Recitation 2: Bias, and For Loop vs. Apply

Seamus Wagner

September 1, 2021

For Loops Can Be Slow (in R)

For loops are (kind of) intuitive and extremely useful. However, in R, they can be slow. This is because R is an *interpreted* language such that the things you do inside a loop get copied with each iteration of the loop. This doesn't happen with *vectorized* operations, like loops in other languages such as C++, that run faster. This can make loops in R take up memory space that other operations don't require.

Below, we're going to walk through some examples of how to re-write loops using functions like sapply() and lapply(). However, as our friends at Stackoverflow remind us, this doesn't mean you have to re-write all of your loops in apply syntax just for the sake of doing it. You can still write relatively fast for loops if you:

- Initialize your objects to their full length outside of the loop. Increasing their size within the loop will eat memory
- Don't do anything in a loop that can be done before the loop.

That said, the apply family of functions – my favorites are sapply() and lapply() – can be useful substitutes and complements. And once you understand the principles behind them, it'll be easy to replace most of your for loops. The apply family of functions can also be used to write in fewer lines what it would take more to write in a normal loop, which is a nice feature when writing longer scripts.

Packages and Options

```
## Load packages
library(data.table)
library(tidyverse)
rm(list=ls())
## Set seed number
set.seed(428754)

options(digits = 3)
```

Part 1

We're going to start by writing a function to generate some fake data. With 100 observations, we'll generate two x variables – each with a mean of 3 and a standard deviation of 5. We'll then generate an outcome variable, y, that is a function of our x variables plus some random error.

```r
# function takes sample size (N), independent variable mean and sd (xmeans, xsds),
  # error mean and sd (emean, esd), and coefficients (betas)
# the values specified here are defaults, so it'll run with a simple makedata() call
  # but you might want to edit them in specific calls
n <- 100
xmean <- 3
xsd <- 5
errmean <- .25
errsd <- .2
betas = c(2, -1, 3)
#Function for generating data (note: could use canned functions for most of this)
#We can also define n, xmean, etc within this line of the function, ie. function(n = 100, xmean = 3, et

generate_data <- function(n, xmean, xsd, errmean, errsd, betas){
  # generate x variables
  x1 <- rnorm(n, xmean, xsd)
  x2 <- rnorm(n, xmean, xsd)

  # generate error term (noise)
  err <- rnorm(n, errmean, errsd)

  # combine x variables, with a vector of 1s for the intercept
  x <- cbind(1, x1, x2)

  # make outcome
  y <- x %*% betas + err

  # combine and return data
  my_data <- data.frame(cbind(x1, x2, y))
  names(my_data) <- c("x1","x2","y")
  return(my_data)
}
my_data <- generate_data(n, xmean, xsd, errmean, errsd, betas)

# make sure you can recover the betas in linear regression coefficients
mod1 <- lm(y ~ x1 + x2, data = my_data)
summary(mod1)
```

```
##
## Call:
## lm(formula = y ~ x1 + x2, data = my_data)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -0.5204 -0.1274  0.0229  0.1285  0.5289
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  2.29685    0.02919    78.7   <2e-16 ***
## x1          -1.00436    0.00531  -189.1   <2e-16 ***
## x2           2.99949    0.00492   609.8   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
##
## Residual standard error: 0.216 on 97 degrees of freedom
## Multiple R-squared:     1,   Adjusted R-squared:     1
## F-statistic: 2.14e+05 on 2 and 97 DF,  p-value: <2e-16
```

Now say we want to repeat this process 1000 times, to see how sensitive our result is to randomness along the way. There are a few different ways to accomplish this. We can make an empty list and fill it with a loop, or we can use lapply to return a list directly.

We're going to start with the loop version:

```
mods_loop <- list(NULL)

# wrapping our operation in system.time() will perform the operation and print how long it took
system.time(
  for(i in 1:1000){
    dat <- generate_data(n, xmean, xsd, errmean, errsd, betas)
    mods_loop[[i]] <- lm(y ~ x1 + x2, data = dat)
  }
)
```

```
##    user  system elapsed
##    0.53    0.00    0.53
```

There are two things to notice here. First, notice that we needed to initialize our list before running our loop – and then fill that list with elements as we go. Second, notice that in addition to your list, you now have objects in your environment, i and dat, that are essentially useless (they're just left over from the last round of the loop). This is because for loops operate in your global environment. This normally isn't a huge issue, since objects like these get overwritten with each pass through the loop. However, if these objects are big (say, a large model object) they can take up a lot of space that you might want further down in your script.

If we re-write this routine using lapply, it will return a list directly and we will not add any extra objects to our environment.

```
# remove those extra objects
rm(dat, i)

system.time(
  mods_lapply <- lapply(1:1000, function(x){
    dat <- generate_data(n, xmean, xsd, errmean, errsd, betas)

    # you don't need to be specific about what to return,
      # but for more complicated stuff it's good to be deliberate about it
      # so I'll store this as an object next time and return that object
    lm(y ~ x1 + x2, data = dat)
  })
)
```

```
##    user  system elapsed
##    0.54    0.00    0.53
```

Notice that these two runs took about the same amount of time to complete. It used to be the case that for loops were always, *always* slower than the apply family of functions or other variants because of computer

science technicalities regarding how R talks to your computer. More recent versions of R have done a lot to improve for loop performance on the time side, such that *most* of the time for smaller-scale operations you won't see major differences in speed.

However, we can extend the lapply version in ways that are more difficult to do in a regular loop. For instance, say we care about the each element of the object returned by system.time(), not just the overall elapsed time, and want to make a data.frame where each row is an iteration through the loop and each column is a different element. In lapply, this is easy, since we can have each round of lapply return a vector and stack those vectors on top of each other using tidyverse's bind_rows() (or do.call(rbind()), but that's less intuitive).

```
# make vector of run lengths
iters <- c(100, 1000, 5000)

apply_time_stack <- bind_rows(
  lapply(iters, function(n){
    s <- system.time(
    mods_apply <- lapply(1:n, function(x){
                      mod <- lm(y ~ x1 + x2, data = generate_data(n, xmean, xsd, errmean, errsd, betas))

                      return(mod)
                      })
                    )
  # return a data.frame with names so bind_rows() knows how to stack them
  return(data.frame(iterations = n,
                    user = s[1],
                    system = s[2],
                    elapsed = s[3]))
}))
```

This is harder to do in a loop, since you have to initialize something before the loop begins. That said, it's not impossible!

```
# you just have to initialize the first row of the df you want separately,
# and THEN stack the rest in a loop, which is annoying and takes longer to write

mods_loop <- list(NULL)
s1 <- system.time(
  for(i in 1:iters[1]){
      mods_loop[[i]] <- lm(y ~ x1 + x2, data = generate_data(n, xmean, xsd, errmean, errsd, betas))
  }
)
loop_time_stack <- data.frame(iterations = iters[1],
                              user = s1[1],
                              system = s1[2],
                              elapsed = s1[3])

for(i in 2:length(iters)){
  s <- system.time(
    # it's very easy to accidentally overwrite your i value in loops
    # make sure you don't do that
  for(ii in 1:iters[i]){
      mods_loop[[ii]] <- lm(y ~ x1 + x2, data = generate_data(n, xmean, xsd, errmean, errsd, betas))
  }
```

```
)
  df <- data.frame(iterations = iters[i],
                               user = s[1],
                               system = s[2],
                               elapsed = s[3])
  loop_time_stack <- data.frame(rbind(loop_time_stack,df))
}
```

In Sum

For loops will not slow you down too much if you don't ask too much of them. For more complex operations, the apply family of functions offers useful alternatives, but you don't need to rewrite all of your loops in corresponding apply format just for the sake of doing it.

Final thoughts

What is sample error? - sample error = estimate - population ATE

Office hours by appt.

Tell me if there are things you want covered