Six sides

Preparing for a data science interview is a daunting prospect, if only because the range of questions that might get asked is so broad: Machine Learning (ML) algorithms, linear algebra, deep learning architectures, SQL syntax, data engineering techniques, MLOps, and more.

On the spectrum of possible questions, statistical problems land somewhere between brain teasers and what’s on LeetCode.

Like brain teasers, statistical problems often start with familiar scenarios—flipping coins, rolling dice, drawing cards—making the problems seem simple on the surface. But their familiarity belies their complexity. By design, job candidates need to think carefully through these types of problems, exploring possibilities and shifting perspectives, before they’re able to find the correct solutions. And much like programming challenges on LeetCode, statistical problems often require the creative application of core concepts (e.g., calculating probabilities, identifying distribution families, estimating confidence intervals, on so on), testing candidates’ basic knowledge and their ability to put theory into practice.

Nonetheless, I find myself increasingly reluctant to use teaser-like problems in data science interviews.

I have several reasons, starting with the fact that Microsoft has long discouraged the use of brain teasers to evaluate candidates. Given the question-answer structure of most interviews, candidates often feel like they have the briefest of moments—the pause in normal conversation—to respond. But brain teasers are only easy when you already know the answers, otherwise they take time to puzzle through. The more time that candidates take to think through their answers, the more pressure they’re likely to feel and the more difficult it becomes for them to focus on solutions. The resulting spiral of negativity undermines the fundamental purpose of teasers, which is to evaluate candidates’ creativity and problem-solving skills, not their ability to provide on-the-spot answers.

Similarly, teaser-like problems often have just one solution, which limits candidates’ opportunity to demonstrate the depth and breadth of their skills. This is a particular challenge given the ever-expanding nature of the data science discipline.

### The coupon collector’s problem

Consider an example: How many times do you need to roll a six-sided die before you get each side at least once?

The question is a variant of the “coupon collector’s problem”. To be clear, a “coupon collector” isn’t someone who’s looking to save money on groceries. In this context, coupons are randomly distributed items (like baseball cards or Happy Meal toys) that are designed to be collected. The problem is to calculate how many packs of cards or Happy Meals a person would typically have to acquire to complete the entire collection.

Using a six-sided die makes the problem a bit more approachable because a die is more universally familiar than baseball cards and Happy Meals, and the size of the collection—only six sides—reduces the numeric complexity.

### Side-wise probabilites

Most candidates will start by determining the probabilities of getting each new side.

The first time you roll the die, you’re certain to get a new side: The probability is 6/6 (100 percent).

The next roll of the die, you’ll still need five sides, so the probability of getting a new side decreases to 5/6 (83 percent). Following that same pattern, the probability of getting the third new side is 4/6 (67 percent).

The probabilities for all six sides is easy to calculate:

| Side | Probability |
| --- | --- |
| 1st | 6/6 (100%) |
| 2nd | 5/6 (83%) |
| 3rd | 4/6 (67%) |
| 4th | 3/6 (50%) |
| 5th | 2/6 (33%) |
| 6th | 1/6 (17%) |

### Role of the geometric distribution

The first real challenge is figuring out how to combine these multiple probabilities into a single result. More specifically, we need to better understand the relationship between probabilities and the number of rolls.

The probability of getting the first side is 100 percent, so it always takes just one roll.

The second side is a bit more interesting. Every attempt to get the second side–every roll of the die—is independent, so the probability that you’ll get a second new side on any *individual* roll is 83 percent. It doesn’t matter if it’s your first roll or your tenth: The probability is always 83 percent.

But how many rolls will it take?

To answer that question, we have to determine the probability of every number of rolls that it might take to get a second new side. For example, to calculate the probability of getting it on *exactly* the second roll, you combine the probability of getting a new face (83 percent) *conditioned on* the probability of *not* getting it on the first roll (17 percent, which is the difference between 100 percent and 83 percent). This—the conditional probability—is simply the product of 83 percent and 17 percent, which rounds to 14 percent.

(5/6 \* 1/6) |>  
 round(2)

## [1] 0.14

The probability of getting the second new side on the *third* roll is the probability of getting a new face, conditioned on the probability of not getting it on the first two rolls. The conditional probability is the product of all three, which rounds to 2 percent.

(5/6 \* 1/6 \* 1/6) |>  
 round(2)

## [1] 0.02

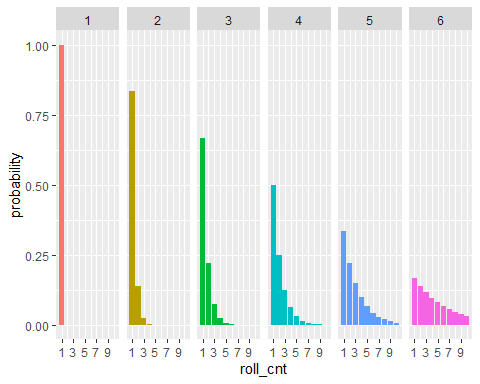
The geometric distribution generalizes this concept, allowing us to calculate the probability of success after any number (*k*) of attempts, given a fixed probability.

In this case, we have six fixed probabilities (i.e., 6/6, 5/6, …, 1/6), one for each new side, and we can use the geometric distribution to calculate the probability of getting each new side after *k* rolls:

# number of sides  
n <- 6  
  
# number of rolls that we'll evaluate for each new side  
k\_range <- 1:10  
  
# the first parameter of R's dgeom() function is the number   
# of failures, so success on the first try would be zero   
# failures, so subtract 1 from the range  
k\_range <- k\_range - 1   
  
# calculate the full range probabilities for n sides   
probability\_matrix <- matrix(n:1, nrow = n) |>   
 apply(1, function(x) dgeom(k\_range, x/n)) |>  
 t()  
  
probability\_matrix |>   
 round(2)

## [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]  
## [1,] 1.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00  
## [2,] 0.83 0.14 0.02 0.00 0.00 0.00 0.00 0.00 0.00 0.00  
## [3,] 0.67 0.22 0.07 0.02 0.01 0.00 0.00 0.00 0.00 0.00  
## [4,] 0.50 0.25 0.12 0.06 0.03 0.02 0.01 0.00 0.00 0.00  
## [5,] 0.33 0.22 0.15 0.10 0.07 0.04 0.03 0.02 0.01 0.01  
## [6,] 0.17 0.14 0.12 0.10 0.08 0.07 0.06 0.05 0.04 0.03

Plotting the geometric distribution for each new side makes it easier to see that, as a “collection” grows, the number of rolls that it will likely take for each new face increases.



Conveniently, the average number of rolls that it takes to get each new side is the inverse of its probability.

The inverse of the probability for the first side is 6/6, so the average number of rolls is 1. The inverse of the probability for the second side is 6/5, so the average number of rolls is 1.2.

And since each new face is an independent effort, we can add these averages to calculate the average number of rolls that it will take to get all six sides.

6/6 + 6/5 + 6/4 + 6/3 + 6/2 + 6/1

## [1] 14.7

Expressed as a formula, the pattern can be generalized to a die of any number (*n*) of sides:

mean\_prob <- function(n){  
 n \* sum(1/(1:n))  
}  
  
mean\_prob(n)

## [1] 14.7

This is as far as a lot of candidates will go. But the problem offers layers of additional complexity.

### Monte Carlo simulation

One way to confirm our expected value is through Monte Carlo simulation. Candidates wouldn’t have this luxury in an interview setting, of course, but they might get asked how to structure a simulation to solve the problem. The technique is remarkably straightforward: Roll a digital die thousands of times and calculate the average number of rolls that it takes to collect all six sides.

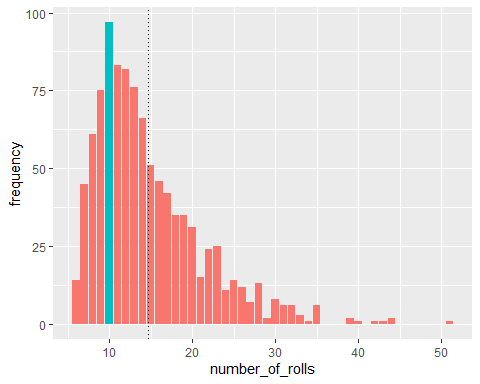
# n = number of sides  
# repetitions = number of times to repeat the simulation  
# seed = random seed  
simulation <- function(n = 6, repititions = 1000, rnd\_seed = 42) {  
 # create a vector to hold simulation results  
 results <- rep(NA, repititions)  
   
 # set a seed to ensure the results are repeatable  
 set.seed(rnd\_seed)  
   
 # conduct the simulation  
 for (i in 1:repititions) {  
 # create a vector to track how many times each side is rolled  
 die <- rep(0, n)  
   
 # roll the die and keep track of the number of times each side comes up  
 # until all sides have come up  
 while (any(die == 0)) {  
 roll <- sample(n, 1)  
 die[roll] <- die[roll] + 1  
 }  
   
 # record the result (i.e., the total number of rolls)  
 results[i] <- sum(die)  
 }  
   
 return(results)  
}  
  
results <- simulation(n = 6, repititions = 1000, rnd\_seed = 42)

The mean of our simulation should be reasonably close to the result of our formula, which it is.

mean(results)

## [1] 14.729

Looking at the distribution of our results as a histogram offers further insight.



First, we’re reminded that we can’t roll a die 14.7 times (the dotted line). We’re really only able to roll a die 14 or 15 times, but the average number of rolls that it will take to get all six sides—over many, many attempts—is 14.7.

Second, the minimum number of rolls is at least six, but it can sometimes take a very large number of rolls to get every side, so our distribution is positively skewed.

max(results)

## [1] 51

skew <- function(x, ...) {  
 mean\_scaled <- x - mean(x, ...)  
 mean(mean\_scaled ^ 3) / sd(x, ...) ^ 3  
}  
  
skew(results)

## [1] 1.437993

As such, the mode—the most common number of times that we’d have to roll a die before getting all six sides—is actually less than 14 or 15. In this case, it’s only 10 rolls.

table(results) |>   
 which.max() |>  
 names() |>  
 as.numeric()

## [1] 10

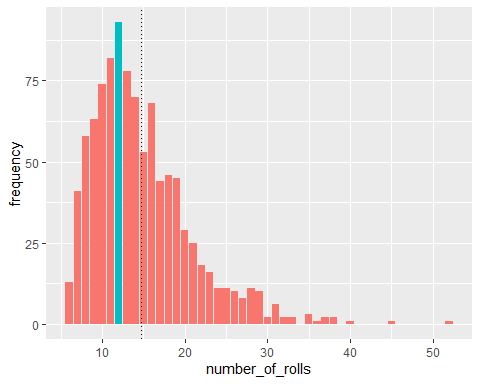
### Mean versus mode

Calling attention to the difference between mean and mode is a potential make-or-break opportunity for candidates. The difference is often important, so astute candidates will ask to clarify what is meant by “how many times”: The *average* number of rolls or the *most common* number of rolls.

If the answer is the latter, the complexity of the problem increases dramatically.

Until we establish an expected value for the mode, we can’t trust the result of our initial simulation. In fact, if we repeat the same simulation with a different seed, we get different results.

# repeat the simulation  
results <- simulation(n = 6, repititions = 1000, rnd\_seed = 59)



The mean is similar:

mean(results)

## [1] 14.748

But the mode has increased from 10 to 12:

table(results) |>   
 which.max() |>  
 names() |>  
 as.numeric()

## [1] 12

### Permutations of roll counts

Looking back at the plot of geometric distributions offers another insight: The highest probability for getting any new side is *always* on the first roll. This is a known property of the geometric distribution: The mode is 1.

But that raises the question: If the mode for each new side is 1, why isn’t the mode for getting all six sides equal to 6 (i.e., the product of 6 \* 1)?

It is, actually, but only if you look at the permutations of all possible roll counts separately.

There’s only one way to roll the die six times:

1 + 1 + 1 + 1 + 1 + 1 = 6.

But there are multiple ways to roll seven times

1 + **2** + 1 + 1 + 1 + 1 = 7  
1 + 1 + **2** + 1 + 1 + 1 = 7  
  **·**  
  **·**  
  **·**  
1 + 1 + 1 + 1 + 1 + **2** = 7

The position of the second roll in the sequence is important, because the probability changes depending on the side that you’re trying to get (second, third, and so on). For example, the probability that two rolls are required to get the *second* side is 0.003, but the probability that two rolls are required for the *third* side is 0.005. (Note, the latter probability is higher, because it’s more difficult to get the third side, and so the probability that it will take more roles is higher.)

# two rolls required to get the second side  
(6/6 \* (1/6 \* 5/6) \* 4/6 \* 3/6 \* 2/6 \* 1/6) |>  
 round(3)

## [1] 0.003

# two rolls required to get the third side  
(6/6 \* 5/6 \* (2/6 \* 4/6) \* 3/6 \* 2/6 \* 1/6) |>  
 round(3)

## [1] 0.005

I want to draw particular attention to the fact the that seven-roll permutations (and any other permutation where *k* > 6) are all products of the six-roll permutation multiplied by at least one additional probability (i.e., the probability of not getting a new side). As such, the probability of any *k* = 6 is always greater than the probability of *k* > 6, confirming that the permutation of rolls with the highest probability is, in fact, *k* = 6.

However, the probability of the six-roll permutation (0.015) is less than the *sum* of the seven-roll permutations (0.039).

# 6-roll probability  
prod(probability\_matrix[, 1]) |>  
 round(3)

## [1] 0.015

# sum of the 7-roll permutations  
(prod(probability\_matrix[, 1]) \* (1 - probability\_matrix[, 1])) |>  
 sum() |>  
 round(3)

## [1] 0.039

### Finding the mode

Calculating the probability for any number (*k*) of total rolls requires us to aggregate the probabilities for all permutations of rolls that add up to *k*. That’s relatively easy for *k* = 7 (which we just did), but it gets more complicated for larger values of *k*, simply because the number of permutations increases rapidly.

| Rolls (*k*) | Permutations |
| --- | --- |
| 6 | 1 |
| 7 | 5 |
| 8 | 15 |
| 9 | 35 |
| 10 | 70 |
| 11 | 126 |
| 12 | 210 |
| 13 | 330 |
| 14 | 495 |

Even with a computer, the task would be difficult, especially if we went all the up to the max number of rolls from our simulation (51). Fortunately, there’s a formula that leverages the principle of inclusion-exclusion to calculate the aggregate probability without having to iterate through the conditional probability of every permutation:

where

*d* = distinct items collected (for *d* ≤ *n* and *d* ≤ *k*)  
*k* = attempts (i.e., rolls)  
*n* = size of the complete set (i.e., sides on the die)

# probability of d unique results, completed on the k-th attempt,  
# where:  
# d = number of different results (e.g., coupons, sides, etc.)  
# k = number of attempts (e.g., draws, rolls, etc.)  
# n = size of complete set  
  
# inclusion-exclusion helper function, where:   
# i = iteration, ranging from 1 to (d - 1)  
permutations\_d <- function(i, d, k) {  
 (-1) ^ (i + 1) \* choose(d, i) \* i \* (d - i) ^ (k - 1)  
}  
  
prob\_d\_k\_exact <- function(d, k, n) {  
 if (d > k | d > n)  
 return(NA)  
 permutations\_d\_sum <- permutations\_d(1:(d - 1), d, k) |>  
 sum()  
 choose(n, d) / (n ^ k) \* permutations\_d\_sum  
}  
  
prob\_d\_k\_exact <- Vectorize(prob\_d\_k\_exact)  
  
# find the probabilities for getting all six sides after 6 - 7 rolls  
prob\_d\_k\_exact(6, 6:7, 6) |>   
 round(3)

## [1] 0.015 0.039

The results of the formula for six- and seven-roll permutations (0.015, 0.039) are the same as the brute-force results that we calculated earlier.

Although we can calculate the probability for any number of rolls, we know that the mode is somewhere between 6 (the minimum number of rolls) and 14.7 (which we can round to 14, because the mode is less than the mean). We simply need to identify the number of rolls with the highest probability over that narrow range:

# find the probabilities for getting all six sides after 6 - 14 rolls  
find\_mode <- prob\_d\_k\_exact(6, 6:14, 6)   
names(find\_mode) <- 6:14  
  
# identify the number of rolls with the highest probability  
which.max(find\_mode) |>  
 labels() |>  
 as.numeric()

## [1] 11

Now we know: The average number of rolls that it takes to get all six sides of a die is 14.7, but the most likely number of rolls is 11.

### Alternative approaches

Ostensibly, the coupon collector’s problem is an excellent interview question because it tests candidates’ knowledge of multiple fundamentals:

* probability
* statistics
* continuous and discrete variables
* expected values
* non-Gaussian distributions
* combinatorics

Some might assert that solving the problem is a min-bar expectation for data scientists.

As I suggested earlier, however, problems like these aren’t particularly good at vetting data science candidates. First and foremost, they test only a narrow range of candidates’ skills and abilities. Moreover, the expectation that candidates can solve teaser-like problem in an hour (or less) is questionable. Even though Medium estimates that this article takes about \_\_\_ minutes to read, there’s a big difference between describing a solution and coming up with one. In fact, there’s a real risk that interviewers—who already know the solutions—might trivialize the difficulty in finding them for the first time (particularly in an interview setting, where time is limited and the pressure is high) and evaluate candidates too harshly.

An alternative approach is the so-called behavioral interview, which focuses much more on candidates and their personal experiences. Interviews typically start with an open-ended prompt: “Tell me about your most recent data science project.” There is no right or wrong answer—candidates can respond however they’d like—so there’s none of pass/fail anxiety that teaser-like problems create. But because candidates have so much latitude, they often end up revealing much more about themselves: the types of work they find most engaging, how (and how well) they work with others, their ability to explain complex topics clearly, and more.

The challenge for interviewers is that *they* need to be familiar with a broad range of techniques—forecasting, natural-language processing, reinforcement learning, and so on—if they’re going to follow candidates down any path. While that might seem daunting at first, interviewers likewise have the freedom to ask about data-quality challenges, algorithms, objective functions, deployment practices, anything to more fully explore candidates’ familiarity with the data science process.

A middle ground is to present candidates with stripped down versions of problems they’d likely face in their day-to-day jobs. The problems need to be simple enough that candidates can solve them in an hour (or an evening), but complex enough to test candidates’ skills and abilities.

Anytime we use the middle-ground approach at Microsoft, we make it clear that don’t expect on-the-spot answers. That’s why we give candidates ample time to come up with solutions. We also make it a point to emphasize that there are no right or wrong answers, in part to mitigate any pass/fail anxiety, but also to suggest that the answers are somewhat beside the point. We’re much more interested in seeing candidates demonstrate the skills that will be critical to their long-term success. The fact that candidates are all asked to solve the same problem makes it a little easier for us to compare approaches. As a bonus, problems like these give us the opportunity to evaluate candidates’ familiarity with our business model (i.e., a subscription-based consumption model), their abilities to solve novel problems (because candidates won’t have seen them before), and candidates’ abilities to identify salient insights and explain their process.

No matter the approach, interviews are often the last—and best—opportunity that teams have to evaluate candidates before making hire/no-hire decisions. To ensure they’re selecting the highest caliber talent, organizations need to be thoughtful about their interview strategies. Questions based on candidates’ own experiences (or the organization’s) will be closer the real-world responsibilities of any role, making it easier for organizations to explore the strengths and weaknesses of candidates as practicing data scientists, which will be much more relevant than candidates’ ability to solve classic statistics problems.