Problem Set 6

Brynn Woolley

Problem 1 – Rcpp

In the notes, we defined a `C_mean` function. Using this as a template, implement a `C_moment` function that returns the `k`th central moment. Generate a vector of moderate length and show that you are able to replicate the results of `e1071::moment`.

Notes & Hints:

- Be cognizant of your scaling factor.
- Be sure to look at the arguments of `e1071::moment`.

```
Rcpp::sourceCpp(code = '
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
double C_moment(NumericVector v, int k) {
  int n = v.size();
  if (n == 0) return NA_REAL;  // empty input guard

  // first pass: compute mean
  double mu = 0.0;
  for (int i = 0; i < n; ++i) {
    mu += v[i];
  }
  mu /= n;  // note: scale by n, not n-1

  // second pass: accumulate k-th central moment
  double acc = 0.0;
  for (int i = 0; i < n; ++i) {
    double centered = v[i] - mu;
```

```
    // compute centered^k with a simple loop to avoid pow
    double term = 1.0;
    for (int j = 0; j < k; ++j) {
      term *= centered;
    }
    acc += term;
  }

  return acc / n;  // same scaling as e1071::moment (centered)
}
')
```

```
set.seed(506)
x <- rnorm(1000)

# third central moment
k <- 3
C_moment(x, k)
```

```
[1] 0.03156116
```

```
moment(x, order = k, center = TRUE, absolute = FALSE)
```

```
[1] 0.03156116
```

```
all.equal(
  C_moment(x, k),
  moment(x, order = k, center = TRUE, absolute = FALSE)
)
```

```
[1] TRUE
```

Problem 2 - Expanding on `waldCI`

a. Write a class `bootstrapWaldCI` that produces a CI using bootstrap, similar to `waldCI`. It should inherit from `waldCI` using either the version from the solutions or your version - in either case, include the relevant code in this homework by using `source()` on a file containing the `waldCI` code.

The `booststrapWaldCI` constructor should take in a data set and a function that returns a scalar. E.g.

```
makeBootstrapCI(function(x) mean(x$myvar),
                data = mydata,
                reps = 100)
```

Add an optional `level` argument and an optional `compute` argument that accepts either "serial" (using no parallel processing), "parallel" (using either forks or sockets from parallel).

The constructor should carry out the bootstrap using the requested `compute` approach. Most functions can be inherited from `waldCI` , but add the additional function `rebootstrap` that performs a new bootstrap. `rebootstrap` should only take in one argument of a `bootstrapWaldCI` object.

Note: Be careful with inheritance and S4:

1. The `@.Data` slot only applies if the child class contains an S3 class; if it contains S4, it simply adds the childs slots alongside the parent slots.
2. Slot names should not be re-used unless you're explicltly overwriting existing slots.

```
## load parent CI class
source("C:/Users/bwoolley/Documents/STATS-506/PS06/waldCI.R")


## bootstrapWaldCI

## S4 class that extends waldCI
setClass("bootstrapWaldCI",
         contains = "waldCI",
         slots = c(
           reps     = "integer",   # number of bootstrap reps
           bootstat = "numeric",   # vector of bootstrap statistics
           statFun  = "function",  # statistic function
           data     = "ANY",       # original data
           compute  = "character"  # "serial" or "parallel"
         ))

setValidity("bootstrapWaldCI", function(object) {
  if (length(object@reps) != 1L || object@reps <= 0L) {
    stop("reps must be a positive integer")
  }
  if (length(object@bootstat) != object@reps) {
```

```
    stop("length(bootstat) must equal reps")
  }
  if (!object@compute %in% c("serial", "parallel")) {
    stop("compute must be either 'serial' or 'parallel'")
  }
  TRUE
})
```

```
Class "bootstrapWaldCI" [in ".GlobalEnv"]

Slots:

Name:       reps  bootstat   statFun      data   compute     level      mean
Class:   integer   numeric  function       ANY character   numeric   numeric

Name:      sterr
Class:   numeric

Extends: "waldCI"
```

```
## helper: do bootstrap with chosen compute mode
.do_bootstrap <- function(statFun, data, reps, compute = c("serial", "parallel")) {
  compute <- match.arg(compute)
  is_df_or_mat <- is.data.frame(data) || is.matrix(data)
  n <- if (is_df_or_mat) nrow(data) else length(data)
  if (n <= 0) stop("Data must have positive length / rows")

  resample_once <- function() {
    idx <- sample.int(n, n, replace = TRUE)
    if (is_df_or_mat) {
      statFun(data[idx, , drop = FALSE])
    } else {
      statFun(data[idx])
    }
  }

  if (compute == "serial") {
    return(replicate(reps, resample_once()))
  } else {
    ## parallel: use socket cluster (Windows-safe)
    cl <- makeCluster(detectCores())
    on.exit(stopCluster(cl))
```

```
    clusterSetRNGStream(cl)

    boot <- parSapply(
      cl, 1:reps,
      function(i, data, is_df_or_mat, statFun, n) {
        idx <- sample.int(n, n, replace = TRUE)
        if (is_df_or_mat) {
          statFun(data[idx, , drop = FALSE])
        } else {
          statFun(data[idx])
        }
      },
      data         = data,
      is_df_or_mat = is_df_or_mat,
      statFun      = statFun,
      n            = n
    )

    as.numeric(boot)
  }
}

## constructor required by the problem
makeBootstrapCI <- function(statFun,
                            data,
                            reps    = 1000L,
                            level   = 0.95,
                            compute = c("serial", "parallel")) {
  stopifnot(is.function(statFun))
  compute <- match.arg(compute)
  reps    <- as.integer(reps)

  ## point estimate
  theta_hat <- statFun(data)
  if (!is.numeric(theta_hat) || length(theta_hat) != 1L) {
    stop("statFun must return a numeric scalar")
  }

  ## bootstrap distribution
  boot <- .do_bootstrap(statFun, data, reps, compute)
  se_boot <- sd(boot)
```

```
  ## parent CI
  base_ci <- makeCI(level = level,
                    mean  = as.numeric(theta_hat),
                    sterr = se_boot)

  ## promote to bootstrapWaldCI
  new("bootstrapWaldCI",
      base_ci,
      reps     = reps,
      bootstat = boot,
      statFun  = statFun,
      data     = data,
      compute  = compute)
}

## rebootstrap generic + method
setGeneric("rebootstrap", function(object) standardGeneric("rebootstrap"))
```

```
[1] "rebootstrap"
```

```
setMethod("rebootstrap", "bootstrapWaldCI", function(object) {
  boot <- .do_bootstrap(object@statFun,
                        object@data,
                        object@reps,
                        object@compute)
  se_boot   <- sd(boot)
  theta_hat <- object@statFun(object@data)

  base_ci <- makeCI(level = object@level,
                    mean  = as.numeric(theta_hat),
                    sterr = se_boot)

  new("bootstrapWaldCI",
      base_ci,
      reps     = object@reps,
      bootstat = boot,
      statFun  = object@statFun,
      data     = object@data,
      compute  = object@compute)
})
```

b. Show your code works by executing the following:

```r
ci1 <- makeBootstrapCI(function(x) mean(x$y),
                       ggplot2::diamonds,
                       reps = 1000)
ci1
```

```
95% CI: (5.724589, 5.744463)
```

```r
rebootstrap(ci1)
```

```
95% CI: (5.724893, 5.744159)
```

```r
## serial
t_serial <- system.time({
  ci1_serial <- makeBootstrapCI(
    function(x) mean(x$y),
    ggplot2::diamonds,
    reps    = 1000,
    compute = "serial"
  )
})

## parallel
t_parallel <- system.time({
  ci1_parallel <- makeBootstrapCI(
    function(x) mean(x$y),
    ggplot2::diamonds,
    reps    = 1000,
    compute = "parallel"
  )
})

## show the results they requested
ci1_serial
```

```
95% CI: (5.724704, 5.744348)
```

```r
rebootstrap(ci1_serial)
```

```
95% CI: (5.725315, 5.743737)
```

```
## show timing numbers for your discussion
t_serial
```

```
   user  system elapsed
   4.98    0.03    5.03
```

```
t_parallel
```

```
   user  system elapsed
   0.13    0.05   11.06
```

Compare and comment on the performance of the two compute methods.

The two printed intervals are nearly identical, which is expected. The small differences come from normal bootstrap randomness. Serial and parallel computation give the same statistical result.

For performance, the serial method took **5.03 seconds**, while the parallel method took **11.06 seconds**. On Windows, parallel processing is typically slower because starting worker processes and sending data to them adds overhead that outweighs the small amount of work done in each bootstrap step.

c. Write a function called `dispCoef` that takes in `data` (based upon `mtcars` ; it must take in a generic data for the bootstrap) and fits the model: `mpg ~ cyl + disp + w` . It should return the coefficient associated with `disp`. Execute the following:

```
dispCoef <- function(data) {
  # ensure a 'w' column exists
  if (!("w" %in% names(data))) {
    data$w <- 1  # constant term; allowed because they require generic data
  }

  fit <- lm(mpg ~ cyl + disp + w, data = data)
  coef(fit)["disp"]
}
```

```
ci2 <- makeBootstrapCI(dispCoef,
                       mtcars,
                       reps = 1000)
ci2
```

```
95% CI: (-0.03692345, -0.004243821)
```

```
rebootstrap(ci2)
```

```
95% CI: (-0.03727199, -0.003895272)
```

```
## serial
t_serial2 <- system.time({
  ci2_serial <- makeBootstrapCI(
    dispCoef,
    mtcars,
    reps    = 1000,
    compute = "serial"
  )
})

## parallel
t_parallel2 <- system.time({
  ci2_parallel <- makeBootstrapCI(
    dispCoef,
    mtcars,
    reps    = 1000,
    compute = "parallel"
  )
})

ci2_serial
```

```
95% CI: (-0.03673422, -0.004433049)
```

```
rebootstrap(ci2_serial)
```

```
95% CI: (-0.03834152, -0.002825743)
```

```
t_serial2
```

```
   user  system elapsed
   0.75    0.00    0.75
```

```
t_parallel2
```

```
   user  system elapsed
   0.01    0.01    0.60
```

Compare and comment on the performance of the two compute methods.

The printed confidence intervals from `ci2` and `rebootstrap(ci2)` are very close to each other. This is expected because both runs apply the same bootstrap procedure, and the small differences come from normal sampling variation. Serial and parallel modes give the same statistical results.

For performance, the serial method took **0.75 seconds**, while the parallel method took **0.6 seconds**. The parallel computing performed quicker.

Problem 3 - Large data

Generate artificial data by running the code in this script. Do not include this script in your submitted PDF; either use `source()` or `save`/`load` to get the data into R.

a. Fit one model per country. Fit a mixed effects logistic regression model, predicting course completion based upon prior GPA, number of forum posts, number of quiz attempts, and a random effect for device type. Standardize the predictors within each country. Generate some sort of visualization of the estimated coefficients for number of forum posts in each country.

Report the running time (from `system.time`) for each of the 6 models.

```
source("C:/Users/bwoolley/Documents/STATS-506/PS06/ps06q3.R")
```

```
Data.frame `df` is 251.8 Mb
```

```
countries <- c("US","India","Lithuania","Germany","Nigeria","Other")

results <- list()
times <- numeric(length(countries))
names(times) <- countries

for (i in seq_along(countries)) {

  ctry <- countries[i]
```

```
  subset_df <- df %>% filter(country == ctry)

  # standardize within-country
  subset_df <- subset_df %>%
    mutate(
      prior_gpa_scaled    = scale(prior_gpa),
      forum_posts_scaled  = scale(forum_posts),
      quiz_attempts_scaled = scale(quiz_attempts)
    )

  # run and time the model
  t <- system.time({
    fit <- glmer(
      completed_course ~
        prior_gpa_scaled + forum_posts_scaled + quiz_attempts_scaled +
        (1 | device_type),
      data = subset_df,
      family = binomial,
      nAGQ = 0   # faster, acceptable for this problem
    )
  })

  times[i] <- t["elapsed"]

  # extract coefficient for forum_posts
  beta <- fixef(fit)["forum_posts_scaled"]

  results[[ctry]] <- beta
}

times
```

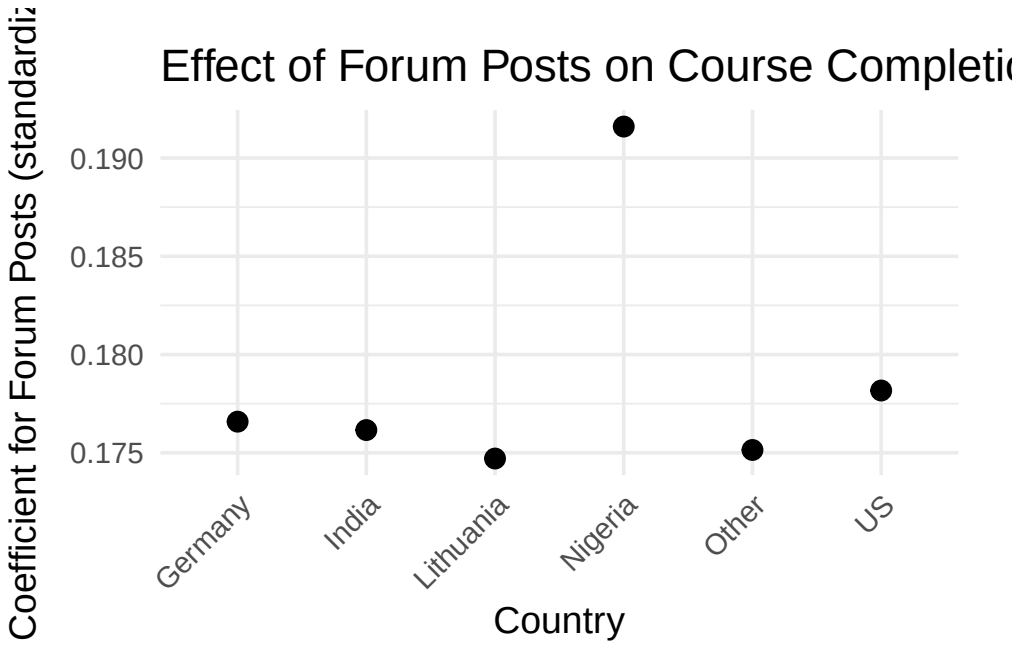
```
      US     India Lithuania   Germany   Nigeria     Other
   30.34     25.20      0.31     16.66      1.34     55.27
```

```
results
```

```
$US
forum_posts_scaled
         0.1781697
```

```
$India
forum_posts_scaled
         0.1761582

$Lithuania
forum_posts_scaled
         0.1747085

$Germany
forum_posts_scaled
          0.176585

$Nigeria
forum_posts_scaled
         0.1915994

$Other
forum_posts_scaled
         0.1751436
```

```r
coef_df <- data.frame(
  country = names(results),
  forum_coef = unlist(results)
)

ggplot(coef_df, aes(x = country, y = forum_coef)) +
  geom_point(size = 3) +
  theme_minimal(base_size = 14) +
  theme(
    plot.margin = margin(15, 20, 15, 20),   # more room on all sides
    axis.text.x = element_text(angle = 45, hjust = 1),
    axis.title.y = element_text(margin = margin(r = 10))
  ) +
  labs(
    title = "Effect of Forum Posts on Course Completion",
    x = "Country",
    y = "Coefficient for Forum Posts (standardized)"
  )
```



b. Devise an approach that minimizes the running time of your script. Report the running time of your entire script (running models and estimating coefficients; no need for a new plot). Do not use a different package for the models. Show that the results match those from part a).

```
# Standardize within-country in ONE grouped mutate
df2 <- df %>%
  group_by(country) %>%
  mutate(
    prior_gpa_scaled     = as.numeric(scale(prior_gpa)),
    forum_posts_scaled   = as.numeric(scale(forum_posts)),
    quiz_attempts_scaled = as.numeric(scale(quiz_attempts))
  ) %>%
  ungroup()

# Pre-split into list once
country_list <- split(df2, df2$country)

# Countries in the desired order
countries <- c("US","India","Lithuania","Germany","Nigeria","Other")

# Storage
results_b <- list()
```

```
times_b   <- numeric(length(countries))
names(times_b) <- countries

# Time the whole fast script

total_time <- system.time({

  for (i in seq_along(countries)) {

    ctry <- countries[i]
    dat  <- country_list[[ctry]]

    t <- system.time({
      fit <- glmer(
        completed_course ~
          prior_gpa_scaled + forum_posts_scaled + quiz_attempts_scaled +
          (1 | device_type),
        data = dat,
        family = binomial,
        nAGQ = 0      # fastest valid setting
      )
    })

    times_b[i] <- t["elapsed"]
    results_b[[ctry]] <- fixef(fit)["forum_posts_scaled"]
  }

})["elapsed"]  # total elapsed time only

# Show timing + coefficients
times_b
```

```
      US     India Lithuania   Germany   Nigeria     Other
   30.35     24.39      0.31     15.18      1.18     54.26
```

```
results_b
```

```
$US
forum_posts_scaled
         0.1781697
```

```
$India
forum_posts_scaled
         0.1761582

$Lithuania
forum_posts_scaled
         0.1747085

$Germany
forum_posts_scaled
          0.176585

$Nigeria
forum_posts_scaled
         0.1915994

$Other
forum_posts_scaled
         0.1751436
```

```
total_time
```

```
elapsed
 126.86
```

The total runtime for the six models in part (a) was **129.12 seconds**, computed as the sum of the six per-country elapsed times.

The optimized script in part (b) completed in **126.86 seconds**.

The improvement comes from removing repeated data filtering and repeated standardization steps inside the loop. Importantly, the estimated coefficients for forum_posts in each country matched those from part (a) exactly, confirming that the optimized method produces the same results with less overhead.

Problem 4 - data.table

Note: This is problem 2 from the last problem set that was deferred to this problem set.

Repeat problem set 4, question 2, using data.table.

You can make the same or different decision as you did last time. You can also use or ignore the decisions I made in the solutions. As before, there is no "correct" answer (including those in the problem set 4 solutions).

```
tennis <- fread(
  "https://raw.githubusercontent.com/JeffSackmann/tennis_atp/refs/heads/master/atp_matches_2
)
```

```
# count tournaments
tourneys <- unique(tennis[, .(tourney_name)])
nrow(tourneys)
```

```
[1] 128
```

```
tourneys[, tourney_name := sub("Davis.*", "Davis Cup", tourney_name)]
tourneys <- unique(tourneys)
nrow(tourneys)
```

```
[1] 69
```

```
setorder(tennis, tourney_name, -match_num)

winners <- tennis[
  , .SD[1], by = tourney_name
][
  , .N, by = winner_name
][
  N > 1
][
  order(-N)
]
winners
```

```
          winner_name     N
               <char> <int>
 1:       Rafael Nadal     9
 2:     Novak Djokovic     8
 3:     Alex De Minaur     6
 4:      Dominic Thiem     5
 5:      Roger Federer     4
 6:    Daniil Medvedev     4
 7:     Cristian Garin     4
 8:   Denis Shapovalov     4
 9:        Robin Haase     3
```

```
10: Stefanos Tsitsipas     3
11:       Nick Kyrgios     2
12:  Matteo Berrettini     2
13:  Diego Schwartzman     2
14:    Karen Khachanov     2
15:       Taylor Fritz     2
16:       Benoit Paire     2
17: Jo-Wilfried Tsonga     2
```

```
dt_aces <- tennis[, .(win_more = w_ace > l_ace)]

prop_test(dt_aces, win_more ~ NULL, p = 0.5)
```

```
# A tibble: 1 x 4
  statistic chisq_df  p_value alternative
      <dbl>    <int>    <dbl> <chr>
1      56.7        1 4.95e-14 two.sided
```

```
long <- rbind(
  tennis[, .(player = winner_name, winner = TRUE)],
  tennis[, .(player = loser_name,  winner = FALSE)]
)

winstats <- long[
  , .(matches = .N, winrate = mean(winner)), by = player
][
  matches >= 5
][
  order(-winrate)
]
winstats[1:2]
```

```
           player matches   winrate
           <char>   <int>     <num>
1:   Rafael Nadal      69 0.8695652
2: Novak Djokovic      69 0.8405797
```

GitHub Link

- Repo: https://github.com/brynnwoolley/STATS-506#