Instructions

In this week’s studio, you will be learning how to write programs that run simulations in R. Scientists develop computer simulations to help them obtain a deeper understanding of a topic. For example, meteorologists simulate weather patterns to better understand climate change. In this studio, you will use the tools of R to simulate numerical data to help you better understand two fundamental ideas in statistics: the law of large numbers and the central limit theorem.

Upon completion of all tasks, a TA will give you credit for today’s studio. If you do not manage to complete all the assigned work during the studio period, do not worry. You can continue to work on this assignment until the following Wednesday at 6 PM. Come by TA hours any time before then to show us your completed work and get credit for today’s studio.

Objectives

By the end of this studio, you will understand two fundamental ideas in statistics: the law of large numbers and the central limit theorem. You will also be more comfortable with programming, having programmed two simulations.

Part 1: Central Limit Theorem Simulation

Simulations are a powerful programming tool. They allow programmers to explore randomly-generated data to better understand any patterns that lurk therein.

The Central Limit Theorem states that the distribution of the sum of a large number of independent, identically distributed variables is approximately normal, regardless of the underlying distribution. For example, if you repeatedly flip a fair coin 1000 times, the sampling distribution of the number of heads should eventually look like a bell curve that peaks at 500, since the probability of heads is 0.5. Your next task is to simulate coin-flip experiments in R many many times, thereby empirically verifying the Central Limit Theorem.

R has a number of built-in functions that generate random numbers, and can thus be used to simulate random experiments. In your simulations, you should represent Heads by 1, and Tails by 0. Then, to simulate 50 coin flips, you can use the `sample` function to create a vector of 50 numbers sampled uniformly at random from the numbers 0 and 1:

```r
sample(0:1, 50, replace = TRUE)
```
The `sample` function randomly draws integers from a given range (in this case, it chooses between 0 and 1, as specified by 0:1). The number of samples to be drawn is determined by the second parameter (in this case, 50 samples are drawn). The `replace` parameter can be set to either `TRUE` or `FALSE`, meaning sample with or without replacement. Sampling with replacement, as we do here, allows the same integer to be drawn multiple times.

Aside: The only time `replace` can be set to `FALSE` is when the number of draws is less than or equal to the size of the range of draws (e.g., `sample(1:1000, 10)`), which is not usually the case for Bernoulli trials. If we were to set `replace` to `FALSE`, we would not be able to sample more than one head or more than one tail!

Create another sample exactly as before, and store your sample in `sample_data`. Next, calculate the mean of `sample_data`. Is it as you expected? You can also visualize `sample_data`. What sort of visualization makes sense for these data? If you are thinking histogram, you are correct. Using the `hist` command, create a histogram of `sample_data`. Title your histogram “Fair Coin”. How many 1s and how many 0s do you observe?

Perhaps the number of 1s and 0s is not exactly as you expected. Rerun the code you just wrote to generate, summarize, and visualize yet another sample. How does the new results compare to the old? Repeat this exercise a few more times.

Hint: To automatically run your code repeatedly, you can embed it in a for loop: ```R
for (i in 1:5) { sample_data <- sample(0:1, 50, replace = TRUE) # mean # histogram }
``` What happens when you execute this for loop? It might be hard to tell, but 5 histograms should have been created. The trouble is, each one was plotted on top of the last, so you can only see the very last histogram. This isn’t very useful. In order to force your program to pause along the way, so that you can see each histogram, type `par(ask = TRUE)` into the RStudio console and hit enter. Rerun your for loop, and observe each histogram.

You now have a functioning random data generator that simulates 50 fair Bernoulli trials and 5 experiments. What if you want to run a sample for any number of trials and any number of experiments? Let’s abstract out these two numbers into two variables, `num_trials` and `num_expts`.

Create the variables `num_trials` and `num_expts`, set them equal to 50 and 5, respectively. Then replace all occurrences of the numbers 50 and 5 in your code with these variables instead. Rerun your program. It should function exactly as before.

Now, just for fun, redefine `num_trials` and `num_expts` to some other values (like 100 and 10). Rerun your code. Wasn’t it easy to make those changes? That is the power of abstraction!

But what happens if you want to run your code for various settings of `num_trials`
or num_expts (as you will need to do to empirically verify the central limit theorem). Should you make lots of copies of your for loop, and reset the values of num_trials and num_expts in between each copy? No! You should not! Repeat code is never a good idea. It complicates debugging, which is already the most difficult part of coding. A better way is to embed your for loop in a function that depends on the values of num_trials and num_expts, and then to repeatedly call your function, with different settings of these parameters.

Embed all your code in a function that takes as input num_trials and num_expts. You can call it something like fair_coin_flips:

```r
fair_coin_flips <- function(num_expts, num_trials) {
}
```

The function you just wrote simulates num_expts random experiments, each of which flips a coin for num_trials trials. Your code also computes the sample mean of each experiment. The central limit theorem says that these sample means are normally distributed. Hence, to verify the theorem, you need to plot a histogram of sample means. In order to do so, you first need to store all the sample means you compute somewhere. You will do so by creating a vector of length num_expts, and then storing the sample mean of each experiment in the vector.

Create a new function clt that takes as input num_expts and num_trials like your earlier function does. At the start of this function, create a vector in which to store the sample means:

```r
sample_means <- numeric(num_expts)
```

This line of code creates a vector of numerics (numbers) of length num_expts, and gives it the name sample_means.

Next, write a for loop that does two things: first, runs an experiment (i.e., generates sample data), and second, stores the mean of that experiment in the sample_means vector.

*Hint:* After creating a vector, say my_vector <- numeric(100), you can store a value in it by indexing into it as follows: my_vector[1] <- 1 or my_vector[100] <- 17. Note that you cannot index into a vector using indices outside the vector’s range, which in this example is 1 to 100, since my_vector is of length 100.

Outside the for loop, print the vector of sample_means, to verify that your code is working correctly. (You can also print this vector inside the for loop for debugging purposes.)

Finally, plot a histogram of the sample_means vector. Note that this histogram must be created outside the for loop, after all the experiments have been run, and the sample_means vector is fully populated.

In summary, your clt function should obey this structure:

```r
clt <- function(num_expts, num_trials) {
   sample_means <- ...
for (i in 1 : num_expts) {
    sample_data <- ...
    sample_means[i] <- ...
}

    print(sample_means)
hist(...)  
}

Finally, create a variable trials and set it to equal to, say, 100. Then, invoke your clt function multiple times:

```
clt(10, trials)
clt(100, trials)
...
clt(10000, trials)
```

What do you notice about the successive histograms? Does your program empirically verify the central limit theorem?

**Seventh-inning stretch.** Stand up and take a deep breath to clear your head a bit. When you sit down again, you should be ready to simulate the central limit theorem!

**Part 2: Law of Large Numbers Simulation**

Your next task is to simulate the law of large numbers. When you run an experiment, the average of the results should, at least in principle, be close to the expected value. The law of large numbers formalizes this claim, stating that as the number of trials approaches infinity, the mean approaches the expected value.

In order to simulate the law of large numbers, let’s simulate a bunch of Bernoulli trials. A Bernoulli trial is an experiment with exactly two possible outcomes: e.g., yes or no, or success or failure, etc.. Once again, we’ll look at coin flips (i.e., heads or tails).

Create a variable num_trials and set it equal to a reasonable number of trials, like 10, or 100, or 1000.

Now let’s run num_trials Bernoulli trials, and store the results in fair_coin_flips.

```
fair_coin_flips <- sample(0:1, num_trials, replace = TRUE)
```

As we sampled from 0:1, the fair_coin_flips vector consists of 0s and 1s.

If you flip a fair coin 100 times, how many heads do you expect to see?

Analogously, if heads are coded as 1s, and tails as 0s, what do you expect the mean of fair_coin_flips to be?
You should expect the mean to be 0.5, since that would mean half the coin flips landed on heads, and half landed on tails. This is, after all, a fair coin.

Examine what your Bernoulli trial produced:

```r
mean(fair_coin_flips)
```

Does the mean equal 0.5? Probably not. Run it again and again. Does it ever equal 0.5?

Try increasing the value of `num_trials`. Does it get closer?

Let’s do this more systematically. Write a for loop with `i` ranging from 1 to `num_trials`, which, on the `i`th iteration, flips `i` coins, and then stores the mean of those `i` coin flips in a vector called `sample_means`:

```r
fair_coin_flips <- sample(0:1, i, replace = TRUE)
sample_means[i] <- mean(fair_coin_flips)
```

Be sure to create the numeric vector `sample_means` before your for loop; otherwise, R won’t recognize the variable name `sample_means`:

```r
sample_means <- numeric(num_trials)
```

Finally, plot the `sample_means`.

Run your for loop and plot the results several times, varying the number of trials, as above. What do you observe? As you increase the number of trials, does the sample mean approach its expected value of 0.5?

Let’s redo these simulations for unfair coins. Add the following parameter to the sample function: `prob = c(p, q)`, where `p` and `q` are non-negative numbers that sum to 1. Recall that the sample function is sampling from the values 0 and 1. The `prob` parameter specifies that 0 should be sampled with probability `p` and 1, with probability `q`.

As above, run `num_trials` Bernoulli trials, but this time store the results in `unfair_coin_flips`. What is the mean of `unfair_coin_flips`, and how does this value relate to `p` and `q`? Create a few sample vectors, for a few different values of `p` and `q`, and calculate their sample means. How do the sample means vary with `p` and `q`, and with the number of trials?

Make a copy of your for loop, and replace the line of code that samples from an unbiased coin with one that samples from a biased one. Run the loop, and plot the results. Observe that the graph now converges to `p` rather than 0.5.

*Note:* Copying code is bad practice! What could you have done here rather than make a copy of your for loop? (*Hint:* Think abstraction.)

Finally, let’s wrap this for loop in a function that takes an input an arbitrary value of `p` (between 0 and 1, inclusive), and a number of trials, `num`. Here is the header for this function, assuming it is named `loln`:
loon <- function(p, num) {
    # for loop
}

Run your function a few times for a fixed value of p, varying the number of trials (e.g., try 10, 100, 1000, and 10000). Do this again, for a few other values of p. Summarize your results.

End of Studio

When you are done please call over a TA to review your work, and check you off for this studio. If you do not finish within the two hour studio period, remember to come to TA office hours to get checked off.