Functions

Ari Anisfeld

8/23/2021

Functions lab

Warm up

1. Let's write a function that takes n-th power of some numeric input.

Recall the form of a function looks like:

```r
name <- function(arg1, arg2) {
  body
}
```

Let's start writing:

a. What's a good name for the function? Replace "name" with your chosen name.
b. The function requires a numeric input and a power. What are good names for these two arguments? Replace the args with your argument names.
c. Finally, write the body.
d. Test that the code works.
e. You want the default power to be 2. Adjust the code to make it so.

Answer:

```r
power <- function(x, n = 2){
  x^n
}
```

2. If you write a function without an explicit `return()` call, what does R return?

Answer: The final line of code in the function.

Random Tips for Functions

In general, trying to write code can sometimes get pretty hairy. A useful approach you can take is to break up your problem into bite sized pieces, try the components separately, and then put them together. You can also `print()` the output of these smaller components as you go to figure out if and where the code is doing something unintended. You'll see some of this philosophy in this lab. It might feel more complicated than previous ones, but with some patience you'll get there!

Simulating Data with Monte Carlo Simulations

This is a preview of material you'll see in Stats I where you will be asked to investigate statistical concepts using Monte Carlo simulations. We'll try not to get too technical in the main body of the lab. There are some "technical notes" which you can ignore!

In a monte carlo simulation, you repeatedly:

1. Generate random samples of data using a known process.
2. Make calculations based on the random sample.
3. Aggregate the results.

Functions and loops help us do these repetitious acts efficiently, without repeatedly writing similar code or copying and pasting.

Today's problem: Let us investigate how good the random number generator in R is.[1] We hypothesize that `rnorm(n, mean = true_mean)` provides random sample of size `n` from the normal distribution with mean = `true_mean` and standard deviation = 1.

The lesson is organized as follows.

1. We do a single simulation.
2. We take the logic of the simulation, encapsulate it in functions and then run 1000s of simulations!

A single simulation

Recall our hypothesis is that `rnorm()` faithfully gives us random numbers from the normal distribution. If we test this with a single random draw, we might be misled. For example, let's draw 30 numbers from a normal distribution with true mean of 0.5 and see if the observed mean appears statistically different from the true mean.

```
# Setting a seed ensures replicability
set.seed(4)

# we set our parameters
true_mean <- .5
N <- 30

# We simulate and observe outcomes
simulated_data <- rnorm(N, mean = true_mean) # the standard deviation is 1 by default!
obs_mean <- mean(simulated_data)
obs_mean
```

```
## [1] 0.9871873
```

Wow! The observed mean is twice what we expected given `true_mean`! Let's calculate a z-score to put that in perspective. (Focus on the formulas, you'll learn the intuition in stats class.[2]

A z-score is calculated $\frac{\bar{X}-\mu}{\frac{s_n}{\sqrt{N}}}$ where $\bar{X}$ is the sample mean, μ is the true mean, s_n is the sample standard deviation and N is the number of observations.

```
obs_sd <- sd(simulated_data)
zscore <- (obs_mean - true_mean) / (obs_sd / sqrt(N))
zscore
```

---

[1]Contrived? yes. This is similar to the problem where you have a coin and want to tell if it's fair. ON the other hand, there's a deeper statistical idea here related to the difference between t-tests and z-tests!

[2]**Techincal note**: If you can't help but be curious about the stats, what we are doing is logically equivalent to a 1000 z-tests.

```
## [1] 3.303849
```

We expect the observed mean of this simulated data will be within 1.96 standard deviations of μ 95 out of 100 times.[3] This observation is 3.3 standard deviations from μ. The probability of that happening by chance is very small. To be more formal about this probability, we can calculate a p-value. Plug in the z-score below:

```
(1 - pnorm(abs(zscore))) * 2
```

```
## [1] 0.000953672
```

This says that the probability of getting this draw by chance is less than 0.1 percent or 1 in 1000. [4]

That outcome seems surprising, but we could also just have made an unusual draw. In this workshop, we want to see how often we get such extreme results. We will repeat the steps above 1000 times each, but first we'll write functions that will make this process smooth!

Writing Helper Functions to Make Our Monte Carlo Simulation

```
# You might need some tidy functions later.
library(tidyverse)
```

We want to develop functions that automate repeated steps in our Monte Carlo. In that way, we can define a few important parameters and run the entire process without rewriting or copying and pasting code over and over again.

As you saw in the motivating example, we must do the following a 1000 times or `B` times if parameterize the number of iterations with `B`:

1. Simulate data and calculate sample statistics.
2. Determine z-scores.
3. Test whether the z-score is outside the threshold.

Finally, we:

4. Measure to what extent our simulations match the theory.

To proceed, we'll write the steps into their own functions, then call them in the right order in the function `do_monte_carlo()`. We are breaking a complicated process into smaller chunks and tackling them one by one!

Let's look at `do_monte_carlo()`. It takes a sample-size N, a `true_mean`, number of iterations `B` (1000 by default) and a significance level `alpha` (.05 by default). It returns the proportion of observed means that are significantly different from the `true_mean` with 95 percent confidence level[5]

1. Before following our road map, think about how you would set up functions to automate this process. What would the inputs and outputs be of each step/function? Your processes will be different from ours, but that doesn't mean ours is better.

Answer: As we said there are many ways to approach this!

---

[3]Technical note: This is only approximately correct as N gets large or "asymptotically".

[4]**Technical note:** Again, you'll learn why this is the formula in stats 1! For the curious, `pnorm()` takes a z-score as the input and returns the probability of observing a value less than or equal to the z-score. So if X is distributed standard normal, $pnorm(z) = P(X \leq z)$. This is an integral that measures the area under the probability density function. To see this in action, you can checkout this link

[5]technically, `1 - alpha` percent confidence level

Solution

```r
get_zscores <- function(obs_mean, true_mean, obs_sd, N) {
  (obs_mean - true_mean) / (obs_sd / sqrt(N))
}

test_significance <- function(zscores, alpha) {
  abs(zscores) > qnorm(1 - alpha/2)
}

calc_mean_and_sd_from_sample <- function(N, true_mean) {
  my_sample <- rnorm(N, true_mean)
  tibble(mean = mean(my_sample), sd = sd(my_sample))
}

make_mc_sample <- function(N, true_mean, B) {

  # preallocate output
  out <- vector("list", B)

  for (i in 1:B) {
    out[[i]] <- calc_mean_and_sd_from_sample(N, true_mean)
  }

  bind_rows(out)
}
```

1. Which observation from `made_up_zscores` is not statistically different from 4.3 with 95 percent confidence? In other words, which observed mean and standard deviation return |z-score| < 1.96?

Answer: Draw number 4, with an observed mean of 4.4, and standard deviation of 1

Check for Significance

1. For example, for a two-tailed z-test at the 95% confidence level, the cutoff is set at 1.96. Verify this using the formula above.[6]

   Answer:

   ```r
   abs(qnorm(.05/2))
   ```

   ```
   ## [1] 1.959964
   ```

Challenge:

```r
get_t_stat <- function(obs_mean, true_mean, obs_sd, N) {
  (obs_mean - true_mean) / (obs_sd / sqrt(N))
}

t_test_significance <- function(zscores, alpha, N) {
  abs(zscores) > qt(1 - alpha/2, N)
}
```

---

[6]Hint: alpha = .05

```r
calc_mean_and_sd_from_sample <- function(N, true_mean) {
  my_sample <- rnorm(N, true_mean)
  tibble(mean = mean(my_sample), sd = sd(my_sample))
}

make_mc_sample <- function(N, true_mean, B) {

  # preallocate output
  out <- vector("list", B)

  for (i in 1:B) {
    out[[i]] <- calc_mean_and_sd_from_sample(N, true_mean)
  }

  bind_rows(out)
}


do_monte_carlo_t <- function(N, true_mean, B= 1000, alpha = .05){
  # step 1: Simulate B random samples and calculate sample statistics
  sample_statistics <- make_mc_sample(N, true_mean, B)
  # step 2: Determine z-scores
  t_stats <- get_t_stat(sample_statistics$mean, true_mean, sample_statistics$sd, N)
  # step 3: Test whether the z-scores are outside the threshold.
  significance <- test_significance(t_stats, alpha, N)
  # step 4: Measure to what extent our simulations match the theory. (We expect a number close to alpha
  mean(significance)
}
```