Applied Class #3 - Running a Simulation Study

Introduction

In our last applied class we learned about Monte Carlo simulation basics, including the following "general recipe"

1. Write a function to carry out the experiment *once*.
2. Use iteration to call that function a large number of times.
3. Store and summarize results; set seed for replicability.

In today's class we'll look at more elaborate examples that call for a more complicated recipe and more powerful tools.

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*.[1] In particular, if $X_1, \dots, X_n$ 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.[2] 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.

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. We'll go through all the steps of this simulation study below.

Functional Programming

Simulation studies involve a lot of tedious book-keeping. To cut down on the tedium and lower the change of making a silly mistake, we'll use the purrr package for *functional programming*, part of the tidyverse family of R packages. To learn more about `purrr`, I suggest consulting the purrr cheatsheet and Chapter 21 of *R for Data Science*. In short, a **functional** is a function that either takes another function as an input or returns another function as its output. If this sounds abstract, don't worry: it will make sense after you've seen some examples.

`purrr::map()`

The simplest example of a functional is `map(x, f)` from the `purrr` package. This functional works as follows:

$$\texttt{map}(\mathbf{x}, f) = \texttt{map}\left(\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, f\right) = \begin{bmatrix} f(x_1) \\ f(x_2) \\ \vdots \\ f(x_n) \end{bmatrix}$$

where $\mathbf{x}$ is a list or vector and f is a function. In other words `purrr:map(x, f)` is *exactly equivalent* to this `for()` loop in terms of the output it creates:

```
results <- vector('list', length(x)) # pre-allocate empty list
for (i in 1:length(x)) {
  results[[i]] <- f(x[[i]]) # x is a list; items could be *anything*
}
```

Perhaps this all sounds a bit abstract. To make it more concrete, here's a simple example. The function `sum_prod()` returns a list containing the sum and product of any numeric vector `v` that is passed to it as an argument:

```
sum_prod <- function(v) {
  # Return the sum and product of a vector v as a named vector
  c('sum' = sum(v), 'product' = prod(v))
}
```

But suppose we have a *list* of vectors `x` and want to apply `sum_prod()` to each element of the list. The `purrr` function `map()` makes this easy:

```
x <- list(c(1, 1, 1), c(1, 2, 3), c(3, 3, 3))

library(tidyverse) # loads purrr
map(x, sum_prod)
```

```
[[1]]
    sum product
      3       1

[[2]]
```

```
    sum product
      6       6

[[3]]
    sum product
      9      27
```

Nice! But notice how the result of `map()` is a *list*. Sometimes this is what we want, but in other cases it's a bit cumbersome. In the present example, it would make more sense to return a *dataframe* in which each row is one of the list elements from `map(x, sum_prod)`. To do this, we simply replace `map()` with `map_dfr()`, where `dfr` means "bind the results together into a data frame by **r**ow." For example:

```
map_dfr(x, sum_prod)
```

```
# A tibble: 3 × 2
    sum product
  <dbl>   <dbl>
1     3       1
2     6       6
3     9      27
```

purrr::pmap()

There's one more command from `purrr` that we'll need below: `pmap()`, which stands for "parallel map." This command `pmap(l, f)` works as follows

$$\texttt{pmap}\left(\begin{bmatrix} l_{11} & l_{12} & \cdots & l_{1r} \\ l_{21} & l_{22} & \cdots & l_{2r} \\ \vdots & & & \\ l_{n1} & l_{n2} & \cdots & l_{nr} \end{bmatrix}, f\right) = \begin{bmatrix} f(l_{11}, l_{12}, \ldots, l_{1r}) \\ f(l_{21}, l_{22}, \ldots, l_{2r}) \\ \vdots \\ f(l_{n1}, l_{n2}, \ldots, l_{nr}) \end{bmatrix}$$

where `l` is a dataframe with `r` columns and `f` is a function that takes r arguments. Like `map()`, `pmap()` returns a list by default. If we want to bind the results into a dataframe by row, `pmap_dfr()` is analogous to `map_dfr()`. Here's a simple example. Notice that the argument names of `my_function()` have to *match* those of `my_dataframe`:

```
my_dataframe <- tibble(x = 1:5, y = 5:1, z = -2:2)
my_dataframe
```

```
# A tibble: 5 × 3
      x     y     z
  <int> <int> <int>
1     1     5    -2
2     2     4    -1
3     3     3     0
4     4     2     1
5     5     1     2
```

```
my_function <- function(x, y, z) {
  z / (x + y)^2
}
pmap(my_dataframe, my_function)
```

```
[[1]]
[1] -0.05555556

[[2]]
[1] -0.02777778

[[3]]
[1] 0

[[4]]
[1] 0.02777778

[[5]]
[1] 0.05555556
```

The Whole Kit & Caboodle

Now we're ready to carry out our simulation study. As described above, we'll approximate the bias of the "usual" estimator of the population variance and compare it to that of the maximum likelihood estimator, which divides by n rather than $(n-1)$. To carry out our simulation study, we'll carry out this slightly expanded recipe:

1. Write a function to generate simulation data.
2. Write a function to calculate your estimates.
3. Run the simulation for fixed parameters. Repeat many times:
   i. Draw simulation data.
   ii. Calculate estimates.
4. Repeat step 4 over a grid range of parameter values.
5. Store and summarize the results.

Step 1: Function to Generate Sim Data

First we write a function to generate one simulation dataset. For simplicity, we'll take the population mean to be zero in our simulation: $\mu = 0$. this doesn't make a difference to the results, but makes life simpler. We'll allow the sample size `n` and the true population variance `s_sq` to vary:

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

Step 2: Function to Calculate Estimates

Next we write a function to calculate our estimators. The base R function `var()` is the "usual" sample variance. In contrast the MLE `mean((x - mean(x))^2)` divides by `n` rather than `(n-1)`.

```
get_estimates <- function(x) {
  c('usual' = var(x),
```

```
  'MLE' = mean((x - mean(x))^2)) # divide by n; not (n - 1)
}
```

Step 3: Run Simulation for Fixed Parameters

Now we're ready to test out `draw_sim_data()` and `get_estimates()` by running our simulation study for *fixed* parameter values: $n = 5$ and $\sigma^2 = 1$. The first step is a tiny bit tricky. After setting the seed, we'll call `map()` with an *anonymous function* of a "dummy argument" `i`. This argument doesn't do anything, but it's a way to "trick" `map()` into working like the base R function `replicate()` that you met last time. To give you the idea, here's an example in which we use `map()` to call `draw_sim_data(n = 5, s_sq = 1)` a total of `3` times and store the resulting simulation datasets in a list:

```
# Generate list of three simulation datasets as an example
set.seed(1693)
sim_datasets <- map(1:3, function(i) draw_sim_data(n = 5, s_sq = 1))
sim_datasets
```

```
[[1]]
[1] -0.5070343 -0.8750463 -0.1883818 -1.1772701  1.8960400

[[2]]
[1]  0.3824529  1.0291447 -1.3998332  1.3983002 -0.3950949

[[3]]
[1]  0.1204153 -0.9611377 -0.3870477  0.7757843  0.6520975
```

Perfect! Now we'll use `map_dfr()` to calculate the "usual" estimator and the MLE for σ^2 in each of the three simulation sets, binding the results together into a dataframe for convenience:

```
# Compute the estimates for all three datasets; bind rows
map_dfr(sim_datasets, get_estimates)
```

```
# A tibble: 3 × 2
  usual   MLE
  <dbl> <dbl>
1 1.47  1.18
2 1.27  1.01
3 0.527 0.421
```

So now we have run a little simulation study with $n = 5, \sigma^2 = 1$ and three replications. Three replications is far to few to get a reliable approximation to the bias of an estimator, so now we'll follow exactly the same procedure but with `5000` simulation replications

```
set.seed(1693)
nreps <- 5000
sim_datasets <- map(1:nreps, function(i) draw_sim_data(n = 5, s_sq = 1))
sim_estimates <- map_dfr(sim_datasets, get_estimates)
sim_estimates
```

```
# A tibble: 5,000 × 2
   usual   MLE
   <dbl> <dbl>
 1 1.47  1.18
 2 1.27  1.01
 3 0.527 0.421
 4 0.858 0.686
 5 0.863 0.690
 6 0.942 0.754
 7 0.521 0.417
 8 2.84  2.27
 9 0.941 0.753
10 0.775 0.620
# ℹ 4,990 more rows
```

Using `sim_estimates` we can approximate the bias of $\widehat{\sigma}^2$ as follows, and compare it to the theoretical bias described above:

```
# Sim parameters: n = 5,  s_sq = 1,
n <- 5
s_sq <- 1
sim_estimates |>
  summarize(bias_true = -s_sq / n, bias_sim = mean(MLE - s_sq))
```

```
# A tibble: 1 × 2
  bias_true bias_sim
      <dbl>    <dbl>
1      -0.2   -0.197
```

Great: our simulation appears to work! In our last class, this would have been the end of the story. But today we have a bit more work to do. Next we'll allow n and σ^2 to *vary* and see how the results change.

Step 4 - Run Sim over Parameter Grid

To make the book-keeping simpler, it's helpful to "wrap up" the code from Step 3 into a function. We'll call this function `run_sim()`. Its purpose is to run the simulation for fixed parameter values and return the "usual" estimates and MLE for each simulation replication. Its arguments are `n` and `s_sq` so we can easily change the parameter values later on:

```
run_sim <- function(n, s_sq) {
  nreps < - 5000
  sim_datasets <- map(1:nreps, function(i) draw_sim_data(n, s_sq))
  sim_estimates <- map_dfr(sim_datasets, get_estimates)
  return(sim_estimates)
}
```

Now if we set the seed to `1693` and call `run_sim(n = 5, s_sq = 1)` we will get exactly the same results as above:

```
set.seed(1693)
run_sim(n = 5, s_sq = 1)
```

```
# A tibble: 5,000 × 2
   usual   MLE
   <dbl> <dbl>
```

```
 1 1.47  1.18
 2 1.27  1.01
 3 0.527 0.421
 4 0.858 0.686
 5 0.863 0.690
 6 0.942 0.754
 7 0.521 0.417
 8 2.84  2.27
 9 0.941 0.753
10 0.775 0.620
# ℹ 4,990 more rows
```

Our next step is to set up a *grid* of parameter values for `n` and `s_sq`. The function `expand_grid()` from `tidyr`, part of the `tidyverse` family of R packages, makes this easy:

```
sim_params <- expand_grid(n = 3:5, s_sq = 1:3)
sim_params
```

```
# A tibble: 9 × 2
      n  s_sq
  <int> <int>
1     3     1
2     3     2
3     3     3
4     4     1
5     4     2
6     4     3
7     5     1
8     5     2
9     5     3
```

Now we're ready to run the whole simulation study! We have a function `run_sim()` that takes two arguments: `n` and `s_sq`. We also have a dataframe `sim_params` with columns named `n` and `s_sq` that correspond to the different combinations of parameter values we'd like to consider. Using `pmap()` we can now run the simulation at the parameter values corresponding to each row of `sim_params`, as follows. Notice how the result is a *list of dataframes*. Each of the dataframes contains 5000 simulation replications of the "usual" estimator and MLE at a particular combination of `n` and `s_sq` values:

```
sim_results <- pmap(sim_params, run_sim)
sim_results[1:2] # just print out the first two list elements
```

```
[[1]]
# A tibble: 5,000 × 2
    usual    MLE
    <dbl>  <dbl>
 1 0.370  0.247
 2 0.373  0.249
 3 0.135  0.0903
 4 2.95   1.97
 5 0.0589 0.0393
 6 0.975  0.650
 7 1.63   1.09
 8 1.30   0.865
 9 1.53   1.02
10 0.823  0.548
# ℹ 4,990 more rows

[[2]]
# A tibble: 5,000 × 2
   usual    MLE
   <dbl>  <dbl>
 1 0.551 0.368
 2 0.513 0.342
 3 0.478 0.319
 4 1.54  1.03
 5 0.410 0.274
 6 2.15  1.43
 7 0.447 0.298
 8 0.145 0.0969
 9 0.455 0.303
10 0.466 0.311
# ℹ 4,990 more rows
```

Step 5: Summarize Results

We're basically done: now we have all the information that we need, so it's merely a matter of summarizing it. To do this, we'll create a function called `get_summary_stats()` that takes as its inputs one of the dataframes from `sim_results` and returns a vector of summary statistics. Then we'll use `map_dfr()` to run `get_summary_stats()` on each of the dataframes from `sim_results`:

```
get_summary_stats <- function(df) {
  c('usual_mean' = mean(df$usual),
    'MLE_mean' = mean(df$MLE),
    'usual_var' = var(df$usual),
    'MLE_var' = var(df$MLE))
}
summary_stats <- map_dfr(sim_results, get_summary_stats)
summary_stats
```

```
# A tibble: 9 × 4
  usual_mean MLE_mean usual_var MLE_var
       <dbl>    <dbl>     <dbl>   <dbl>
1      0.990    0.660     0.966   0.429
2      1.96     1.31      3.82    1.70
3      2.93     1.95      8.58    3.81
4      0.995    0.746     0.688   0.387
5      2.00     1.50      2.60    1.46
6      2.95     2.21      5.80    3.26
7      0.997    0.797     0.484   0.310
8      2.02     1.62      2.09    1.34
9      3.03     2.42      4.71    3.01
```

Next we'll use the function [bind_cols()]() from `dplyr` to append the values from `sim_params` to our dataframe `summary_stats`:

```r
summary_stats <- bind_cols(sim_params, summary_stats)
summary_stats
```

```
# A tibble: 9 × 6
      n  s_sq usual_mean MLE_mean usual_var MLE_var
  <int> <int>      <dbl>    <dbl>     <dbl>   <dbl>
1     3     1      0.990    0.660     0.966   0.429
2     3     2      1.96     1.31      3.82    1.70 
3     3     3      2.93     1.95      8.58    3.81 
4     4     1      0.995    0.746     0.688   0.387
5     4     2      2.00     1.50      2.60    1.46 
6     4     3      2.95     2.21      5.80    3.26 
7     5     1      0.997    0.797     0.484   0.310
8     5     2      2.02     1.62      2.09    1.34 
9     5     3      3.03     2.42      4.71    3.01 
```

Finally, we can compare the theoretical bias of the MLE to the approximate bias we calcualted from our simulation study, along with the RMSE of each estimator:

```r
summary_stats |>
  mutate(MLE_bias = -s_sq / n,
         MLE_sim_bias = MLE_mean - s_sq,
         usual_sim_bias = usual_mean - s_sq,
         MLE_rmse = sqrt(MLE_sim_bias^2 + MLE_var),
         usual_rmse = sqrt(usual_sim_bias^2 + usual_var)) |>
  select(MLE_bias, MLE_sim_bias, MLE_rmse, usual_rmse)
```

```
# A tibble: 9 × 4
  MLE_bias MLE_sim_bias MLE_rmse usual_rmse
     <dbl>        <dbl>    <dbl>      <dbl>
1   -0.333       -0.340    0.738      0.983
2   -0.667       -0.691    1.48       1.96 
3   -1           -1.05     2.22       2.93 
4   -0.25        -0.254    0.672      0.829
5   -0.5         -0.498    1.31       1.61 
6   -0.75        -0.789    1.97       2.41 
7   -0.2         -0.203    0.592      0.696
8   -0.4         -0.383    1.22       1.45 
9   -0.6         -0.578    1.83       2.17 
```

Overall, our simulation study did a good job of approximating the true bias of the MLE. In spite of its bias, however, notice that the MLE has a *lower RMSE* than the usual estimator. That's because it has a lower variance to compensate for its higher bias.

Exercise

Now it's your turn! Using the example from above as a template, you'll carry out your own simulation study to approximate the bias of two alternative estimators of the population covariance between X and Y.

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 $\widehat{\sigma}_{xy}$ along with the "usual" estimator.

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

Footnotes

1. Just so we're clear, it's *not* a good idea to pursue unbiasedness at all costs! There is always and everywhere a **bias-variance tradeoff**. The point here is that dividing by $(n-1)$ ensures that the estimator is unbiased, *not* that unbiased estimators are necessarily what we should use in practice.↩
2. 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.↩