Writing better, faster code in R

Julia Wrobel

Overview

Today, we cover:

- Resampling methods
  - Permutation tests
- Improving code speed:
  - Benchmarking
  - Vectorization
  - Parallelization

Announcements:

- HW2 posted and due 2/11 at 10:00AM
- No class tomorrow (Thursday, January 29) but still have Office Hours

Permutation tests

Typically bootstrap is used for CI rather than hypothesis testing. For hypothesis testing and p-values, we can use a **permutation test**. - Idea: use resampling to generate a **null distribution** for a test statistic, then compare it to the one you observe in the real data

- **Null distribution**: the distribution of a quantity of interest (i.e. $\hat{\beta}$) if the null hypothesis H_0 is true
- The null distribution is available theoretically in some cases. For example, assume $Y_i \sim N(\mu, \sigma^2), i = 1, \dots, n$.
  - Under $H_0 : \mu = 0$, we have $\bar{Y} \sim N(0, \sigma^2/n)$
    - Test H_0 by comparing $\bar{Y}$ with $N(0, \sigma^2/n)$
- Use **permutation test** when null distribution cannot be obtained theoretically

Permutation tests

The basic procedure of permutation test for H_0:

- Permute data under H_0 B times. Each time recompute the test statistics. The test statistics obtained from the permuted data form the null distribution.
- Compare the observed test statistics with the null distribution to obtain statistical significance.

Permutation test example

Assume there are two sets of independent normal r.v.'s with the same known variance and different means:

- $X_i \sim N(\mu_1, \sigma^2)$
- $Y_i \sim N(\mu^2, \sigma^2)$

Our goal is to test $H_0 : \mu_1 = \mu_2$. Define test statistic: $t = \bar{X} - \bar{Y}$. Permutation test steps:

1. Randomly shuffle labels of X and Y
2. Compute $t^* = \bar{X}^* - \bar{Y}^*$
3. Repeat `nperm` times. Resulting t^* values form the **empirical null distribution** of t.
4. To compute p-values calculate $Pr(|t^*| > |t|)$

Improving speed

Two ways to find bottlenecks in code:

- Benchmarking
- Profiling

Comparing R codes for speed

- R package `microbenchmark` is well-suited for comparing small chunks of code
- Your code can often be significantly improved

```
library(microbenchmark)
x <- 120
microbenchmark(
  sqrt(x),
  x^(0.5)
)
```

```
Unit: nanoseconds
    expr min lq  mean median uq  max neval
 sqrt(x)   0  0 29.11      0  0 2665   100
 x^(0.5)  41 41 81.59     41 82 2009   100
```

Comparing R codes for speed

```r
p <- 1000
x <- runif(p, min = 100, max = 120)
microbenchmark(
  sqrt(x),
  x^(0.5)
)
```

```
Unit: nanoseconds
    expr  min   lq     mean median   uq   max neval
 sqrt(x)  820  861  1619.09 1086.5 1927  9143   100
 x^(0.5) 8364 8446 10253.69 9143.0 9717 55309   100
```

Comparing R codes for speed

```r
p <- 100000
x <- runif(p, min = 100, max = 120)
microbenchmark(
  sqrt(x),
  x^(0.5)
)
```

```
Unit: microseconds
    expr      min       lq     mean   median       uq      max neval
 sqrt(x)  85.772 133.7215 211.1685 142.6800 157.6655 2560.573   100
 x^(0.5) 841.853 900.7290 933.2395 928.3015 966.2060 1055.668   100
```

Comparing R codes for speed

- Take advantage of **crossprod** and **tcrossprod** functions. Suppose we want to calculate $x^T Ax$

```
p <- 3000
x <- rnorm(p)
A <- matrix(rnorm(p^2), p, p)
microbenchmark(
    t(x) %*% A %*% x,
    crossprod(x, A %*% x)
)
```

```
Unit: milliseconds
                  expr      min       lq     mean   median       uq      max
      t(x) %*% A %*% x 2.966145 3.036522 3.151172 3.140641 3.234818 3.543384
 crossprod(x, A %*% x) 6.249138 6.364635 6.490777 6.453072 6.541878 8.006111
```

Comparing R codes for speed

- **Hack** for calculating $\|x\|_2^2$ for large p

```
p <- 100000
x <- rnorm(p)
as.numeric(crossprod(x))
```

```
[1] 100031.9
```

```
sum(x^2)
```

```
[1] 100031.9
```

Comparing R codes for speed

- **Hack** for calculating $\|x\|_2^2$ for large p

```
microbenchmark(
    as.numeric(crossprod(x)),
    sum(x^2)
)
```

```
Unit: microseconds
                      expr   min    lq     mean median    uq
 as.numeric(crossprod(x)) 3.075 3.116 3.26360  3.116 3.157
                   sum(x^2) 3.649 3.813 5.16805  4.100 5.863
```

Example: R Loops

R is very bad at resizing objects since it copies to resize

- Do not grow an object inside of a loop!
- Instead, make an empty object first and then fill elements.

```
bad = function (x){
  obj = c()
  for(i in 1:x){
    obj = c(obj, i)
    }
  return(obj)
}
```

Example: R Loops

```
microbenchmark(bad (100), better (100))
```

```
Unit: microseconds
         expr    min      lq     mean median      uq      m
    bad(100) 15.867 17.1585 30.84020 17.876 19.1265 1170.7
 better(100)  2.501  2.6855 14.92359  2.829  3.2595 1186.3
```

Measuring speed in simulations

If you are interested in measuring computation time in a simulation study, say, to compare how fast different methods are, I **would not** recommend `microbenchmark`. Instead, do the following:

```
library(tictoc)


tic()
## do some stuff
large_vector <- rnorm(1e7)  # Create a vector of 10 millio
sum_large_vector <- sum(large_vector)
```

Vectorization

Why vectorization?

R is **vectorized**, meaning it efficiently performs operations on entire vectors or arrays in a single step, avoiding explicit loops and leveraging optimized low-level code for speed and simplicity. - Often, there is more than one way to do something in R - Take advantage of vectorization! - Often it is more concise and significantly faster

```
# non vectorized squaring operation
x <- c(1, 2, 3, 4, 5)
result <- numeric(length(x))
for (i in seq_along(x)) {
```

Why vectorization?

Another example- which is the vectorized version?

```
x <- matrix(rnorm(30), 10, 3)



colMeans(x)
```

```
[1] -0.1358174 -0.4133417 -0.2395754
```

```
apply(x, 2, mean)
```

```
[1] -0.1358174 -0.4133417 -0.2395754
```

Why vectorization?

- **colSums**, **colMeans** and corresponding row functions are vectorized

```
n <- 100
p <- 3000
A <- matrix(rnorm(n * p), n, p)
microbenchmark(
    colMeans(A),
    apply(A, 2, mean)
)
```

Break

Let's stop and try vectorization Examples 1 and 2 in the lab.

Next up will be profiling and parallelization.

Parallel computing

A modern CPU (Central Processing Unit) is at the heart of every computer. While traditional computers had a single CPU, modern computers can ship with mutliple processors, which in turn can each contain multiple cores. These processors and cores are available to perform computations.

A computer with one processor may still have 4 cores (quad-core), allowing 4 computations to be executed at the same time.

Parallel computing is the simultaneous use of multiple processors or computers to solve a problem by dividing it into smaller, independent tasks, i.e. operating **in parallel**.

Parallel computing

You can check how many **cores** your computer has to see how many tasks can be run at once.

```
# Load the parallel package
library(parallel)

# Get the number of cores
detectCores()
# 12
```

When to parallelize

- Using 2 cores does not mean your code will be $2\times$ faster
- Not all tasks can be parallelized
- Loops and repetitive tasks are great candidates
  - What are some computations we have looked at already that might be good candidates for parallelization?

Example : foreach and doParallel

Example : foreach and doParallel

```
library(doParallel)
library(foreach)

# Set up parallel backend with 10 cores
num_cores = detectCores() - 2
cl = makeCluster(num_cores)
registerDoParallel(cl)

# define  Monte Carlo function to estimate Pi
monte_carlo_pi <- function(n) {
```

Example : foreach and doParallel

- Useful arguments:
  - multiple iterators
  - error catching
  - combine

```
foreach(
  ...,
  .combine,
  .init,
  .final = NULL,
  .inorder = TRUE,
  .multicombine = FALSE
```

Resources

- Advanced R: chapters 22-24
- foreach vignette
- furrr package for tidyverse parallelization

Extra

Profiling your code

Some good references on profiling to learn more (a lot of overlap):

- Advanced R
- profvis R package
- Rstudio guide
- proftools R package

There has been some changes on how profiling works from R v.3 to v.4 which (sometimes) makes profiler output confusing

Profiling in R

- `profvis` is built on R's built-in profiler tool, `Rprof`
- `Rprof` is a statistical profiler that uses sampling
- When you run `profvis`, it stops the R interpreter every 10ms (default interval) and records which function is currently executing, as well as the entire call stack (i.e., which function called that function)
  - The results are not deterministic

Profiling in R

From R programming for Data Science

> *"Rprof() keeps track of the function call stack at regularly sampled intervals and tabulates how much time is spent inside each function. By default, the profiler samples the function call stack every 0.02 seconds. This means that if your code runs very quickly (say, under 0.02 seconds), the profiler is not useful. But if your code runs that fast, you probably don't need the profiler."*

Profiling: return to Ex. 2 (powers of a matrix)

Rprof() function gives a report of (approximately) how much time each function/operation within your code takes. To see the effect of memory allocation, enable tracking of **garbage collection** (GC)

```
Rprof(gc.profiling = TRUE) # start monitoring
invisible(powers1(x, 8)) # suppress function output
Rprof(NULL) # stop monitoring
summaryRprof() # see the report
```

Rprof()

- **by.total** divides the time spend in each function by the total run time
- **by.self** first subtracts out time spent in functions above the current function in the call stack (**more useful typically**)

Garbage collection (GC)

Garbage collection - freeing the memory from objects that are no longer in use (more in Section 2.6 of Advanced R)

If significant time of your program is spend in GC

- you may be doing a lot of dynamic memory allocation (the case of powers1)
- you may be storing a lot of temporary objects
- you may be consistently changing objects of large sizes

Profiling: return to Ex. 3 (powers of a matrix)

```
powers3 <- function(x, dg){
  # allocate memory in advance
  pw <- matrix(x, length(x), dg)
  prod <- x # current product
  for (i in 2:dg){
    prod <- prod * x
    pw[ , i] <- prod # no cbind
  }
  return(pw)
}
```