# Statistical Packages

Problem Set 7

*Mohamed Salem*

*October 20, 2019*

## Problem 2: Bootstrapping

(a) The code is not working as intended due to two issues:

1- The author mis-specified the number of boot times variable in his/her vector of samples and for loop 2- The author saved the resulting sampling as a numeric vector, then fed the result into a linear regression command, which is invalid as the linear regression command will refuse datasets that are not in a data frame format.

(b) We begin by cleaning our code:

```
# We begin by recycling code from an old assignment to import and
# munge the dataset

# The next line specifies the url which the data will be imported
# from
url1 <- "http://www2.isye.gatech.edu/~jeffwu/wuhamadabook/data/Sensory.dat"

# This creates a vector to hold the names available in the header
# of the data
name_placeholder <- c("Item", "first", "second", "third", "fourth",
    "fifth")

# This reads data from the source table after skipping the first
# line which is the header
Sensory_data_1 <- read.table(url1, fill = T, header = T, skip = 1,
    na.string = c("", "null", "NaN"), stringsAsFactors = FALSE, sep = " ")

# This applies our previously stored names to the imported
# dataframe
names(Sensory_data_1) <- name_placeholder

# The following uses dplyr and tidyr to clean the data
Sensory_data_1 <- as_tibble(Sensory_data_1)
Sensory_data_p1 <- Sensory_data_1 %>% filter(is.na(fifth))
Sensory_data_p2 <- Sensory_data_1 %>% filter(!is.na(fifth))
Sensory_data_p1 <- Sensory_data_p1 %>% mutate(adjitem = ceiling(row_number()/2)) %>%
    select(adjitem, Item, first, second, third, fourth)
names(Sensory_data_p1) <- name_placeholder
Sensory_data_1 <- bind_rows(Sensory_data_p2, Sensory_data_p1) %>%
    arrange(Item)

Sensory_data_1 <- bind_rows(Sensory_data_p2, Sensory_data_p1) %>%
```

```
    arrange(Item)

Sensory_data_clean <- Sensory_data_1 %>% gather(Item) %>% mutate(operator = Item) %>%
    mutate(Item = ceiling(row_number()/3)) %>% mutate(Item = Item -
    10 * floor(Item/10)) %>% mutate(Item = Item + (Item == 0) * 10)

# Clearing workspace
rm(Sensory_data_1, Sensory_data_2, Sensory_data_p1, Sensory_data_p2)
```

```
## Warning in rm(Sensory_data_1, Sensory_data_2, Sensory_data_p1,
## Sensory_data_p2): object 'Sensory_data_2' not found
```

(b) Now we run a bootstrap and save the time taken to complete the tasks. Results are displayed as a table below the code:

```
# Bootstrapped parameters with saved system time
p2_sys_t <- system.time({
    coef_bs <- function(formula, data, indices) {
        d <- data[indices, ]  # allows boot to select sample
        fit <- lm(formula, data = d)
        return(coef(fit))
    }
    bs_mat <- boot(Sensory_data_clean, coef_bs, R = 100, sim = "balanced",
        formula = value ~ operator)
    coef_bs_mat <- data.frame(bs_mat$t)
    names(coef_bs_mat) <- names(bs_mat$t0)
})

# summary table with elapsed time and average of bootstrapped
# coefficients
sum_mat <- data.frame(matrix(NA, nrow = 1, ncol = 7))
names(sum_mat) <- c("elapsed time", "beta 0", "beta 1", "beta 2",
    "beta 3", "beta 4")
sum_mat[1, 1] <- p2_sys_t[3]
sum_mat[1, 2] <- mean(coef_bs_mat$`(Intercept)`)
sum_mat[1, 3] <- mean(coef_bs_mat$operatorfirst)
sum_mat[1, 4] <- mean(coef_bs_mat$operatorsecond)
sum_mat[1, 5] <- mean(coef_bs_mat$operatorthird)
sum_mat[1, 6] <- mean(coef_bs_mat$operatorfourth)
tbl_sum <- (kable(sum_mat))
tbl_sum <- kable_styling(tbl_sum, position = "center")
```

| elapsed time | beta 0 | beta 1 | beta 2 | beta 3 | beta 4 | NA |
|---|---|---|---|---|---|---|
| 0.144 | 4.272092 | 0.3212295 | 0.7964931 | -0.1035814 | 0.9110717 | NA |

(c) Next we use parallel computing to accomplish the same task. This task is amenable to parallel computing because it can be broken up into a number of smaller tasks which are each then handled by one of the cores

```r
# Bootstrapped parameters with saved system time in parallel
# computing We begin by constructing our own bootstrap
# function which can be combined with one of the 'apply'
# family of commands
p2_sys_t_par <- system.time({
    cl <- makeCluster(8)
    clusterExport(cl, c("Sensory_data_clean"))
    op_1 <- subset(Sensory_data_clean, operator == "first")
    op_2 <- subset(Sensory_data_clean, operator == "second")
    op_3 <- subset(Sensory_data_clean, operator == "third")
    op_4 <- subset(Sensory_data_clean, operator == "fourth")
    op_5 <- subset(Sensory_data_clean, operator == "fifth")
    mat_list <- list(op_1, op_2, op_3, op_4, op_5)
    sampler <- function(df, n = 20, m = 100) {
        bs_mat <- data.frame(matrix(NA, nrow = n, ncol = ncol(df) *
            m))
        for (m in seq(1, m, by = 1)) {
            for (i in seq(1, n, by = 1)) {
                bin <- sample(c(1:n), 1)
                bs_mat[i, ((ncol(df) * m) - (ncol(df) - 1)):(ncol(df) *
                  m)] <- df[bin, ]
            }
        }
        assign(paste(deparse(substitute(df)), "_bs", sep = ""),
            bs_mat)
        return(get(paste(deparse(substitute(df)), "_bs", sep = "")))
    }
    ## This finally produces our bootstrapped sample
    bs_mat_par <- parSapply(cl, mat_list, sampler)
    ## Next we run our linear regression and save the resulting
    ## coefficients
    coeff_mat_bs <- data.frame(matrix(NA, nrow = 100, ncol = 5))
    names(coeff_mat_bs) <- c("beta_0", "beta_1", "beta_2", "beta_3",
        "beta_4")
    for (i in seq(1, 100, by = 1)) {
        dat_mat <- data.frame(rbind(cbind(bs_mat_par[[(i * 3 -
            1)]], bs_mat_par[[i * 3]]), cbind(bs_mat_par[[(300 +
            (i * 3 - 1))]], bs_mat_par[[(300 + (i * 3))]]), cbind(bs_mat_par[[(600 +
            (i * 3 - 1))]], bs_mat_par[[(600 + (i * 3))]]), cbind(bs_mat_par[[(900 +
            (i * 3 - 1))]], bs_mat_par[[(900 + (i * 3))]]), cbind(bs_mat_par[[(1200 +
            (i * 3 - 1))]], bs_mat_par[[(1200 + (i * 3))]])))
        dat_mat$X1 <- as.numeric(levels(dat_mat$X1))[dat_mat$X1]
        lmfit <- lm(as.numeric(X1) ~ as.factor(X2), data = dat_mat)
        coeff_mat_bs[i, 1] <- coef(lmfit)[1]
        coeff_mat_bs[i, 2] <- coef(lmfit)[2]
        coeff_mat_bs[i, 5] <- coef(lmfit)[3]
        coeff_mat_bs[i, 3] <- coef(lmfit)[4]
        coeff_mat_bs[i, 4] <- coef(lmfit)[5]
    }
    stopCluster(cl)
})

# summary table with elapsed time and average of bootstrapped
```

```r
# coefficients
sum_mat <- data.frame(matrix(NA, nrow = 1, ncol = 6))
names(sum_mat) <- c("elapsed time", "beta 0", "beta 1", "beta 2",
    "beta 3", "beta 4")
sum_mat[1, 1] <- p2_sys_t_par[3]
sum_mat[1, 2] <- mean(coeff_mat_bs$beta_0)
sum_mat[1, 3] <- mean(coeff_mat_bs$beta_1)
sum_mat[1, 4] <- mean(coeff_mat_bs$beta_2)
sum_mat[1, 5] <- mean(coeff_mat_bs$beta_3)
sum_mat[1, 6] <- mean(coeff_mat_bs$beta_4)
tbl_sum_par <- (kable(sum_mat))
tbl_sum_par <- kable_styling(tbl_sum_par, position = "center")
```

Parallel computing time results:

| elapsed time | beta 0 | beta 1 | beta 2 | beta 3 | beta 4 |
|---|---|---|---|---|---|
| 2.856 | 3.8121 | 0.41975 | 0.93985 | -0.0884 | 1.04255 |

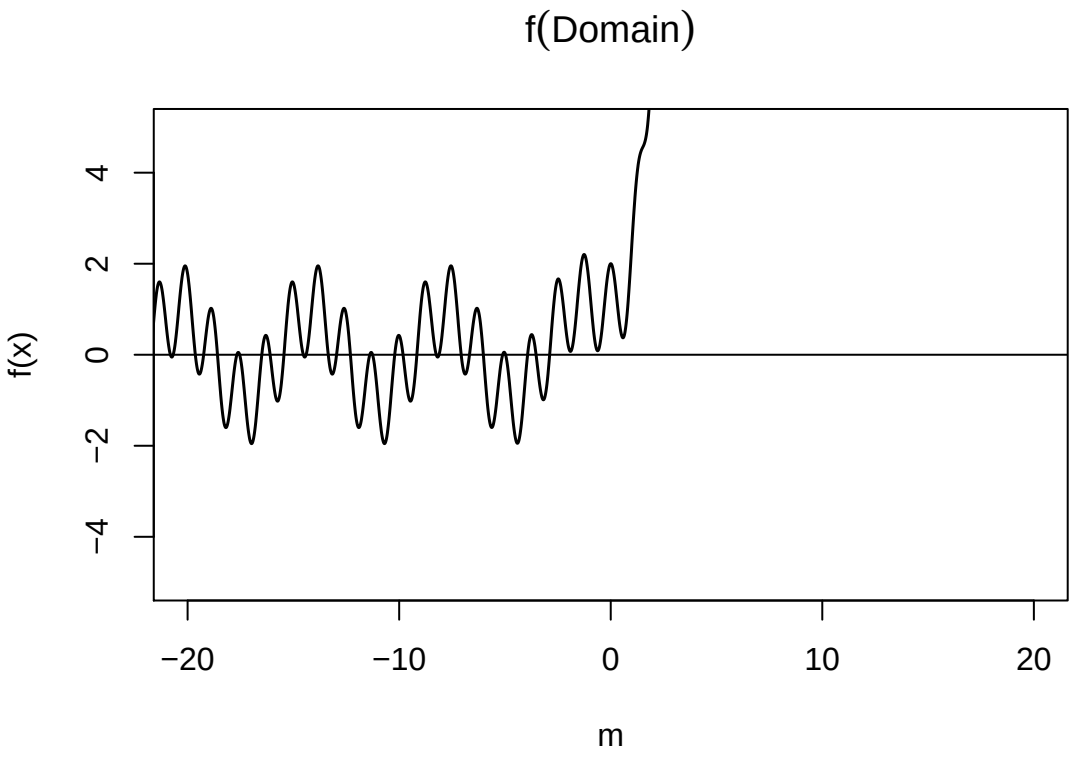
We observe that in this case, parallel computing has led to slower results. This is probably due to two reasons:
1- Our custom-made bootstrap functions had multiple commands to form a function that works nicely with the 'apply' family of functions, this could've slowed down our computation time as opposed to using the prepackaged bootstrap function
2- Our dataset is not large enough that the use of parallel computing would offset the time taken to distribute tasks among the cores and having the cores communicate to collate their results.

## Problem 3: Newton's Method

We begin by using a simplified version of our code from a previous problem set. The result allows us to see that the function seems to have infinitely many roots:

```r
# A function to implement the Newton-Rhapson method of finding
# roots with the arguments f, eps, x0, N, representing our
# function, our tolerance level, our initial guess, and maximum
# number of iterations, respectively. Function adapted from code
# by Yin Zhao, available at:
# https://www.academia.edu/7031789/Newton-Raphson_Method_in_R
# Adapted on 9/24/2019

f <- function(x) {
    3^x - sin(x) + cos(5 * x)
}

NewtonRhapson_ng <- function(xg, f) {
    h = 0.001
    i = 1
    x1 = xg
    x0 = xg
    xstatic = xg
    derv = 1
    while (i <= 10000) {
        derv = (f(x0 + h) - f(x0))/h
        if (is.na(derv) == TRUE) {
```

```
            return(x1)
            break
        }
        x1 = (x0 - (f(x0)/derv))
        i = i + 1
        if (abs(x1 - x0) < 1e-04) {
            break
        } else {
            x0 = x1
        }
    }
    return(x1)
}

xgrid <- matrix((seq(-1, 1, length.out = 1000)), nrow = 1000, ncol = 1)
p3_sys_t <- system.time({
    rt_tbl <- data.frame(apply(xgrid, 1, function(iter) NewtonRhapson_ng(iter,
        f)))
})
rt_tbl_unq <- unique(round(rt_tbl, 5))
sum_mat <- data.frame(matrix(c(p3_sys_t[3], rt_tbl_unq[, 1]), nrow = 1,
    ncol = 1 + length(rt_tbl_unq[, 1])))
names(sum_mat) <- c("system_time", rep("roots", length(rt_tbl_unq[,
    1])))
tbl_sum <- (kable(sum_mat))
tbl_sum <- kable_styling(tbl_sum_par, position = "center")

m <- seq(-50, 50, length = 10000)
FnGrph <- plot(m, f(m), type = "l", lwd = 1.5, main = expression(f(Domain)),
    xlim = c(-20, 20), ylim = c(-5, 5), ylab = "f(x)")
abline(h = 0)
```




| elapsed time | beta 0 | beta 1 | beta 2 | beta 3 | beta 4 |
|---|---|---|---|---|---|
| 2.856 | 3.8121 | 0.41975 | 0.93985 | -0.0884 | 1.04255 |

(b) Next, we carry out the same task using our parallel computing resources:

```
# We're going to run a simplified version of our NR function (with
# no graphs) to produce roots for a certain grid this time using
# parallel computing
p3_sys_t_par <- system.time({
    numCores <- detectCores()
    cl <- makeCluster(8)
    clusterExport(cl, c("NewtonRhapson_ng", "f"))
    rt_tbl <- data.frame(parApply(cl, xgrid, 1, function(iter) NewtonRhapson_ng(iter,
        f)))
    stopCluster(cl)
})

rt_tbl_unq <- unique(round(rt_tbl, 5))
sum_mat <- data.frame(matrix(c(p3_sys_t[3], rt_tbl_unq[, 1]), nrow = 1,
    ncol = 1 + length(rt_tbl_unq[, 1])))
names(sum_mat) <- c("system_time", rep("roots", length(rt_tbl_unq[,
    1])))
tbl_sum <- (kable(sum_mat))
tbl_sum <- kable_styling(tbl_sum, position = "center")
```

| system_time | roots | roots | roots | roots | roots | roots | roots | roots | roots | r |
|---|---|---|---|---|---|---|---|---|---|---|
| 0.148 | -3.52872 | -2.88706 | -4.97151 | -3.93011 | -13.35177 | -22.77655 | -5.10744 | -14.52987 | -6.67606 | -8.1 |

## Problem 4: Gradient Descent

(a) Changing our stopping rule to include our knowledge of the true parameter is doable; yet applying this stopping rule requires that we already have knowledge of the true value of our parameter, which is generally not the case and is in fact the reason we carry our gradient descent. Also, depending on our initial guess, our method may actually never converge to the true parameter; therefore if the true parameter is the only stopping rule, our algorithm will attempt to go on forever while diverging further away from the true parameter. Starting close to the true value, however, lowers the number of iterations required for convergence.

```
# quick gradient descent
set.seed(1256)
theta <- as.matrix(c(1, 2), nrow = 2)
X <- cbind(1, rep(1:10, 10))
h <- X %*% theta + rnorm(100, 0, 0.2)
# need to make guesses for both Theta0 and Theta1, might as well
# be close
alpha <- 0.01  # this is the step size
m <- 100  # this is the size of h
tolerance <- 0.001  # stopping tolerance
grdscnt <- function(k) {
    sum_mat <- matrix(NA, nrow = 1, ncol = 5)
    names(sum_mat) <- c("starting_value_1", "starting_value_2", "stopping_value_1",
        "stopping_value_2", "#_iterations")
    for (j in seq(1, 1000, by = 1)) {
        theta0 <- c(runif(1, min = 0, max = 2), rep(0, 9999))
        theta1 <- c(runif(1, min = 1, max = 3), rep(0, 9999))
        i <- 2
        current_theta <- as.matrix(c(theta0[i - 1], theta1[i - 1]),
            nrow = 2)
        sum_mat[1, 1] <- theta0[1]
        sum_mat[1, 2] <- theta1[1]
        # gradient descent function
        theta0[i] <- theta0[i - 1] - (alpha/m) * sum(X %*% current_theta -
            h)
        theta1[i] <- theta1[i - 1] - (alpha/m) * sum((X %*% current_theta -
            h) * rowSums(X))
        rs_X <- rowSums(X)  # can precalc to save some time
        z <- 0
        while (abs(theta0[i] - theta0[i - 1]) > tolerance && abs(theta1[i] -
            theta1[i - 1]) > tolerance && z <= 5e+06) {
            if (i == 1000)
                {
                  theta0[1] = theta0[i]
                  theta1[1] = theta1[i]
                  i = 1
                }  ##cat('z=',z,'\n',sep='')}
            z <- z + 1
            i <- i + 1
            current_theta <- as.matrix(c(theta0[i - 1], theta1[i -
                1]), nrow = 2)
            theta0[i] <- theta0[i - 1] - (alpha/m) * sum(X %*% current_theta -
                h)
            theta1[i] <- theta1[i - 1] - (alpha/m) * sum((X %*% current_theta -
```

```
                h) * rs_X)
        }
        sum_mat[1, 3] <- theta0[i]
        sum_mat[1, 4] <- theta1[i]
        sum_mat[1, 5] <- z
    }
    return(sum_mat)
}

cl <- makeCluster(8)
clusterExport(cl, c("grdscnt", "X", "h", "theta", "alpha", "m", "tolerance"))
k <- data.frame(matrix(c(1:1000), nrow = 1000, ncol = 1))
names(k) <- c("runs")
sum_mat <- data.frame(t(rbind(parApply(cl, k, 1, grdscnt))))
stopCluster(cl)

tbl_sum <- (kable(head(sum_mat)))
tbl_sum <- kable_styling(tbl_sum, position = "center")
```

| starting_value_1 | starting_value_2 | stopping_value_1 | stopping_value_2 | X._iterations |
|---|---|---|---|---|
| 0.3135274 | 2.108265 | 0.3142193 | 2.103959 | 0 |
| 0.7324746 | 1.745509 | 0.7703818 | 2.026574 | 6 |
| 1.4999585 | 2.301379 | 1.4446064 | 1.932084 | 9 |
| 0.4129353 | 2.607156 | 0.3571324 | 2.107188 | 5 |
| 1.0456527 | 1.381455 | 1.1175723 | 1.970119 | 6 |
| 0.4135502 | 1.611943 | 0.4806645 | 2.072855 | 9 |

I think this algorithm is much less efficient than the least squares or the maximum likelihood method for the purpose of obtaining linear regression coefficients. However, when we start close to our true parameters, our solution converges quickly, with exactly 0% of my starting values yielding convergence after 5million iterations.