Chapter 9: Applications to Computing

9.2: Probability via Simulation

Slides (Google Drive) Starter Code (GitHub)

9.2.1 Motivation

Even though we have learned several techniques for computing probabilities, and have more to go, it is still hard sometimes. Imagine I asked the question: "Suppose I randomly shuffle an array of the first 100 integers in order: $[1, 2, \ldots, 100]$. What is the probability that exactly 13 end up in their original position?" I'm not even sure I could solve this problem, and if so, it wouldn't be pretty to set up nor actually type into a calculator.

But since you are a computer scientist, you can actually avoid computing hard probabilities! You could also even verify that your hand-computed answers are correct using this technique of "Probability via Simulation".

9.2.2 Probability via Simulation

We first need to define another notion or way of thinking about a probability. If we had some event E, then we could define $\mathbb{P}(E)$ to be the long-term proportion of times that event E occurs in a random experiment. That is,

$$\frac{\#\text{ of trials where } E \text{ occured}}{\#\text{ trials}} \to \mathbb{P}(E)$$

as the number of trials goes to ∞.

For example, if E is the event we roll a 4 on a fair six-sided die, the probability is $\mathbb{P}(E) = 1/6$. That means, if I were to roll this die 6 million times, I should expect to see about 1 million 4's! In reverse, if I didn't know $\mathbb{P}(E)$ and wanted to compute it, I could just simulate many rolls of this fair die! Obviously, the more trials, the better your estimate. But you can't possibly sit around forever rolling this die - a computer can do this MUCH faster, simulating millions of trials within seconds.

This also works for averages, in addition to probabilities. I think this topic is best taught by examples, so we'll see one of each!

Example(s)

Suppose a weighted coin comes up heads with probability 1/3. How many flips do you think it will take for the first head to appear? Use code to estimate this average!

Solution You may think it is just 3, and you would be correct! We'll see how to prove this mathematically in chapter 3 actually. But for now, since we don't have the tools to compute it, let's use our programming skills!

The first thing we need to do is to simulate a single coin flip. Recall that to generate a random number, we use the `numpy` library in Python.

```
np.random.rand() # returns a single float in the range [0,1)
```

What about this following line of code?

```
if np.random.rand() < p:
```

This might be a bit tricky: since `np.random.rand()` returns a random float between $[0, 1)$, the function returns a value $< p$ with probability exactly p! For example if $p = 1/2$, then `np.random.rand() < 1/2`, which happens with probability $1/2$ right? In our case, we'll want $p = 1/3$, which will execute with probability $1/3$.

This allows us to simulate the event in question: the first "Heads" appears whenever `np.random.rand()` returns a value $< p$. And, if it is $\geq$ `p`, the coin flip turned up "Tails".

The following function allows us to simulate ONCE how long it took to get heads.

```
def sim_one_game() -> int: # return an integer
    flips = 0
    while True:
        flips += 1
        if np.random.rand() < p:
            return flips
```

We start with our number of flips being 0. And we keep incrementing flips until we get a head. So this should return an *integer*! We just need to simulate this game many times (call this function many times), and take the average of our samples! Then, this should give us a good approximation of the true average time (which happens to be 3)!

The code above is duplicated below, as a helper function. Python is great because you can define functions inside other functions, only visible to the parent function!

```
import numpy as np

def coin_flips(p, ntrials=50000) -> float:

    def sim_one_game() -> int: # internal helper function
        flips = 0
        while True:
            flips += 1
            if np.random.rand() < p:
                return flips

    total_flips = 0
    for i in range(ntrials):
        total_flips += sim_one_game()
    return total_flips / ntrials

print(coin_flips(p=1/3))
```

Notice the helper function is the exact same as above! All we did was call it `ntrials` times and return the average number of flips per trial. This is it! The number 50000 is arbitrary: any large number of trials is good! □

Now to tackle the original problem:

Example(s)

Suppose I randomly shuffle an array of the first 100 integers in order: $[1, 2, \ldots, 100]$. What is the probability that exactly 13 end up in their original position? Use code to estimate this probability! Hint: Use `np.random.shuffle` to shuffle an array randomly.

Solution Try it yourself before looking at the answer below!

```python
import numpy as np

def prob_13_original(ntrials=50000) -> float:

    def sim_one_shuffle() -> int: # internal helper function
        arr = np.arange(1, 101) # Creates array: [1,2,...,100]
        np.random.shuffle(arr)

        num_orig = 0 # Count how many elements are in original position
        for i in range(1, 101): # 1,2,...,100
            if arr[i - 1] == i: # Python is 0-indexed
                num_orig += 1

        return int(num_orig == 13) # Returns 1 if True, 0 if False


    num_succ = 0  # Count how many times exactly 13 were in original
    for i in range(ntrials):
        num_succ += sim_one_shuffle()
    return num_succ / ntrials

print(prob_13_original())
```

Take a look and see how similar this was to the previous example!

□