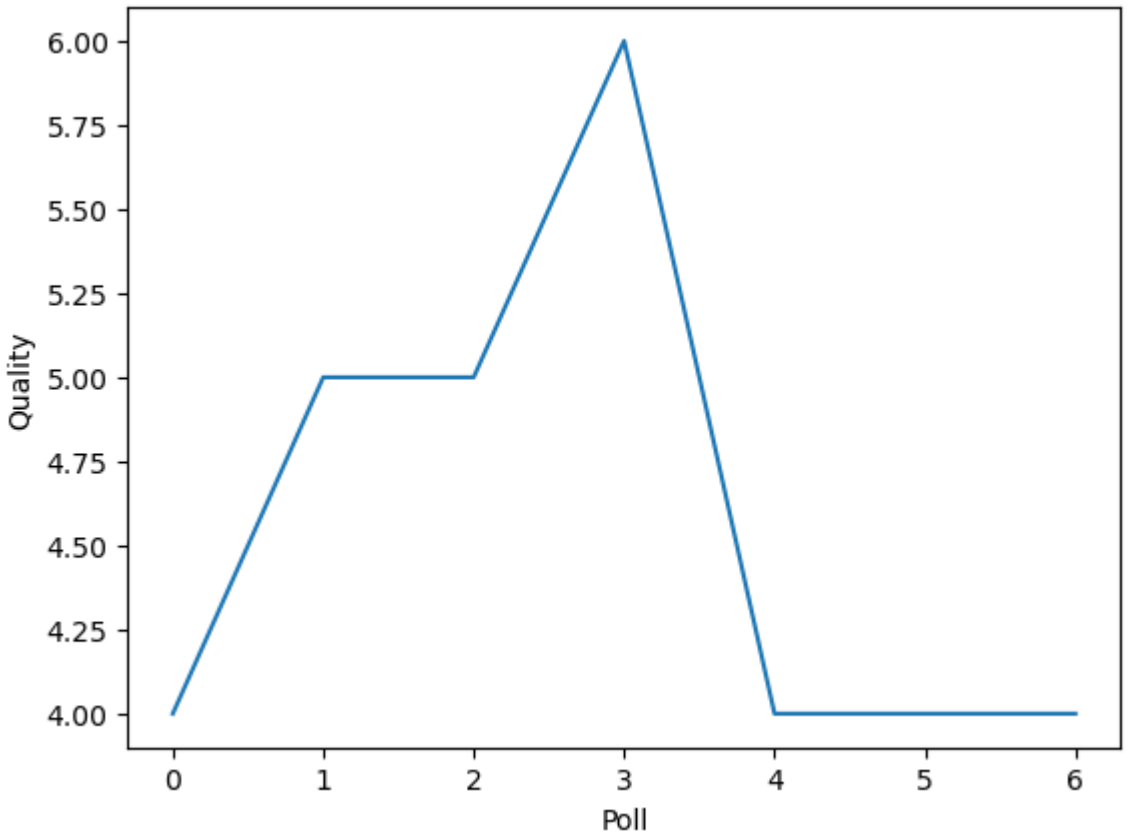
In [24]:
```
plt.plot(state_pop)
plt.xlabel('State')
plt.ylabel('Population')
plt.show()
```



So the state populations range from about 1 million to a little over 100,000 people, with the 1st, 3rd, and 4th states being the most populous.

Now let's look at the poll qualities.

In [25]:
```
plt.plot(poll_quality)
plt.xlabel('Poll')
plt.ylabel('Quality')
plt.show()
```

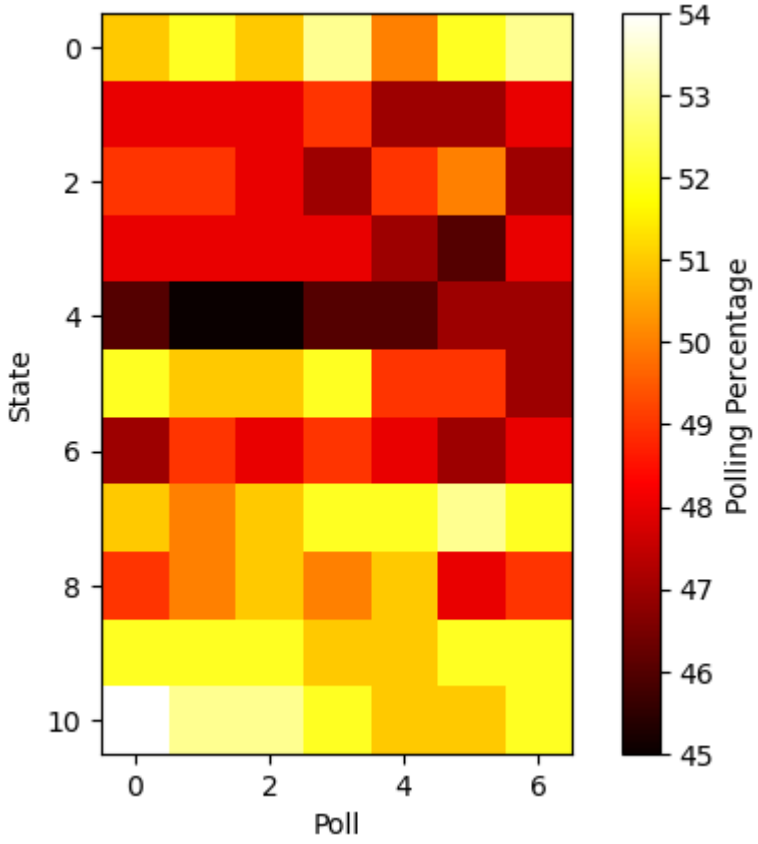


So poll 3 looks like the best (probably done by UT pollsters), and polls 0,4,5, and 6 being of lower quality.

Now let's look at the polling results. But before we do, let's think about what we should see. The polling data from different states could be very different (just as in the U.S. this fictional country could have "red" and "blue" states or whatever). But, if the polls are at all decent, they should be *correlated* across the states. That is, if one poll finds that a state strongly favors Barnes, then another poll shouldn't show that same state strongly favoring Noble. Make sense?

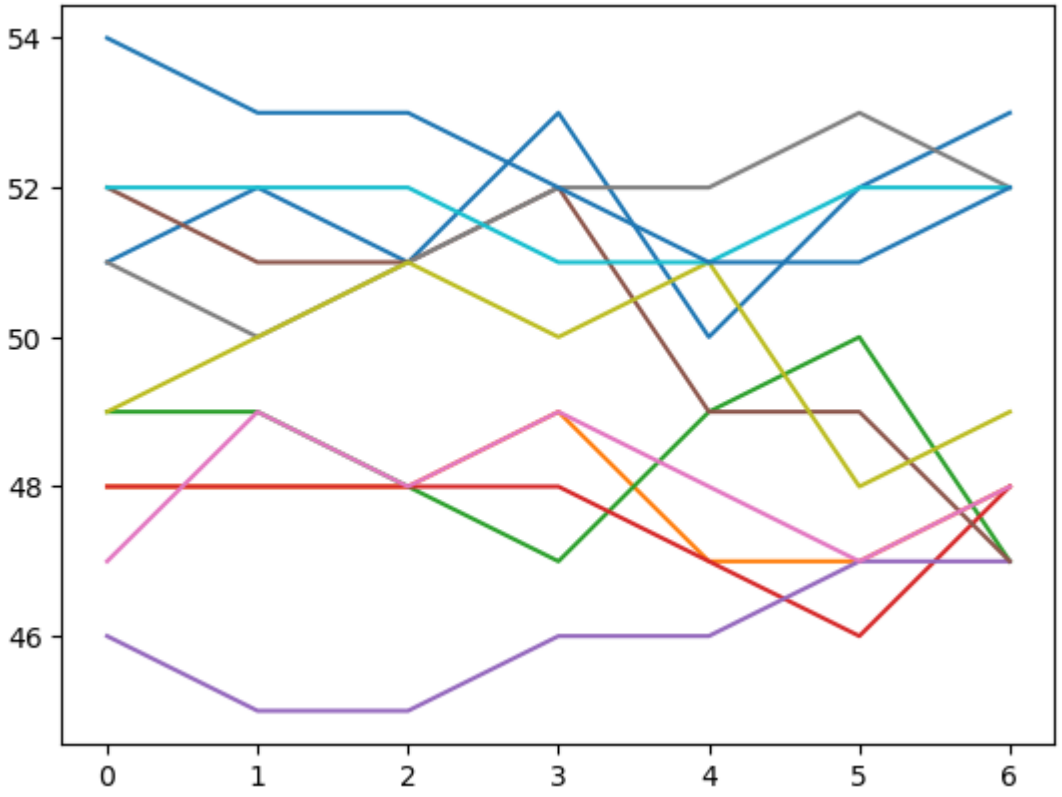
Because the polling data are in an 11 state by 7 poll matrix, we'll look at them as a heatmap – an image in which each pixel value represents the value of a single state/poll combination.

In [26]:
```
plt.imshow(state_polls, cmap='hot')
plt.colorbar(label="Polling Percentage")
plt.xlabel('Poll')
plt.ylabel('State')
plt.show()
```



In [27]:
```
plt.plot(state_polls.T)
```

Out[27]:
```
[<matplotlib.lines.Line2D at 0x15cfd4f90>,
 <matplotlib.lines.Line2D at 0x15d0f0550>,
 <matplotlib.lines.Line2D at 0x15d0f26d0>,
 <matplotlib.lines.Line2D at 0x15d0f2a50>,
 <matplotlib.lines.Line2D at 0x15d0f2cd0>,
 <matplotlib.lines.Line2D at 0x15d0f30d0>,
 <matplotlib.lines.Line2D at 0x15d0f3650>,
 <matplotlib.lines.Line2D at 0x15d0f3a50>,
 <matplotlib.lines.Line2D at 0x15d0f2e10>,
 <matplotlib.lines.Line2D at 0x15d100210>,
 <matplotlib.lines.Line2D at 0x15d100410>]
```



Let's look at this figure and see if it makes sense. Remember, the states can be quite different, but the polls should all reflect the same basic trend across states. And, indeed, we see that all the polls reflect state 4 going for Noble, where as state 10 seems to tilt heavily towards Barnes.

Now that we've wrapped out heads around the data and (hopefully) convinced ourselves that they pass the "sniff test", we can move on to the actual analysis: predicting the outcome of the election.

Prediction

Before we dive into coding, let's think about what we would do to make a single prediction from the data at hand. In doing that, we'll have figured out what we need to do on each simulated election in our analysis.

A reasonable approach would be to

- [x] compute a *weighted average* of the polls for each state, where the weights are the poll qualities
- [x] multiply the resulting average "% for Barnes" estimates by the state populations to get the # of predicted votes for Barnes from each state
- [x] compute the electoral college votes for Barnes for each state
    - determine the states in which Barnes won
    - get the electoral college votes for those states towards Barnes
- [x] sum up the electoral college votes for Barnes across the states
- [x] if the summed electoral college votes for Barnes is more than half of the total number, Barnes wins!

To paint a more complete picture of the election, though (and as per the assignment), we'll also want to store some stuff in addition to whether Barnes wins or not. Like

- the number of electoral college votes for Barnes
- the number of popular (overall) votes for Barnes
- whether or not Barnes would have won the election based on the popular vote

These three items actual don't add much to what we need to do. We've already decided we need to do the first. To compute the other two, we just need to:

- sum the votes from all the states to get the overall popular vote, and store it
- see if popular vote for Barnes was over half the population, and record the result as a win or loss.

Note: Whether Barnes won the (electoral college) election or the popular vote can either be computed for each simulation and stored, or computed after all the simulations are done from the vote totals in each simulation. Think about it.

Computing the weighted averages of the polls

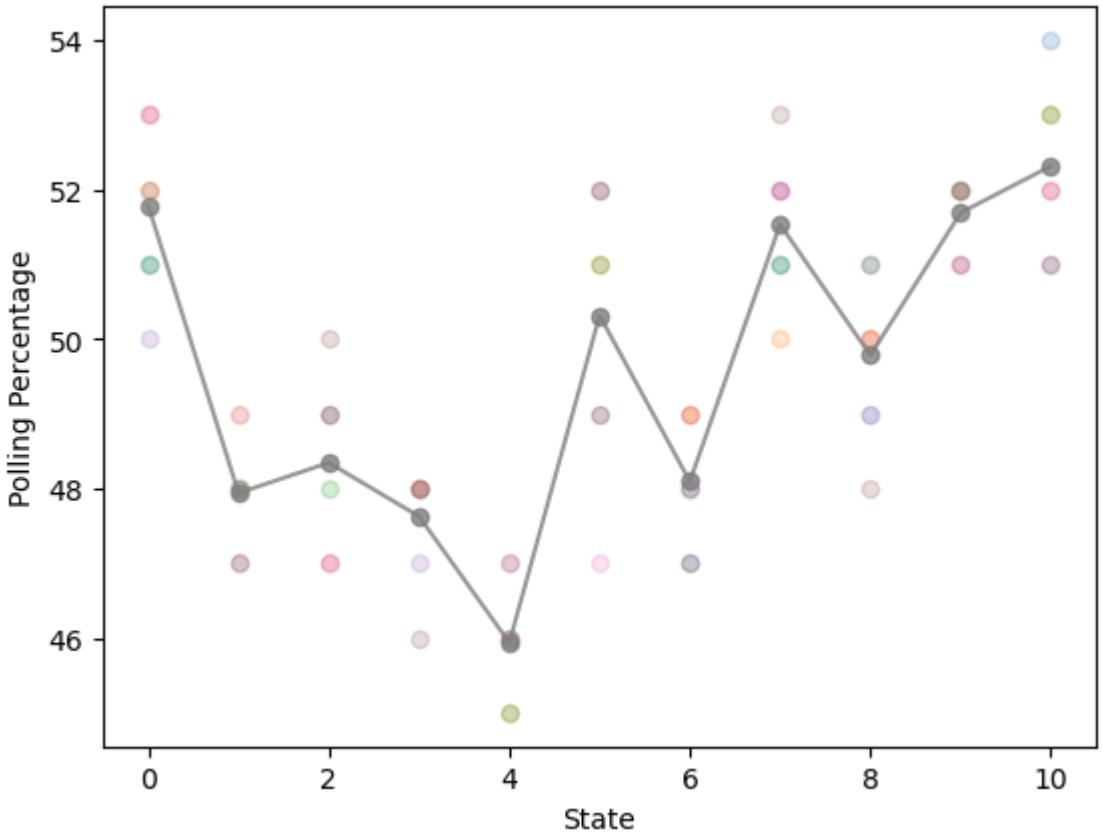
Let's start with step 1, computing a weighted average for each state. There are some clever ways to do this, but we'll use an easy-to-understand `for()` loop to compute the weighted average for each state.

In [28]:
```
# compute the average poll value for each state weighted by poll quality
n_states = len(state_pop) # we could set this to 11, but this is more general
weighted_polls = np.zeros(n_states) # initialize the vector for the weighted polls

for i in range(n_states):
    weighted_polls[i] = np.average(state_polls[i,:], weights=poll_quality)

weighted_polls
```

Out[28]:
```
array([51.78125, 47.9375 , 48.34375, 47.625  , 45.9375 , 50.3125 ,
       48.09375, 51.53125, 49.78125, 51.6875 , 52.3125 ])
```

To see if this makes sense, let's plot the original poll values by state, and then plot the weighted averages with a different color & symbol.

In [29]:
```
plt.plot(np.arange(11), state_polls, 'o', alpha=0.2);
plt.plot(np.arange(11), weighted_polls, 'o-', alpha=0.8);
plt.xlabel('State')
plt.ylabel('Polling Percentage')
plt.show()
```

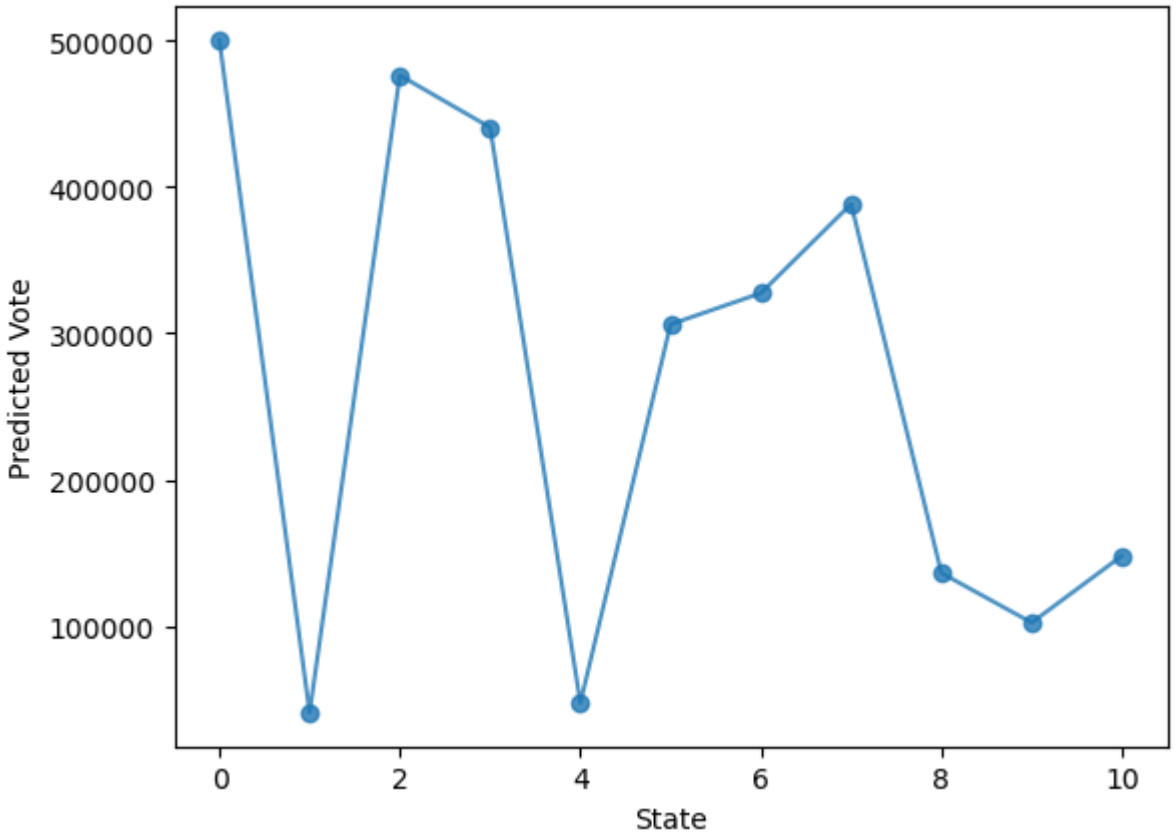


Okay, that looks reasonable. The weighted averages are always somewhere within the poll values, and state 4 has indeed gone for Noble.

Computing predicted votes per state

Now let's compute the predicted votes for each state based on our weighted poll average.

In [30]:
```
pred_vote_by_state = weighted_polls/100 * state_pop
plt.plot(np.arange(11), pred_vote_by_state, 'o-', alpha=0.8);
plt.xlabel('State')
plt.ylabel('Predicted Vote')
plt.show()
```



This is a great time for another reality check of our code! Since the polling is all near 50%, the graph of predicted votes we just made should strongly resemble the plot of state populations we made earlier, just scaled down by 0.5 on the y (population) axis.

Scroll back up to the population figure and see if this is true.

Compute the electoral college votes for Barnes

Now we need to compute which states Barnes won, and award those electoral college votes to Barnes.

First, let's make a vector indicating how many electoral college votes each state gets:

In [31]:
```
ppl_per_EC_vote = 1000 # given in the problem
EC_votes_by_state = state_pop/ppl_per_EC_vote
```

```
EC_vote_threshold = sum(EC_votes_by_state)/2
EC_votes_by_state
```

Out[31]:
```
array([966.727,  84.986, 985.4  , 925.493, 103.087, 608.575, 680.984,
       753.782, 273.659, 197.839, 282.816])
```

Then we'll compute the states won by Barnes

In [32]:
```
Barnes_won = pred_vote_by_state > state_pop/2
Barnes_won
```

Out[32]:
```
array([ True, False, False, False, False,  True, False,  True, False,
        True,  True])
```

Next, we'll just add up the electoral college votes from all the states that Barnes won.

In [33]:
```
EC_votes_for_Barnes = np.sum(EC_votes_by_state[Barnes_won])
print(f"Barnes won {EC_votes_for_Barnes} electoral college votes")
if EC_votes_for_Barnes > EC_vote_threshold:
    print("Barnes won!")
else:
    print("Barnes lost...")
```

```
Barnes won 2809.739 electoral college votes
Barnes lost...
```

Compute the popular vote results

This is easy, we just sum the votes across the states.

In [34]:
```
pred_vote = np.sum(pred_vote_by_state)
```

Did Barnes win the election? All we need to determine this is to see if he won more than half of the total votes.

In [35]:
```
Barnes_won = pred_vote > sum(state_pop)/2
Barnes_won
```

Out[35]:
```
False
```

And that's it!

Doing the simulations

To do the simulations, you need to bundle key parts of the above code (without the plotting and reality checking) into `for()` loop, and store the needed things.

The key is adding realistic variability to the ***polling results*** for each simulation (each pass through the `for()` loop).

Your new best friend is `rnd.binomial()` – remember that you know the poll sample size (2000 for every poll), and the polling result for each state-by-poll combination, so you can create a new state x poll matrix of polling data for each simulation that reflects the ***binomial*** error that is inherent in the polls.

You also need to add voter turnout variability. You know the population of each state, and you know that, on average, 60% of people turn out to vote, so...

Have at it!

In [38]:
```
# Arrays to store results
num_simulations= 10000
electoral_votes_results = np.zeros(num_simulations)
popular_votes_results = np.zeros(num_simulations)
```

In [78]:
```
n_trials = 2000
p_success = state_polls/100
n_states = len(state_pop)
weighted_polls = np.zeros(n_states)

ppl_per_EC_vote = 1000 # given in the problem
EC_votes_by_state = state_pop/ppl_per_EC_vote
EC_vote_threshold = sum(EC_votes_by_state)/2

rnd.seed(42)
for i in range(num_simulations):
    # Resetting the vote counts for each simulation
    electoral_votes_Barnes = 0
    pop_votes_Barnes = 0

    #create variability in polls using rnd.binomial:
    polls_variability = rnd.binomial(n=n_trials, p=p_success, size=(n_states, 7))
    polls_var_percent=polls_variability/2000 #creates percentage of pollers that voted for Barnes
```

```
    #find weighted avg of the polls (with variability)
    for k in range(n_states):
        weighted_polls[k] = np.average(polls_var_percent[k,:], weights=poll_quality)

    #find voter turnout (with variability)
    for l in range(n_states):
        voter_turnout_percent = np.random.normal(loc=.6, scale=0.05, size=11) # Simulate voter turnout varia
        voter_turnout_percent = np.clip(voter_turnout_percent, 0, 1) # Ensure voter turnout values are withi
        voter_turnout = state_pop * voter_turnout_percent  #find voter actual turnout

    #find popular vote via weighted avg of polls (with variability) + voter turnout (with variability)
        pred_vote_by_state = weighted_polls * voter_turnout
        pred_pop_vote = np.sum(pred_vote_by_state)

        #find electoral vote:
        Barnes_won = pred_vote_by_state > voter_turnout/2
        EC_votes_for_Barnes = np.sum(EC_votes_by_state[Barnes_won])

 # Storing the results of the current simulation
    electoral_votes_results[i] = EC_votes_for_Barnes # Store the electoral votes for candidate A for this ex
    popular_votes_results[i] = pred_pop_vote # Store the popular votes for candidate A for this experiment
```

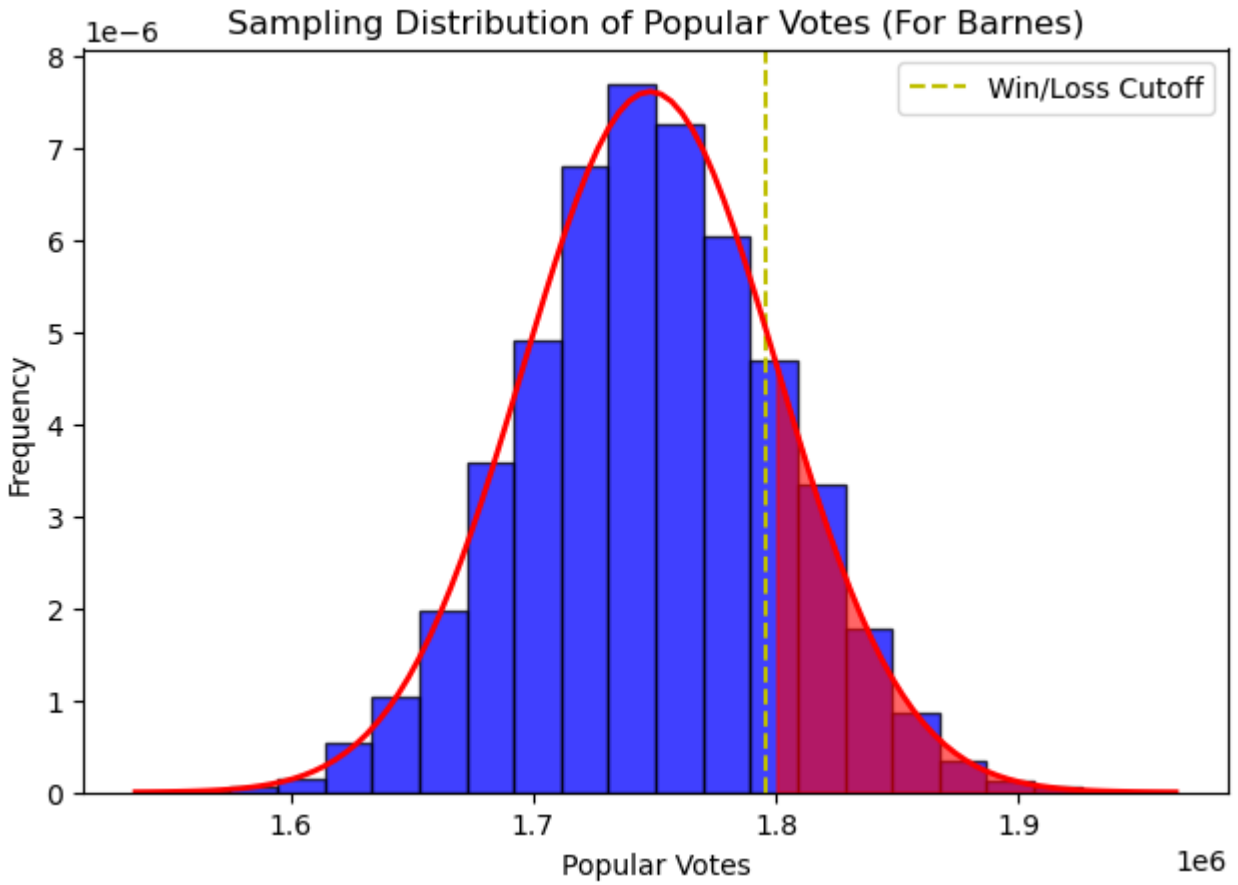
Post Simulation:

Plotting:

In [43]:
```
import scipy.stats as stats
```

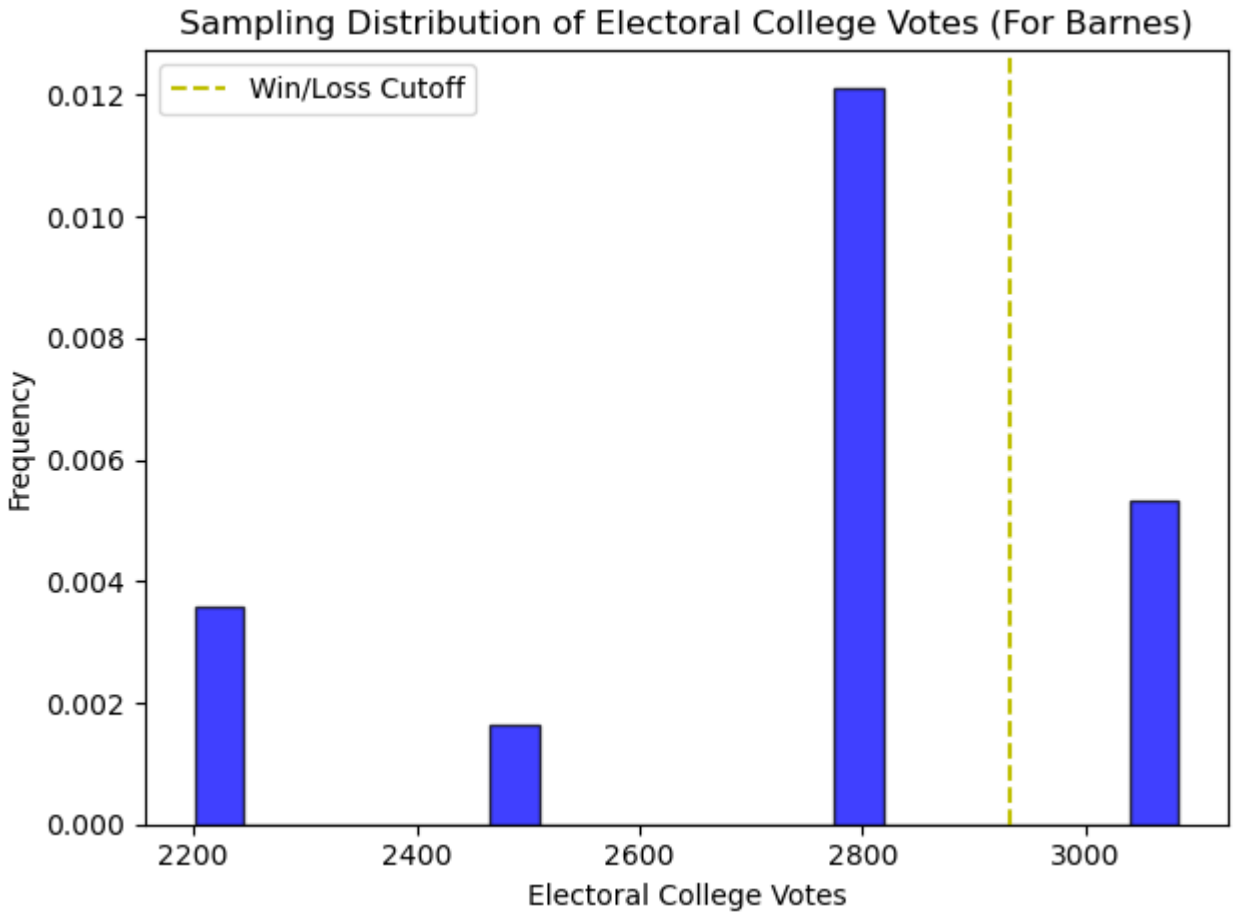
In [83]:
```
## Popular votes
pop_vote_threshold = sum(voter_turnout)/2

plt.hist(popular_votes_results, bins=20, density=True,
         color='b', alpha=0.75, edgecolor='k')
plt.title('Sampling Distribution of Popular Votes (For Barnes)')
plt.xlabel('Popular Votes')
plt.ylabel('Frequency')
plt.axvline(x=pop_vote_threshold, color='y', linestyle='--',label='Win/Loss Cutoff')


# Generate points for the best-fit normal distribution
xmin, xmax = plt.xlim()
```

```
mean_popular_votes = np.mean(popular_votes_results)
std_dev_popular_votes = np.std(popular_votes_results)

x = np.linspace(xmin, xmax, 100)
p = stats.norm.pdf(x, mean_popular_votes, std_dev_popular_votes)
plt.plot(x, p, 'red', linewidth=2)]



#shading
plt.fill_between(x, p, where=(x > pop_vote_threshold), color='red', alpha=0.6)


plt.tight_layout()
plt.legend()
plt.show()
```



In [85]:

```
## Electoral college:
plt.hist(electoral_votes_results, bins=20, density=True,
         color='b', alpha=0.75, edgecolor='k')
plt.axvline(x=EC_vote_threshold, color='y', linestyle='--', label='Win/Loss Cutoff')
plt.title('Sampling Distribution of Electoral College Votes (For Barnes)')
plt.xlabel('Electoral College Votes')
plt.ylabel('Frequency')
plt.legend()
plt.tight_layout()

plt.show()
```



Probability:

In [81]:
```
#Electoral
mean_ec_votes = np.mean(electoral_votes_results)
std_ec_votes = np.std(electoral_votes_results)

electoral_votes = EC_vote_threshold
p_less_than = stats.norm.cdf(electoral_votes, mean_ec_votes, std_ec_votes)
p_win=1- p_less_than
p_win
```

Out[81]:
```
0.26645370456750295
```

In [82]:
```
#Popular
mean_popular_votes = np.mean(popular_votes_results)
std_dev_popular_votes = np.std(popular_votes_results)

p_less_than = stats.norm.cdf(pop_vote_threshold, mean_popular_votes, std_dev_popular_votes)
p_win=1- p_less_than
p_win
```

Out[82]:
```
0.17988657674145636
```

Unfortunately, it looks like Barnes only has a 26.65% chance of winning the electoral college and a 17.99% of winning the popular vote. :(Thus, Noble is likely to win the election.