R Refresher Session: Simulation the Tidy Way

Frank DiTraglia

Oxford Econometrics Summer School

Introduction

This practical session provides an overview of how to carry out a basic simulation study in R using tools from the `tidyverse` family of R packages. We assume basic familiarity with R at the level of Hands-On Programming with R. If you need a quick refresher, you may find it helpful to consult my notes on this book. Some basic familiarity with `dplyr` and `ggplot2` would also be helpful. The first two lessons of https://empirical-methods.com provide a quick overview.

Two extremely useful packages for carrying out simulation studies that we will discuss below are `purrr` and `furrr`, which provides functions equivalent to those from `purrr` that run in *parallel*. Because we will only have time to scratch the surface of these packages, I suggest consulting the preceding links for more information. You may also find the `purrr` cheatsheet helpful.

A Biased Estimator of σ^2

My introductory statistics students often ask me why the sample variance, S^2, divides by $(n-1)$ rather than the sample size n:

$$S^2 \equiv \frac{1}{n-1}\sum_{i=1}^{n}(X_i - \bar{X}_n)^2$$

The answer is that dividing by $(n-1)$ yields an *unbiased estimator*: if $X_1, ..., X_n$ are a random sample from a population with mean μ and variance σ^2, then $\mathbb{E}[S^2] = \sigma^2$. So what would happen if we divided by n instead? Consider the estimator $\widehat{\sigma}^2$ defined by

$$\widehat{\sigma}^2 \equiv \frac{1}{n}\sum_{i=1}^{n}(X_i - \bar{X}_n)^2.$$

If $X_i \sim \text{Normal}(\mu, \sigma^2)$ then $\widehat{\sigma}^2$ is in fact the *maximum likelihood estimator* for σ^2. With a bit of algebra, we can show that $\mathbb{E}[\widehat{\sigma}^2] = (n-1)\sigma^2/n$ which clearly does *not* equal the population variance.[1] It follows that

$$\text{Bias}(\widehat{\sigma}^2) \equiv \mathbb{E}[\widehat{\sigma}^2 - \sigma^2] = -\sigma^2/n$$

so $\widehat{\sigma}^2$ is biased *downwards*. Because the bias goes to zero as the sample size grows, however, it is still a consistent estimator of σ^2.

An Example Simulation Study

Another way to see that $\widehat{\sigma}^2$ is biased is by carrying out a simulation study. To do this, we generate data from a distribution with a *known* variance and calculate $\widehat{\sigma}^2$. Then we generate a *new* dataset from the same distribution and again calculate the corresponding value of $\widehat{\sigma}^2$. Repeating this a large number of times, we end up with many estimates $\widehat{\sigma}^2$, each based on a dataset of the same size drawn independently from the same population. This collection of estimates gives us an *approximation* to the sampling distribution of $\widehat{\sigma}^2$. Using this approximation, we can get a good estimate of $\text{Bias}(\widehat{\sigma}^2)$ by comparing the sample mean of our simulated estimates $\widehat{\sigma}^2$ to the *true* variance σ^2. Since we already know how to calculate the bias directly, this is overkill, but it's a nice example for illustrating the key steps in carrying out a simulation study.

If you're already familiar with simulation studies, the approach I take below may seem a little odd. For example, there are no `for` or `while` loops at any point in this code. Instead I take a "tidy" approach based on high-level functional programming abstractions provided by `purrr` and store simulation results as list columns. If you don't know what any of this means, don't worry! Everything will become clear in a moment when we walk through an example. If you do know what this means and are wary, trust me: this is a much faster, cleaner, and saner way to work.

Step 0 - Set the Seed

"Random" draws from a computer are not in fact random: they are *perfectly deterministic* but cleverly constructed to ensure that they look and behave *as if* they were truly random. Whenever you run a simulation study, it's good practice to take advantage of this fact and "set the seed" of the random number generator before running your simulation study. This ensures that if you re-run your simulation tomorrow you'll get exactly the same results. Pick a positive integer, and supply it as the argument of `set.seed()` as follows:

```
set.seed(1983)
```

Step 1 - Write a Function to Generate Simulation Data

This function returns a vector of `n` standard normal draws with variance `s_sq`. Notice that R parameterizes the normal distribution using the *standard deviation* rather than the variance:

[1] To see this, first rewrite $\sum_{i=1}^n (X_i - \bar{X}_n)^2$ as $\sum_{i=1}^n (X_i - \mu)^2 - n(\bar{X}_n - \mu)^2$. This step is just algebra. Then take expectations, using the fact that the X_i are independent and identically distributed.

```
draw_sim_data <- function(n, s_sq) {
  rnorm(n, sd = sqrt(s_sq))
}
```

See `?rnorm` for more details. We allow the simulation parameters `n` and `s_sq` to vary so that we can explore how these affect the bias of $\hat{\sigma}^2$.

Step 2 - Write a Function to Calculate Your Estimate

This function calculates $\hat{\sigma}^2$ as defined above:

```
get_estimate <- function(x) {
  sum((x - mean(x))^2) / length(x) # divides by n not (n-1)
}
```

Again, if you're unsure about any of the R functions used, you can always get help by entering `?` followed by the name of the command at the R console, e.g. `?sum`.

Step 3 - Run the Simulation for Fixed Parameter Values

This step is a bit more complicated, so we'll break it into parts. The first thing we need to do in a simulation study is generate a large number of replicate datasets, each with the same parameter values. We can do this using the function `rerun()` from the `purrr` package as follows:

```
library(tidyverse)
n_reps <- 5
sim_reps <- rerun(n_reps, draw_sim_data(5, 1))
sim_reps
```

```
## [[1]]
## [1] -0.01705205 -0.78367184  1.32662703 -0.23171715 -1.66372191
##
## [[2]]
## [1]  1.99692302  0.04241627 -0.01241974 -0.47278737 -0.53680130
##
## [[3]]
## [1]  0.1334630 -0.9277063  2.2074408 -0.5044774 -0.7275908
##
## [[4]]
## [1]  0.593223401  0.154716749 -0.720989534 -0.130735800 -0.004721653
##
## [[5]]
## [1] -1.5804783 -1.2597907 -1.0548884  0.3127123 -0.1062695
```

Notice that `rerun()` returns a *list* of datasets. You might be tempted to loop over these and use `get_estimate()` to calculate $\hat{\sigma}^2$. That's exactly what we'll do, but we won't write the loop explicitly. Instead we'll use `map_dbl()` from `purrr` to "map" over the simulation replications:

```
map_dbl(sim_reps, get_estimate)
```

```
## [1] 0.9641819 0.8588692 1.3057141 0.1820873 0.5171066
```

Isn't that simpler than writing an explicit `for` loop? The function `map_dbl()` always returns a numeric vector. In contrast the plain vanilla function `map()` returns a *list*

```
map(sim_reps, get_estimate)
```

```
## [[1]]
## [1] 0.9641819
##
## [[2]]
## [1] 0.8588692
##
## [[3]]
## [1] 1.305714
##
## [[4]]
## [1] 0.1820873
##
## [[5]]
## [1] 0.5171066
```

A list is more general, but it's not what we want here. See the `purrr` cheatsheet for some more discussion.

Sometimes something goes wrong in a single simulation replication, causing the whole thing to crash. For example, if you simulate a logistic regression and obtain a simulation draw with *perfect separation* (the Hauck-Donner phenomenon), the maximum likelihood estimator will not exist. If you want to ensure that `map()` keeps running and merely keeps track of the "bad" dataset, you can use the functions `safely()` and `possibly()` from `purrr`. Here's a simple example:

```
c(-1, 3, 4) %>%
  map_dbl(possibly(log, NA))
```

```
## Warning in .f(...): NaNs produced
```

```
## [1]       NaN 1.098612 1.386294
```

In our example, there's no way for `get_estimate()` to throw an error so this isn't needed, but it's good to be aware of.

Next we'll create a function that uses `rerun()` to create a large number of simulation datasets, and then uses `map_dbl()` combined with `get_estimate()` to actually *run* our simulation study:

```
get_estimates <- function(n, s_sq, nreps = 5000) {
  rerun(nreps, draw_sim_data(n, s_sq)) %>%
    map_dbl(get_estimate)
}
```

Notice that I set the number of simulation replications, `nreps`, to 5000 by default but allow the user of `get_estimates()` to change this argument if desired. Now we can run our simulation study at fixed parameter values and calculate, e.g., the estimated bias of $\widehat{\sigma}^2$:

```
sims <- get_estimates(5, 1)
mean(sims) - 1 # calculate the bias of the estimator
```

```
## [1] -0.1882871
```

This agrees quite well with the analytical result of $-1/5$. But ideally we'd like to run the simulation over a *range* of parameter values. That's what we'll do in the next step.

Step 4 - Run the Simulation over a Grid of Parameters

First we'll set up a grid of values for the parameters `n` and `s_sq`. These are the *arguments* that we'll pass to `get_estimates()`. To do this, we'll use the function `expand_grid()` from `tidyr`, available as part of the `tidyverse`

```
sim_params <- expand_grid(n = 3:5, s_sq = seq(from = 1, to = 3, by = 0.5))
sim_params
```

```
## # A tibble: 15 x 2
##        n  s_sq
##    <int> <dbl>
##  1     3   1
##  2     3   1.5
##  3     3   2
##  4     3   2.5
##  5     3   3
##  6     4   1
##  7     4   1.5
##  8     4   2
##  9     4   2.5
## 10     4   3
## 11     5   1
## 12     5   1.5
## 13     5   2
## 14     5   2.5
## 15     5   3
```

Now, again you might be tempted to say that we should loop over the rows of `sim_params` to carry out the full simulation. This is *basically* what we'll do, but there are two wrinkles. First, we won't use an *explicit* loop; instead we'll use the function `pmap()` from `purrr`. Second, we'll *attach* all of the resulting simulated values of $\widehat{\sigma}^2$ to `sim_params` as *list columns*, forming a tibble of simulation results called `sim_results` as follows

```
sim_results <- sim_params %>%
  mutate(sims = pmap(., get_estimates))
sim_results
```

```
## # A tibble: 15 x 3
##        n  s_sq sims
##    <int> <dbl> <list>
##  1     3   1   <dbl [5,000]>
##  2     3   1.5 <dbl [5,000]>
##  3     3   2   <dbl [5,000]>
##  4     3   2.5 <dbl [5,000]>
##  5     3   3   <dbl [5,000]>
##  6     4   1   <dbl [5,000]>
##  7     4   1.5 <dbl [5,000]>
##  8     4   2   <dbl [5,000]>
##  9     4   2.5 <dbl [5,000]>
## 10     4   3   <dbl [5,000]>
## 11     5   1   <dbl [5,000]>
## 12     5   1.5 <dbl [5,000]>
## 13     5   2   <dbl [5,000]>
## 14     5   2.5 <dbl [5,000]>
## 15     5   3   <dbl [5,000]>
```

The dot `.` in the mutate statement is just shorthand for `sim_params`, the first argument of `mutate()` that is passed via the pipe `%>%`. For more details on `pmap()` see the purrr cheatsheet Notice how much simpler this was than writing an explicit loop. There's another advantage to this approach that we'll see in the next step.

Step 5

The advantage of "attaching" the simulation results as list columns in the previous step is that we can now use `map()` type operations to manipulate the simulation draws for $\widehat{\sigma}^2$ in any way that we desire. For example:

```
sim_results %>%
  mutate(sim_mean = map_dbl(sims, ~ mean(.x)),
         sim_var = map_dbl(sims, ~ var(.x)),
         sim_bias = sim_mean - s_sq,
         theoretical_bias = -s_sq / n)
```

```
## # A tibble: 15 x 7
##        n  s_sq sims          sim_mean sim_var sim_bias theoretical_bias
##    <int> <dbl> <list>           <dbl>   <dbl>    <dbl>            <dbl>
```

```
## 1      3   1   <dbl [5,000]>    0.666   0.452   -0.334                -0.333
## 2      3   1.5 <dbl [5,000]>    0.978   0.950   -0.522                -0.5
## 3      3   2   <dbl [5,000]>    1.34    1.71    -0.656                -0.667
## 4      3   2.5 <dbl [5,000]>    1.66    2.72    -0.840                -0.833
## 5      3   3   <dbl [5,000]>    1.98    3.86    -1.02                 -1
## 6      4   1   <dbl [5,000]>    0.751   0.376   -0.249                -0.25
## 7      4   1.5 <dbl [5,000]>    1.11    0.844   -0.387                -0.375
## 8      4   2   <dbl [5,000]>    1.51    1.50    -0.490                -0.5
## 9      4   2.5 <dbl [5,000]>    1.89    2.39    -0.610                -0.625
## 10     4   3   <dbl [5,000]>    2.25    3.36    -0.755                -0.75
## 11     5   1   <dbl [5,000]>    0.795   0.317   -0.205                -0.2
## 12     5   1.5 <dbl [5,000]>    1.23    0.770   -0.272                -0.3
## 13     5   2   <dbl [5,000]>    1.59    1.30    -0.407                -0.4
## 14     5   2.5 <dbl [5,000]>    1.97    1.95    -0.526                -0.5
## 15     5   3   <dbl [5,000]>    2.41    2.84    -0.588                -0.6
```

Note the use of the "`purrr`-style" function syntax, e.g. `~ mean(.x)`. The "column" `sims` is actually a *list* of numeric vectors, and we are using `map_dbl()` to iterate over it. In particular, we compute the mean and variance of each list item. This corresponds to the simulation carried out a particular combination of parameter values, giving us a simple and nicely-formatted table of results.

Bonus Step: Running it in Parallel

Suppose we wanted to use a larger parameter grid, e.g.

```r
many_params <- expand_grid(n = 3:10, s_sq = seq(from = 1, to = 3, by = 0.1))
many_params
```

```
## # A tibble: 168 x 2
##        n  s_sq
##    <int> <dbl>
## 1      3   1
## 2      3   1.1
## 3      3   1.2
## 4      3   1.3
## 5      3   1.4
## 6      3   1.5
## 7      3   1.6
## 8      3   1.7
## 9      3   1.8
## 10     3   1.9
## # ... with 158 more rows
```

Depending on the details of our simulation, things could get fairly slow as we increase the parameter grid in this way. But notice that each combination of parameter values is really a *separate simulation study*. There's no "dependence" between then: we could in principle run each of them on a different computer. This is what is called an "embarrassingly parallel" problem. If your machine has multiple cores, it's worth using them to speed things up.

One important point to note is that parallel processing involves fixed costs: your machine needs to break the problem up into pieces, distribute the pieces to different "workers" and then collect the results. This means that running something in parallel may *not* be faster than running it in serial. As the run time increases, the situation becomes more favorable, since we can effectively "amortize" the fixed cost over the overall run time.

We'll use the package `furrr` to replace `pmap()` with an equivalent function that runs in parallel on machines with multiple cores. First let's time the serial (non-parallel) version on the expanded parameter grid:

```
set.seed(4321)
system.time(
  results_serial <- many_params %>%
    mutate(sims = pmap(., get_estimates))
)
```

```
##    user  system elapsed
##   7.128   0.000   7.129
```

Now we'll use `furrr`. There are only three differences you need to be aware of. First, we need to set the random seed in a different way when using `furrr` to ensure that each of the "workers" has access to the appropriate random draws. Second, before we begin we need to use a function called `plan()` to specify how many workers (cores) we want to use. Third, rather than `pmap()` we use an essentially identical function called `future_pmap()`

```
library(furrr)
```

```
## Loading required package: future
```

```
plan(multisession, workers = 2)

my_options <- furrr_options(seed = 4321)
system.time(
  results_parallel <- many_params %>%
    mutate(sims = future_pmap(., get_estimates, .options = my_options))
)
```

```
##    user  system elapsed
##   0.218   0.005   4.271
```

```
# Return to serial processing
plan(sequential)
```

The results from `pmap()` and `future_pmap()` are not numerically identical despite our having used the same seed because of the different ways in which they use the machine's underlying random number generator:

```
head(all.equal(results_serial, results_parallel))
```

```
## [1] "Component \"sims\": Component 1: Mean relative difference: 1.01108"
## [2] "Component \"sims\": Component 2: Mean relative difference: 1.009128"
## [3] "Component \"sims\": Component 3: Mean relative difference: 0.9901693"
## [4] "Component \"sims\": Component 4: Mean relative difference: 1.010396"
## [5] "Component \"sims\": Component 5: Mean relative difference: 1.000142"
## [6] "Component \"sims\": Component 6: Mean relative difference: 1.002225"
```

Nevertheless, the parallel version is reproducible so we'll get the same results if we re-run it with the same seed. We're also getting the right answer to our problem:

```
results_parallel %>%
  mutate(sim_mean = map_dbl(sims, ~ mean(.x)),
         sim_var = map_dbl(sims, ~ var(.x)),
         sim_bias = sim_mean - s_sq,
         theoretical_bias = -s_sq / n)
```

```
## # A tibble: 168 x 7
##        n  s_sq sims          sim_mean sim_var sim_bias theoretical_bias
##    <int> <dbl> <list>           <dbl>   <dbl>    <dbl>            <dbl>
##  1     3   1   <dbl [5,000]>    0.664   0.439   -0.336           -0.333
##  2     3   1.1 <dbl [5,000]>    0.735   0.544   -0.365           -0.367
##  3     3   1.2 <dbl [5,000]>    0.785   0.607   -0.415           -0.4
##  4     3   1.3 <dbl [5,000]>    0.874   0.728   -0.426           -0.433
##  5     3   1.4 <dbl [5,000]>    0.948   0.886   -0.452           -0.467
##  6     3   1.5 <dbl [5,000]>    1.01    1.00    -0.491           -0.5
##  7     3   1.6 <dbl [5,000]>    1.10    1.22    -0.497           -0.533
##  8     3   1.7 <dbl [5,000]>    1.11    1.23    -0.586           -0.567
##  9     3   1.8 <dbl [5,000]>    1.24    1.50    -0.561           -0.6
## 10     3   1.9 <dbl [5,000]>    1.26    1.58    -0.636           -0.633
## # ... with 158 more rows
```

Now that you understand the basics of carrying out a tidy simulation study, it's your turn to give it a go!

Exercises

1. Read the help file for the function `rmvnorm()` from the package `mvtnorm`. Once you understand how it works, use it to write a function that generates `n` draws from a bivariate standard normal distribution with correlation coefficient `r`. Check you work by generating a large number of simulations and calculating the sample variance-covariance matrix using the base R function `var()`.

2. The function `cov()` calculates the sample covariance between X and Y as $S_{xy} =$

$\frac{1}{n-1}\sum_{i=1}^{n}(X_i - \bar{X})(Y_i - \bar{Y})$. In contrast, the maximum likelihood estimator $\widehat{\sigma}_{xy}$ for jointly normal observations (X_i, Y_i) divides by n rather than $(n-1)$. Write a function that takes a matrix with two columns and `n` rows as its input and calculates $\widehat{\sigma}_{xy}$.

3. Use the functions you wrote in the preceding two parts to carry out a simulation study investigating the bias of $\widehat{\sigma}_{xy}$. Use 5000 replications and a parameter grid of `n` $\in \{5, 10, 15, 20, 25\}$, `r` $\in \{-0.5, 0.25, 0, 0.25, 0.5\}$. Summarize your findings.

Solutions

```
library(tidyverse)
library(furrr)
library(mvtnorm)

draw_sim_data <- function(n, r) {
  var_mat <- matrix(c(1, r,
                      r, 1), 2, 2, byrow = TRUE)
  rmvnorm(n, sigma = var_mat)
}

get_estimate <- function(dat) {
  stopifnot(ncol(dat) == 2)
  x <- dat[,1]
  y <- dat[,2]
  mean((x - mean(x)) * (y - mean(y)))
}

get_estimates <- function(n, r, nreps = 5000) {
  rerun(nreps, draw_sim_data(n, r)) %>%
    map_dbl(get_estimate)
}

sim_params <- expand_grid(n = c(5, 10, 15, 20, 25),
                          r = c(-0.5, -0.25, 0, 0.25, 0.5))

plan(multisession, workers = 4)

my_options <- furrr_options(seed = 4321)

sim_results <- sim_params %>%
  mutate(sims = future_pmap(., get_estimates, .options = my_options))

sim_bias <- sim_results %>%
  mutate(sim_mean = map_dbl(sims, ~ mean(.x)),
```

```r
         bias = sim_mean - r)

sim_bias %>%
  select(n, r, bias) %>%
  ggplot(aes(x = n, y = bias)) +
  geom_point() +
  geom_line() +
  facet_wrap(~ r)
```