Lab 9

Maria Oljaca

Problem 1: Vectorization

The following functions can be written to be more efficient without using parallel. Write a faster version of each function and show that (1) the outputs are the same as the slow version, and (2) your version is faster.

1. This function generates an `n x k` dataset with all its entries drawn from a Poisson distribution with mean `lambda`:

```r
fun1 <- function(n = 100, k = 4, lambda = 4) {
  x <- NULL

  for (i in 1:n){
    x <- rbind(x, rpois(k, lambda))
  }

  return(x)
}
```

Writing a faster version of the function:

```r
fun1alt <- function(n = 100, k = 4, lambda = 4) {
  matrix(rpois(n * k, lambda), nrow = n, ncol = k)
}
```

Showing that `fun1alt` generates a matrix with the same dimensions as `fun1` and that the values inside the two matrices follow similar distributions:

```r
# Checking if the dimensions are the same
identical(dim(fun1()), dim(fun1alt()))
```

```
[1] TRUE
```

```r
# Checking if the distributions are similar
summary(fun1())
```

```
       V1              V2              V3               V4        
 Min.   : 0.0   Min.   :0.00   Min.   : 0.00   Min.   : 0.00  
 1st Qu.: 3.0   1st Qu.:3.00   1st Qu.: 2.00   1st Qu.: 3.00  
 Median : 4.0   Median :4.00   Median : 4.00   Median : 4.00  
 Mean   : 4.1   Mean   :3.94   Mean   : 4.08   Mean   : 4.19  
 3rd Qu.: 5.0   3rd Qu.:5.00   3rd Qu.: 5.00   3rd Qu.: 6.00  
 Max.   :10.0   Max.   :9.00   Max.   :10.00   Max.   :11.00  
```

```r
summary(fun1alt())
```

```
       V1              V2               V3              V4    
 Min.   : 0.00   Min.   : 0.00   Min.   :0.00   Min.   : 0  
 1st Qu.: 3.00   1st Qu.: 2.00   1st Qu.:3.00   1st Qu.: 3  
 Median : 4.00   Median : 3.00   Median :4.00   Median : 4  
 Mean   : 4.03   Mean   : 3.71   Mean   :3.91   Mean   : 4  
 3rd Qu.: 5.00   3rd Qu.: 5.00   3rd Qu.:5.00   3rd Qu.: 5  
 Max.   :11.00   Max.   :10.00   Max.   :9.00   Max.   :12  
```

Showing that `fun1alt` is faster:

```r
library(microbenchmark)
# Benchmarking
microbenchmark(
  fun1(),
  fun1alt()
)
```

```
Unit: microseconds
      expr   min     lq    mean median     uq    max neval
    fun1() 327.2 413.75 484.732 437.95 507.70 1076.4   100
 fun1alt()  13.8  14.60  18.397  15.40  18.15   69.8   100
```

2. This function finds the maximum value of each column of a matrix:

```
# Data Generating Process (10 x 10,000 matrix)
set.seed(1234)
x <- matrix(rnorm(1e4), nrow=10)

# Find each column's max value
fun2 <- function(x) {
  apply(x, 2, max)
}
```

Writing a faster version of the function:

```
library(matrixStats)
fun2alt <- function(x) {
  colMaxs(x)
}
```

Showing that both `fun2` and `fun2alt` return the same output for a given input matrix, `x`:

```
result1 <- fun2(x)
result2 <- fun2alt(x)
identical(result1, result2)
```

```
[1] TRUE
```

Showing that `fun2alt` is faster:

```r
microbenchmark(
  fun2(x),
  fun2alt(x)
)
```

```
Unit: microseconds
       expr   min      lq     mean   median      uq    max neval
    fun2(x) 903.1 960.05 1298.933 1070.90 1500.60 3395.1   100
 fun2alt(x)  40.2  43.35   98.913   48.05   71.05 3711.4   100
```

Problem 2: Parallelization

We will now turn our attention to the statistical concept of bootstrapping. Among its many uses, non-parametric bootstrapping allows us to obtain confidence intervals for parameter estimates without relying on parametric assumptions. Don't worry if these concepts are unfamiliar, we only care about the computation methods in this lab, not the statistics.

The main assumption is that we can approximate the results of many repeated experiments by resampling observations from our original dataset, which reflects the population.

1. This function implements a serial version of the bootstrap. Edit this function to parallelize the `lapply` loop, using whichever method you prefer. Rather than specifying the number of cores to use, use the number given by the `ncpus` argument, so that we can test it with different numbers of cores later.

```r
library(doParallel)
```

```
Loading required package: foreach

Loading required package: iterators

Loading required package: parallel
```

```r
library(foreach)

my_boot <- function(dat, stat, R, ncpus = 1L) {
  # Getting the random indices
  n <- nrow(dat)
  idx <- matrix(sample.int(n, n * R, TRUE), nrow = n, ncol = R)

  # Creating a cluster
  cl <- makeCluster(ncpus)
  registerDoParallel(cl)

  # Parallelizing the function
  ans <- foreach(i = 1:R, .combine = rbind) %dopar% {
    stat(dat[idx[, i], , drop = FALSE])
  }

  # Stopping the cluster
  stopCluster(cl)

  return(ans)
}
```

2. Once you have a version of the `my_boot()` function that runs on multiple cores, check that it provides accurate results by comparing it to a parametric model:

```r
# Bootstrap of an OLS
my_stat <- function(d) coef(lm(y ~ x, data = d))

# DATA SIM
set.seed(1)
n <- 500
R <- 1e4

x <- cbind(rnorm(n))
y <- x * 5 + rnorm(n)

# Checking if we get something similar as lm
ans0 <- confint(lm(y ~ x))
ans1 <- my_boot(dat = data.frame(x, y), my_stat, R = R, ncpus = 2L)

# Check if the results are similar
t(apply(ans1, 2, quantile, c(0.025, 0.975)))
```

```
                   2.5%      97.5%
(Intercept) -0.1386903 0.04856752
x            4.8685162 5.04351239
```

```
ans0
```

```
                  2.5 %     97.5 %
(Intercept) -0.1379033 0.04797344
x            4.8650100 5.04883353
```

3. Check whether your version actually goes faster when it's run on multiple cores (since this might take a little while to run, we'll use `system.time` and just run each version once, rather than `microbenchmark`, which would run each version 100 times, by default):

```
system.time(my_boot(dat = data.frame(x, y), my_stat, R = 4000, ncpus = 1L))
```

```
   user  system elapsed 
   0.68    0.06    8.25 
```

```
system.time(my_boot(dat = data.frame(x, y), my_stat, R = 4000, ncpus = 2L))
```

```
   user  system elapsed 
   0.73    0.10    5.04 
```