tu16_SimulationsI

March 28, 2023

1 Computer Simulations

Computer simulations, also known as Monte Carlo simulations, are incredibly powerful. They allow us to compute the likely outcomes of situations or events when it would be difficult, expensive, or impossible to recreate the situations or events in the real world many times over.

As a simple example, if NASA were working on a new two-stage rocket, they would need to know, for a given launch site, where the first stage is likely to land once it detaches and falls to earth. That information would be crucial, and no launch would be allowed without it. Of course, equations using Newtonian physics and aerodynamics can be used to predict an expected spot for impact. But, obviously, the discarded stage won't hit in that exact spot every time. So, what we really want is a *distribution*, like a 2D histogram, of likely landing spots. In other words, a map of the danger zone. To construct a map of the danger zone, we could:

- Build a bunch of rockets and test-fire them into space, recording the impact coordinates of the fallen stage each time.

or

- Tinker with the constants in the equations (such as wind speed and direction at various altitudes) by hand, recording the impact coordinates predicted by the equations each time.

The second option would be incredibly tedious and would be subject to bias (how would the exact wind directions be chosen?). The first option would expensive, time-consuming and, of course, incredibly dangerous.

As a second example, consider forcasting the landfall of hurricanes. Here, it's obviously impossible to "test fire" a bunch of hurricanes and, even if we could, doing so would defeat the very purpose of forcasting the landfall! Very elaborate sets of equations can predict *a* landfall for a particular set of constants (ocean temperatures, prevailing winds, etc.) but, again, what we want is a *distribution* of likely locations for landfall – a map of the danger zone.

In each of these situations, the solution is to run the equations on a computer – that is, to do a simulations – many many times over and, each time, having the computer select probable values for the constants (air temperature etc.) *at random*. Having the values for the constants be set to new random values each time will make the final outcome – impact site, landfall location, etc. – also vary, and the final distribution of these outcomes will give us the information we want.

Simulations are now ubiquitous in data science. They are, in fact, used in hurricane forcasting as well as in any other area in which people make predictions, such as sports, politics, and warfare (the first use of computer simulations was to help develop the first nuclear weapon). The key to doing computer simulations, however, is to be able to pump realistic *random* values into the equations on

each simulation, so that the variability in these values plays out as realistic variability in the final outcome. So where do we get these random numbers?

1.1 Pseudorandom Numbers

A computer is a deterministic machine meaning that, in any given state, it will always produce the same output for a given input. So how can a computer possibly produce random numbers? It can't. But it can produce ***pseudorandom numbers***, which are sequences of numbers that are random *for all practical intents and purposes.*

Pseudorandom numbers are so widely used and so good (that is, statistically indistinguishable from true random numbers) that most books and articles nowadays just refer to them as "random numbers". Similarly, the algorithms and programs used to generate them are called "random number generators" rather than "pseudorandom number generators".

The way computers make random numbers is conceptually quite simple. A starting value – a "seed" – is passed through an algorithm that produces a single output value, the first "random" number. This output is passed back though the algorithm to produce the second random number, and so on. The trick is to design an algorithm that makes it almost impossible to guess the input value from the output value. In other words, the numbers should appear to be unrelated or *independent*, and therefore random.

Here is a very simple algorithm for a (pseudo) random number generator. It takes three numbers:

- a number, s, the "seed"
- a number, a, called the "multiplier"
- and a number, m, called the "modulus"

It then computes the remainder – the modulus – of: $(seed * a)/m$

That equation yields our first random number. To get the second number, we feed the first output back into the equation in place of the seed. To get the third, we feed in the second and so on.

Here is code to implement the above algorithm – an actual (psuedo) random number generator!

[1]:
```
s, a, m = 1, 10, 23  # seed, multiplier, modulus
ps_rands = []          # empty list to hold our numbers

for i in range(20) :
    if i > 0 :
        x = (ps_rands[i-1]*a)%m
        ps_rands.append(x)
    else :
        x = (s*a)%m
        ps_rands.append(x)

ps_rands
```

[1]:
```
[10, 8, 11, 18, 19, 6, 14, 2, 20, 16, 22, 13, 15, 12, 5, 4, 17, 9, 21, 3]
```

Run the code and see if you can find a pattern in these numbers! If you can't, then they pass at least one informal test of randomness – the test of you!

Re-run the above code with various values for the seed. What happens?

The same sequences of numbers show up for a seed.

Playing with the above code hopefully made you realize that this simple random number generator is actually not a good one. But it's also quite simple, so we got what we paid for!

A moment's thought should convince us that, if a random number generator outputs a number, say a 5, that it has already generated, then the sequence between the first and second 5 will just keep repeating. It has to, because the algorithm will always generate the same output for a given input! And sequences of numbers that repeat are not random! So one of the keys to good random number generators is to have them yield number sequences that go on for a very very very long while before they start repeating.

Modern random number generators are quite good; we don't need to worry about them, we just need to use them. Importantly, however, they are still algorithms, which means that given the same starting point – the seed – they will always produce the same sequence of numbers. This is a feature not a bug! It means that as long as we know our seed, we can always reproduce our analyses!

> Note: Modern computer operating systems are gathering true random numbers all the time, like the timing between your keyboard strokes, mouse movements, communications between the CPU and hard disk, and even the small fluctuations in noise around the computer! It uses these random numbers to encrypt your data and communications.
>
> If you use numpy to generate random numbers without specifing a seed, numpy will dip into the random numbers used by your computer's OS. But then, of course, you won't know what your seed was and won't be able to reproduce your analysis!

1.2 Preliminaries

In order to actually do simulations, we are going to be using the random number generating capabilities of numpy, so let's import it as usual. We're also going to want to make some histograms, so let's import Seaborn as well.

[2]:
```
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
```

And run this command to make sure our plots show up.

[3]:
```
%matplotlib inline
```

1.3 Using numpy's random number generator

The prefered method for getting random numbers using numpy is a two-step process. The process is:

1. Create a random number generator object at the start of a script or simulation

2. Use that random number generator to draw samples from you desired distribution

Here's how we create a random number generator with a seed of 42:

[4]:
```
rng = np.random.default_rng(42) # seed with the answer to the ultimate question
```

We're allowed to name our generator whatever we want, but it's convention to name these generators "rng" unless there is a good reason to do otherwise.

We can now use this generator throughout a given project or project phase. If we wanted a number from a normal distribution, for example, we would call the `rng.normal()` method.

In the cell below, call `rng.normal()` 10 or so times, and see what you get.

[5]:
```
rng.normal()
```

[5]:
```
0.30471707975443135
```

Is that about what you would expect? Why?

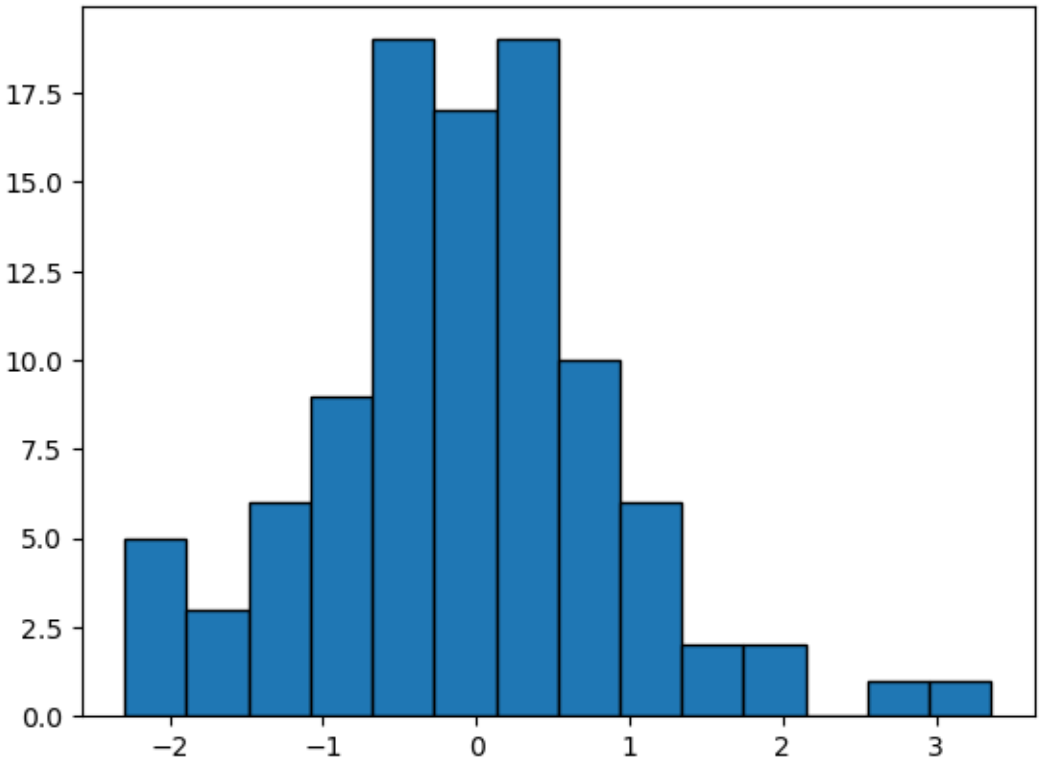
Yes that is what is expected. The numbers center around 0, rarely generating numbers above 2, which is expected with a normal distribution.

As always, we can use `rng.<tab>` to see what methods (statistical distributions in this case) are available. We can then use the help (e.g. `help(rng.normal)` to get help for that particular method.

In the cell below, get 100 samples from the Student's t distribution with 41 degrees of freedom, and plot (e.g. histogram) the resulting distribution.

[6]:
```
t_dist = rng.standard_t(size = 100, df = 41)
plt.hist(t_dist, edgecolor = 'black', bins = 14)
```

[6]:
```
(array([ 5.,  3.,  6.,  9., 19., 17., 19., 10.,  6.,  2.,  2.,  0.,  1.,
         1.]),
 array([-2.29110938, -1.88765571, -1.48420204, -1.08074837, -0.6772947 ,
        -0.27384103,  0.12961264,  0.53306631,  0.93651999,  1.33997366,
         1.74342733,  2.146881  ,  2.55033467,  2.95378834,  3.35724201]),
 <BarContainer object of 14 artists>)
```



Re-run the above a few times. Do the distributions look reasonable for a t distribution with $df > 30$?

Pretty much. The curve is basically a bell curve.

1.4 Simple simulations

The very idea of simulating something complicated can seem daunting if not impossible at first. How would we simulate something as complicated as a soccer game or traffic on a social media platform? Where would we even start? To get the idea of simulations, let's start by simulating a couple of very simple situations, coin flips and rolls of the dice.

1.4.1 Coin Flips

A coin flip is probably the simplest physical situation we could simulate, so that's a great place to start. Any event that has to have one of two outcomes is known as a "Bernoulli trial" (after a famous mathematicion Daniel Bernoulli). This includes coin flips, answers to yes/no questions, consumer "to buy or not to buy" decisions, voting behavior, etc.

Generally, we are interested in the sum of the results of many Bernoulli trials – for example, it is the total number of people that voted for canditate A (vs. candidate B) that determines the winner of an election. A single set of Bernolli trials is called an "experiment". The outcome of such

experiments, the sum of some number of Bernolli trials, has a *binomial* distribution. Thus, such simulations are done using the `rng.binomial()` method.

The binomial distribution has two parameters,

- the number of trials (e.g. the number of people voting in an election), and
- the probability of one of the outcomes occuring on a single trial.

(note that only one probability is needed, because the probability of the second is 1 minus the probability of the first)

In terms of a coin, this translates to the number of times we flip a coin, and the probability that our coin comes up "heads" on any one trial.

A Fair Coin First, let's reset our random number generator, `rng`, with a seed of 42 so everybody gets the same output.

Reset `rng` with a seed of 42 below:

[7]:
```
rng = np.random.default_rng(42)
```

(We'll be resetting the generator back to 42 a lot – so, if you're feeling plucky, you could write yourself a little function with a short name to do this!)

[8]:
```
def reset(seed = 42):
    rng = np.random.default_rng(seed)

    return rng
```

And now let's simulate a fair coin or, more precisely, an experiment on a fair coin.

In the cell below, use `rng.binomial()` to do a simulation giving the number of heads resulting from a fair coin being flipped 10 times:

[9]:
```
rng.binomial(n = 1, p = 0.5, size = 10)
```

[9]:
```
array([1, 0, 1, 1, 0, 1, 1, 1, 0, 0])
```

Because we reset our random number generator, you should have gotten a 6. Run the above code a few more times, noting the numbers.

Now reset the generator to a seed of 42 again:

[10]:
```
rng = reset()
```

And get a new sequence of number-of-heads for 10 flips of a fair coin to confirm you get the same sequence.

[11]:
```
rng.binomial(n = 1, p = 0.5, size = 10)
```

[11]:
```
array([1, 0, 1, 1, 0, 1, 1, 1, 0, 0])
```

Now reset the seed to something else, and run a few more binomial experiments.

[12]:
```
rng = reset(30)
```

[13]:
```
rng.binomial(n = 1, p = 0.5, size = 10)
```

[13]:
```
array([0, 0, 0, 1, 1, 1, 0, 0, 0, 1])
```

You should see a different sequence of numbers because you used a different seed.

Simulating Many Experiments Simulating one experiment or event isn't very helpful in and of itself. Just because a single simulation said we'll get 6 out 10 heads doesn't mean we'll get that if we really do the coin flips!

The advantage of computer simulations is that they let us easily and rapidly compute the outcome of thousands or even millions of experiments in order to compute the *distribution* of likely outcomes – the "danger zone" in the rocket and hurricane examples.

You might be thinking "time for a `for` loop!" right now. Great! The `for` loop is used quite a bit in doing simulations! In this case, however, the `rng.binomial()` method takes an optional 3rd argument, which is the number of experiments to run. (The fact that it has this third argument is a clue that was written with simulations in mind!)

In the cell below, take advatage of of this third argument to simulate 100000 experiments, each consisting of 10 flips of a fair coin. Assign the output to the name `n_heads` (i.e. `n_heads = rng.binomial...`).

[14]:
```
rng = reset() # reset the genny

rng.binomial(n = 10, p = 0.5, size = 100000) # define of trials, probability of␣
↪a head on each trial
# and number of experiments to simulate
```

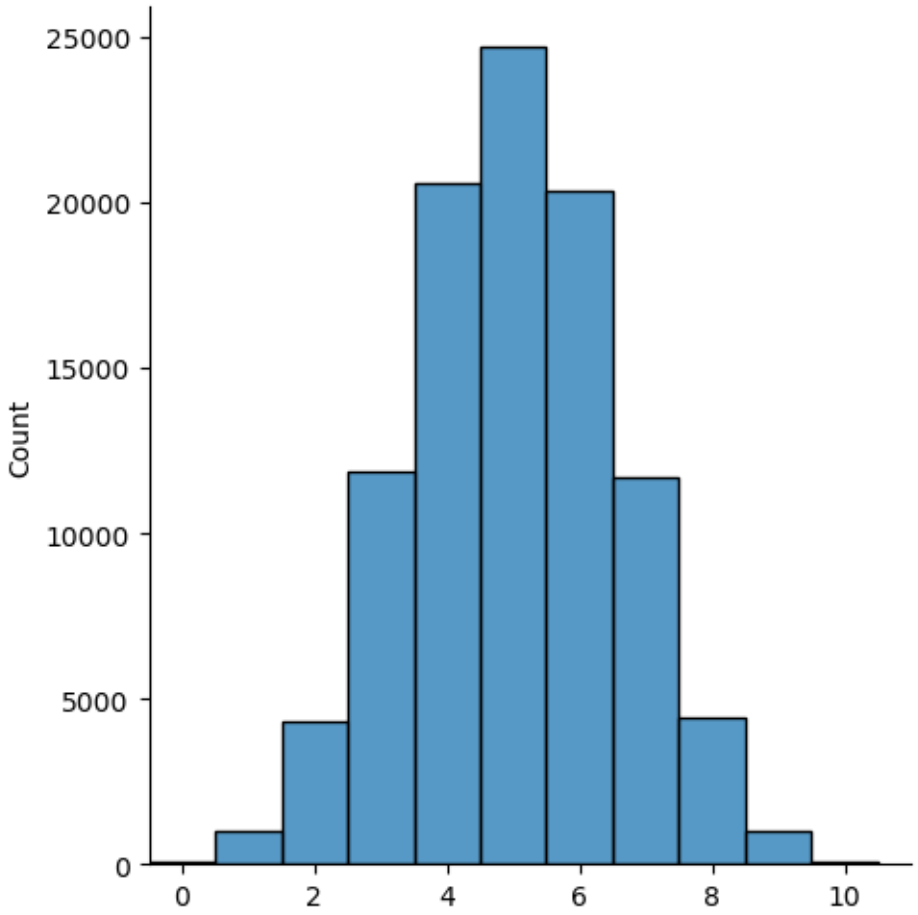
[14]:
```
array([6, 5, 7, …, 5, 3, 7])
```

[15]:
```
n_heads =  rng.binomial(n = 10, p = 0.5, size = 100000) # run the simulations␣
↪using rng.binomial
```

In the cell below, make a histogram of the outcomes, `n_heads`, from our 100000 experiments. For best results, set the binwidth to 1 and make the x-axis go from 0 to 10. The "goto" `sns.displot()` should work well for this!

[16]:
```
my_hist = sns.displot(n_heads, binwidth = 1, binrange = (-0.5,10.5))
my_hist.set(xlim = (-0.5,11))
```

[16]: <seaborn.axisgrid.FacetGrid at 0x7fe84b44ca90>



Based upon this histogram, if somebody flipped a coin 10 times and came up with 2 heads, would you suspect that person of cheating? Why or why not?

It's unlikely but I wouldn't suspect cheating.

What about 10 heads?

I would because it's extremely unlikely.

The Normal (Gaussian) Approximation A binomial distribution can be approximated as a Gaussian with

$\mu = n * p$

and

$\sigma = \sqrt{n * p * (1 - p)}$

Use the cell below to caluculate the mean and standard deviation of n_heads.

[17]:
```
n_heads.mean()
```

[17]:
```
4.99786
```

[18]:
```
n_heads.std()
```

[18]:
```
1.5813903440959793
```

Does the Gaussian approximation seem like a good one in this case?

Yep it's a pretty good approximation.

An Unfair Coin Perhaps surprisingly, it's almost impossible to make a biased coin because they are so thin relative to the size of their sides. However, we *can* easily simulate a biased coin so we would know what to look for if anyone did figure out how to make one.

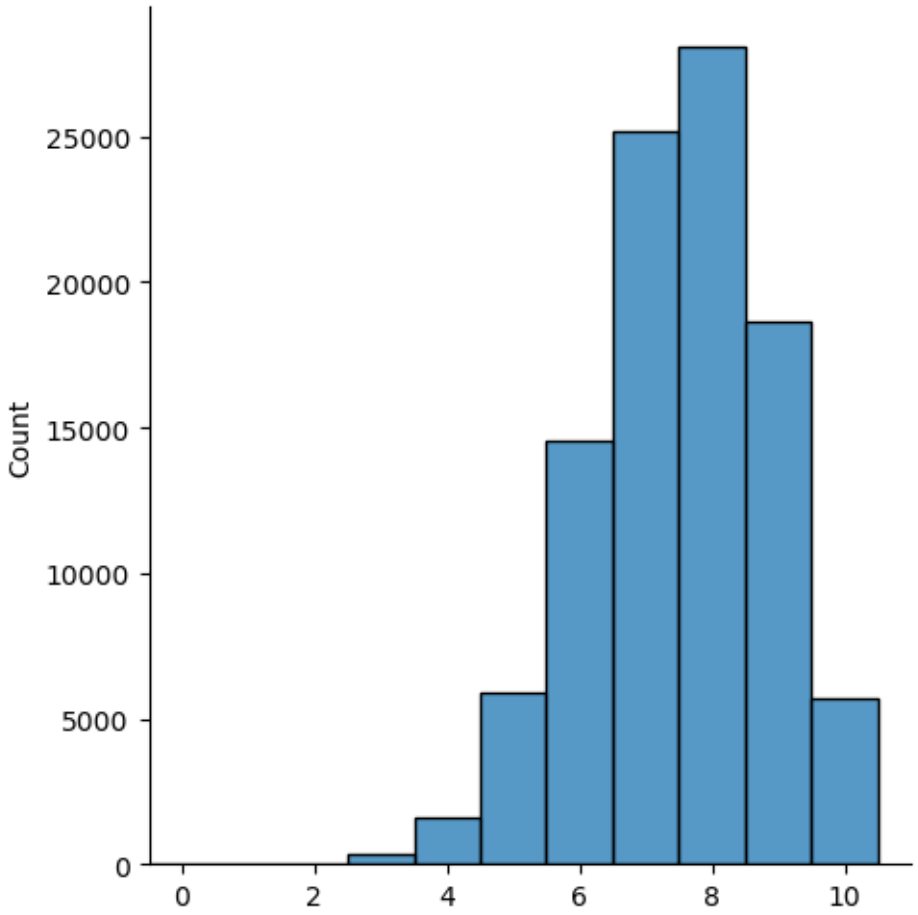
Use the cell below to simulate the same experiment as above, but using a coin that lands on heads 75% of the time on average.

[19]:
```
rng = reset()
n_heads =  rng.binomial(n = 10, p = 0.75, size = 100000)
```

Now make a histogram of the outcomes (for the rest of the notebook, we'll just take it for granted that we'll be making histograms of our outputs).

[20]:
```
unfair_hist = sns.displot(n_heads, binwidth = 1, binrange = (-0.5,10.5))
unfair_hist.set(xlim = (-0.5,11))
```

[20]:
```
<seaborn.axisgrid.FacetGrid at 0x7fe84b9489d0>
```



How did it change the distribution? Did the mean shift? Did the shape change?

The mean shifted to around 8 and the distribution itself is skewed.

1.5 Simulating Elections

Armed with nothing more than our ability to model coinflips, we can also simulate a surprising amount of other things, such as elections. Literally anything that can be reduced to a series to two-alternitave choices can be simulated using the binomial distribution.

Let's say there is an upcoming election in which one million people were expected to vote, and candidate "A" is leading in the polls at 52% vs. 48%. Here is a simple simulation of this election – the number of votes for candidate A – assuming the polling is accurate.

[21]:
```
rng.binomial(1000001, .52)
```

[21]: 519499

Re-run the above simulation until you think you have a good idea of how often the underdog – poor candidate B – is likely to win.

Now, in the cell below, change the simulation so that there are only 11 voters – like perhaps a soccer team is chosing between two potential captains, and only the starters get to vote. Run the simulation until you get again get a feel for how often the underdog prevails.

[22]:
```
rng.binomial(n = 11, p = 0.52)
```

[22]:
```
5
```

In the cell below, offer an explanation of why, with the same p = 0.52, the underdog prevails more in one case than another.

Because it's a small sample size.

1.5.1 Simulating election results based on a single poll

Above, we simulated one election at a time but, as with the coins, we really want to do many many experiments so that we can see the distribution of likely election outcomes.

single poll, many elections Let's say we have a poll of 2000 likely voters for a mayoral race from a smallish city that shows candidate A up by 51% to 49%. We expect a voter turnout of 100,000 people. We want to simulate this election, and we'll do the simulation 20,000 times to generate the distribution of likely outcomes.

In the cell below, reset the random number generator to a seed of 42.

[23]:
```
rng = reset()
```

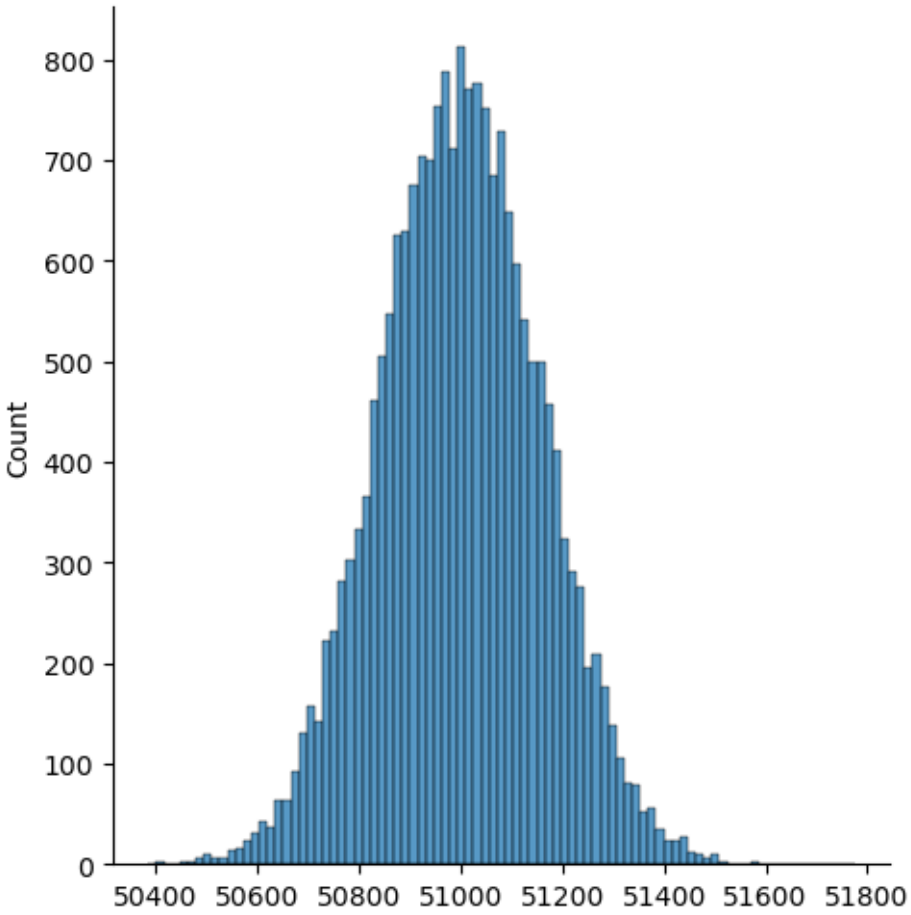
Now define names (variables) for the probability corresponding to 51%, the expected number of voters, 100,000, and the number of simulations to run, 20,000.

[24]:
```
vote_a = rng.binomial(n = 100000, p = 0.51, size = 20000)
```

And run and plot the our simulations.

[25]:
```
vote_hist = sns.displot(vote_a)
vote_hist
```

[25]:
```
<seaborn.axisgrid.FacetGrid at 0x7fe84b46a4c0>
```



Around how many times did candidate B pull off the upset in our simulated elections?

i dont know

Polls aren't perfect! If we conduct a poll, we get an *estimate* of how many people in a sample (usually around 2000 people) will vote for each candidate. Polls are always reported with a "margin of error" that will be around 2% for a poll with n = 2000. For the above poll, a news blurb might read something like

"Candidate A is leading Candidate B by 51% to 49%, but that's within the margin of error of 2%, so it's currently a statistical tie."

What is the margin of error? It is simply the 95% confidence interval on the percentage. What it's saying is that 1) the outcome of a poll is inherently variable and 2) the variability (which is approximately normally distributed) is such that +/- 2% encloses 95% of the distribution of possible poll outcomes. (If we remember our stats, we'll realize that the *standard error* of the poll is thus about 1%).

The bottom line is that, if our above simulation was correct, candidate B should have won at least

some of the time! The problem is that we were treating the *estimate* from the poll as a fixed constant, rather than as a variable that should fluxuate from experiment to experiment in our simulation.

How would we go about incorporating this variability into our simulations? Easy! Instead of considering the poll to be a fixed result, we'll include *a simulation of the poll* into each experiment, along with the simulation of the election.

simulate poll -> simulate election From a coding standpoint, one obvious way to add polling variability to each experiment would be something like this:

```
for i in range(n_sims) :
    # compute a poll result using rng.binomial() where p = original poll estimate
    # compute an election result where p = simulated poll result
    # store election result
```

However, if we get help on the binomial via `help(rng.binomial)`…

[26]:
```
help(rng.binomial)
```

```
Help on built-in function binomial:

binomial(…) method of numpy.random._generator.Generator instance
    binomial(n, p, size=None)

    Draw samples from a binomial distribution.

    Samples are drawn from a binomial distribution with specified
    parameters, n trials and p probability of success where
    n an integer >= 0 and p is in the interval [0,1]. (n may be
    input as a float, but it is truncated to an integer in use)

    Parameters
    ----------
    n : int or array_like of ints
        Parameter of the distribution, >= 0. Floats are also accepted,
        but they will be truncated to integers.
    p : float or array_like of floats
        Parameter of the distribution, >= 0 and <=1.
    size : int or tuple of ints, optional
        Output shape.  If the given shape is, e.g., ``(m, n, k)``, then
        ``m * n * k`` samples are drawn.  If size is ``None`` (default),
        a single value is returned if ``n`` and ``p`` are both scalars.
        Otherwise, ``np.broadcast(n, p).size`` samples are drawn.

    Returns
    -------
    out : ndarray or scalar
        Drawn samples from the parameterized binomial distribution, where
```

```
    each sample is equal to the number of successes over the n trials.

See Also
--------
scipy.stats.binom : probability density function, distribution or
    cumulative density function, etc.

Notes
-----
The probability density for the binomial distribution is

.. math:: P(N) = \binom{n}{N}p^N(1-p)^{n-N},

where :math:`n` is the number of trials, :math:`p` is the probability
of success, and :math:`N` is the number of successes.

When estimating the standard error of a proportion in a population by
using a random sample, the normal distribution works well unless the
product p*n <=5, where p = population proportion estimate, and n =
number of samples, in which case the binomial distribution is used
instead. For example, a sample of 15 people shows 4 who are left
handed, and 11 who are right handed. Then p = 4/15 = 27%. 0.27*15 = 4,
so the binomial distribution should be used in this case.

References
----------
.. [1] Dalgaard, Peter, "Introductory Statistics with R",
       Springer-Verlag, 2002.
.. [2] Glantz, Stanton A. "Primer of Biostatistics.", McGraw-Hill,
       Fifth Edition, 2002.
.. [3] Lentner, Marvin, "Elementary Applied Statistics", Bogden
       and Quigley, 1972.
.. [4] Weisstein, Eric W. "Binomial Distribution." From MathWorld--A
       Wolfram Web Resource.
       http://mathworld.wolfram.com/BinomialDistribution.html
.. [5] Wikipedia, "Binomial distribution",
       https://en.wikipedia.org/wiki/Binomial_distribution

Examples
--------
Draw samples from the distribution:

>>> rng = np.random.default_rng()
>>> n, p = 10, .5  # number of trials, probability of each trial
>>> s = rng.binomial(n, p, 1000)
# result of flipping a coin 10 times, tested 1000 times.

A real world example. A company drills 9 wild-cat oil exploration
```

```
wells, each with an estimated probability of success of 0.1. All nine
wells fail. What is the probability of that happening?

Let's do 20,000 trials of the model, and count the number that
generate zero positive results.

>>> sum(rng.binomial(9, 0.1, 20000) == 0)/20000.
# answer = 0.38885, or 39%.
```

We can see that either of the first two arguments can be vectors, allowing us to specify different probabilities (or sample sizes) for each experiment if we wish. So now we can conduct a new poll for each of our experiments with one call to `rng.binomial()`, and use the resulting poll numbers for our simulated elections. We'll simulate each poll assuming that the original poll result, p = 0.51, is our best guess as to the "true" underlying probability.

First, let's reset our generator back to 42.

[41]:
```
rng = reset()
```

Let's set up our constants.

[51]:
```
p = 0.51 # best guess of "true" probabilit
n = 1000 # poll sample size
size = 200 # number of simulations to run
```

And conduct the polls.

[59]:
```
polls = rng.binomial(n, p, size) # get the polling results
```

Now will convert the poll results, which are in terms of total number of people saying they will vote for "A", to probabilities.

[63]:
```
prob = polls/1000 # convert to probabilities
```

In the cell below, look at the first 10 or so probabilities we got.

[65]:
```
prob[:9]
```

[65]:
```
array([0.519, 0.523, 0.507, 0.497, 0.486, 0.527, 0.521, 0.49 , 0.525])
```

Now that we have simulated polling data that reflect realistic poll-to-poll variability, we can simulate the elections.

(re) Set the number of expected voters.

[66]:
```
rng = reset()
voters = 100000 # medium city - expect around 100k voter turnout
```

And conduct and plot the results of the simulated elections! Remember to use your vector of poll probabilities for the probability argument!

[67]:
```
med_poll = rng.binomial(voters, prob)
```

[68]:
```
med_poll
```

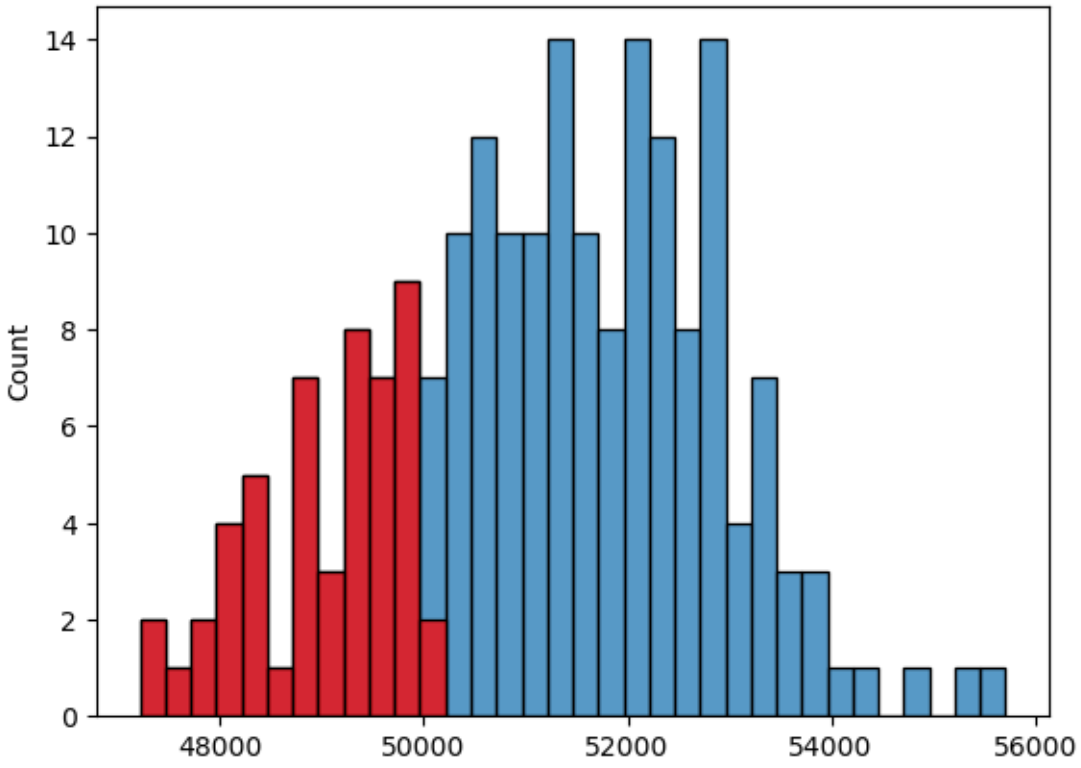
[68]:
```
array([52193, 52397, 50850, 49711, 48724, 52469, 52196, 48668, 52853,
       50502, 49235, 51893, 52079, 51551, 54304, 50227, 48213, 49888,
       51980, 50622, 50437, 49489, 50847, 49814, 49424, 52157, 51305,
       50001, 51449, 51936, 51043, 49130, 51294, 51216, 51737, 52742,
       50221, 51429, 49501, 53770, 52828, 49721, 53365, 49223, 51712,
       52889, 51267, 50262, 52060, 52614, 49881, 52357, 53304, 49249,
       48401, 54954, 52241, 50154, 48162, 49303, 51348, 49053, 51976,
       52728, 51607, 53626, 50282, 49501, 52533, 51566, 52808, 50274,
       52289, 49417, 49555, 53205, 47960, 53243, 50772, 50962, 50849,
       48870, 51692, 52628, 52117, 49879, 52773, 50973, 51319, 50883,
       52886, 52628, 48417, 51731, 51427, 51488, 51780, 50747, 49420,
       50982, 50857, 49671, 52390, 51177, 52622, 52867, 50024, 48299,
       50078, 50649, 51434, 52874, 48883, 51108, 51323, 50655, 49665,
       52179, 49075, 50629, 48157, 47622, 51983, 52206, 49996, 49924,
       51066, 49839, 50032, 51144, 50326, 51425, 52720, 51146, 51463,
       48809, 53334, 50410, 52134, 52879, 53751, 53706, 51784, 55566,
       48906, 48937, 49978, 50439, 50667, 48018, 50677, 49456, 52357,
       52717, 52442, 50279, 50776, 47830, 51768, 51500, 52339, 52251,
       53168, 48249, 50604, 52343, 50479, 52759, 52296, 52320, 54067,
       49930, 53702, 51823, 52613, 53015, 48101, 51284, 51050, 52670,
       53351, 50617, 51377, 52116, 49931, 53421, 51620, 47213, 52985,
       53436, 53921, 50799, 47311, 51694, 48823, 55227, 50707, 50979,
       51992, 50678])
```

Notice that Candidate B now pulls off the upset a substantial number of times!

In the cell below, highlight the upsets by overlaying a histogram of B's wins on the main histogram. Pro tip: use `sns.histplot()` instead of `sns.displot()`, and set the binwidths the same for both.

[95]:
```
b_poll = med_poll[med_poll < 50000]
sns.histplot(med_poll, binwidth = 250)
sns.histplot(b_poll, binwidth = 250, color = 'r')
```

[95]:
```
<AxesSubplot:ylabel='Count'>
```



And compute, in the cell below, the percentage of elections that came out for Candidate B.

[80]:
```
sum(med_poll < 50000)/200
```

[80]:
```
0.255
```

1.5.2 Simulating election results based on a multiple polls

Not all polls are created equally and, in high profile elections such as for US president, many polls by different organizations are conducted to see which way voters are leaning. So faced with, say, three polls with slightly different results, how would we go about forcasting the election?

Data scientists (such as the person behind the website fivethirtyeight) routinely "grade" polls. They do this in part by going back to all the recent elections and seeing how close to the election result each poll was. So what would we do if we had the following three polls?

- Poll X: 53% to 47%, grade = C
- Poll Y: 51% to 49%, grade = A
- Poll Z: 51.5% to 48.5%, grade = B

One thing we could do is simple take poll Y and ignore the other two. The problem with this is that it ignores the information provided by the other two polls – just because they have lower grade doesn't mean they are useless!

The solution is to compute a *weighted average* of the polls, giving each poll more or less influence on the final result based on its grade.

Weighted Averages Weighted averages are often used to compute final grades in a class. So, for example, if a certain class has a grade breakdown of 50% final paper, 25% quizzes, and 25% attendence, and a certain students grades are 95, 85, and 100, respectively, then the final grade would be computed as

[31]:
```
0.5*95 + 0.25*85 + 0.25*100
```

[31]:
```
93.75
```

Or, if we wanted to be a little more clear in the context of Python script:

[32]:
```
weights = np.array([0.5, 0.25, 0.25])
grades = np.array([95, 85, 100])
final_grade = sum(weights * grades)
final_grade
```

[32]:
```
93.75
```

We can even compute a weighted sum when the weights don't sum to 1.0. All we do is convert the weights so that they *do* sum to 1.0 by dividing by the sum of the weights. So, for example, if the above weights were given as 5, 2.5, and 2.5, we could do the following:

[33]:
```
weights = np.array([5, 2.5, 2.5])
sum_of_w = np.sum(weights)
grades = np.array([95, 85, 100])
final_grade = np.sum(weights * grades)/sum_of_w
final_grade
```

[33]:
```
93.75
```

(Notice that, in the first example, we did the same thing in our heads when we turned 50% into 0.5, etc.)

Weighted Polls by Grade To simulate an election based on a weighted sum of polls, we need to convert the letter grades into numbers. The choice here is up to us and somewhat arbitrary. Let's start with the familiar gpa scale where an A is a 4, a B is a 3, etc.

Now that we've figured out our weights, we just need to repeat all the above steps, conducting each poll separately, weighting it, and then combining the results.

Define the probalities, weights, and other things:

[34]:
```
px, py, pz = 0.53, 0.51, 0.515        # actual poll results
wx, wy, wz = 2, 4, 3                  # letter grade weights
sum_of_w = wx + wy + wz               # sum of the weights
samp_sz = 2000                        # poll sample size
```

```
n_sims = 20000                          # number of simulations to run
```

Conduct the polls.

[35]:
```
x_poll_results = rng.binomial(samp_sz, px, n_sims) # polling results for x
y_poll_results = rng.binomial(samp_sz, py, n_sims) # polling results for y
z_poll_results = rng.binomial(samp_sz, pz, n_sims) # polling results for z
```

Convert the poll totals to probabilities.

[36]:
```
x_poll_probs = x_poll_results/samp_sz            # probabilities for x
y_poll_probs = y_poll_results/samp_sz            # probabilities for y
z_poll_probs = z_poll_results/samp_sz            # probabilities for z
```

Set the number of voters.

[37]:
```
n_voters = 100000 # medium city - expect around 100k voter turnout
```

Simulate sets of elections, separately for each poll.

[38]:
```
elec_results_x = rng.binomial(n_voters, x_poll_probs, n_sims)
elec_results_y = rng.binomial(n_voters, y_poll_probs, n_sims)
elec_results_z = rng.binomial(n_voters, z_poll_probs, n_sims)
```

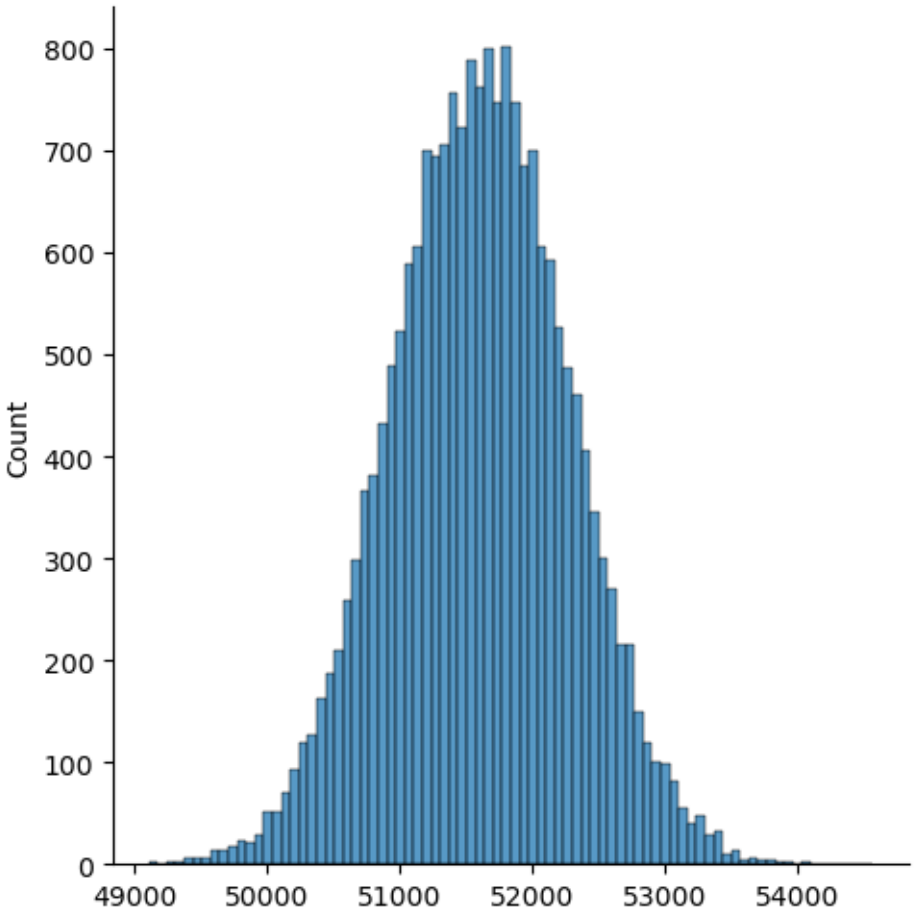
Compute the weighted average.

[39]:
```
elec_results = (wx*elec_results_x + wy*elec_results_y + wz*elec_results_z) /␣
 ↪sum_of_w
```

And plot the results!!!

[40]:
```
sns.displot(elec_results)
```

[40]:
```
<seaborn.axisgrid.FacetGrid at 0x7fe84b204520>
```



Does this look reasonable? Since we now have two polls indicating $> 51\%$ support for Candidate A, we'd expect the distribution to get pulled a little higher, and therefor Candidate B having fewer upsets…

Let's do two more quick reality checks.

First, if we change the weights and/or the initial probabilities dramatically, we should be able to see a clear and sensible change in the outcome.

Go ahead and do this, and describe what you did and what happened as a result:

For a second reality check, consider that, if our simulation worked, then *the mean number of votes for Candidate A should be close to that predicted by the weighted average of the initial probabilities!

Use the cells below to make this comparison.

Mean number of votes for Candidate A:

[97]:
```
med_poll.mean()
```

[97]:
```
51147.635
```

Weighted sum of the initial probabilities:

[101]:
```
results = np.array([0.53, 0.51, 0.515])
weights = np.array([2, 4, 3])
sum(results * weights)/sum(weights)
```

[101]:
```
0.5161111111111111
```

Do they check out?

They're pretty close!
