STATS-506 HW6

Zekai Xu

2025-11-30

Github Repo

Question 1

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

// [[Rcpp::export]]
double C_mean(NumericVector v) {
  double sum = 0.0;
  int n = v.size();
  for (int i = 0; i < n; ++i) {
    sum += v[i];
  }
  return sum / n;
}

// [[Rcpp::export]]
double C_moment(NumericVector x, int k) {
  int n = x.size();
  double mu = C_mean(x);     // sample mean
  double acc = 0.0;

  for (int i = 0; i < n; ++i) {
    double diff = x[i] - mu; // x_i - mean
    double term = 1.0;
```

```
    for (int j = 0; j < k; ++j) {
      term *= diff;
    }
    acc += term;
  }

  return acc / static_cast<double>(n);
}
')

# Test
x <- rnorm(1000)
all.equal(
  C_moment(x, 2),
  moment(x, order = 2, center = TRUE)
)
```

```
[1] TRUE
```

```
all.equal(
  C_moment(x, 3),
  moment(x, order = 3, center = TRUE)
)
```

```
[1] TRUE
```

Question 2

Part a

```
source("./waldCI.R")

#-----Define bootstrapWaldCI-----
setClass(
  "bootstrapWaldCI",
  contains = "waldCI",
  slots = c(
    reps = "integer",
```

```
    bootStats = "numeric",
    compute = "character",
    statFun = "function",
    origData = "data.frame"
  )
)

#' Bootstrap one resample
#' @param fun Function from data.frame to numeric.
#' @param data Data frame to resample.
#' @return Numeric scalar statistic.
.bootstrap_once <- function(fun, data) {
  n <- nrow(data)
  idx <- sample.int(n = n, size = n, replace = TRUE)
  fun(data[idx, , drop = FALSE])
}

#' Make a bootstrap-based Wald CI
#' @param fun Function from data.frame to numeric.
#' @param data Data frame.
#' @param reps Integer; number of bootstrap replicates.
#' @param level Confidence level.
#' @param compute "serial" or "parallel".
#' @param ncores Number of cores for parallel.
#' @return Object of class bootstrapWaldCI.
makeBootstrapCI <- function(fun,
                            data,
                            reps,
                            level   = 0.95,
                            compute = c("serial", "parallel"),
                            ncores  = detectCores()) {
  stopifnot(is.function(fun), is.data.frame(data))
  reps    <- as.integer(reps)
  compute <- match.arg(compute)

  est <- fun(data)

  if (compute == "serial") {
    bootStats <- replicate(
      n    = reps,
      expr = .bootstrap_once(fun = fun, data = data)
    )
```

```
  } else {
    if (.Platform$OS.type == "windows") {
      cl <- makeCluster(ncores)
      on.exit(stopCluster(cl), add = TRUE)
      bootStats <- parSapply(
        cl   = cl,
        X    = seq_len(reps),
        FUN  = function(i, fun, data) .bootstrap_once(fun = fun, data = data),
        fun  = fun,
        data = data
      )
    } else {
      bootStats <- unlist(
        mclapply(
          X        = seq_len(reps),
          FUN      = function(i) .bootstrap_once(fun = fun, data = data),
          mc.cores = ncores
        )
      )
    }
  }

  se <- sd(bootStats)
  z  <- qnorm(1 - (1 - level) / 2)
  ci <- est + c(-1, 1) * z * se

  new(
    "bootstrapWaldCI",
    estimate  = est,
    se        = se,
    level     = level,
    lower     = ci[1],
    upper     = ci[2],
    reps      = reps,
    bootStats = bootStats,
    compute   = compute,
    statFun   = fun,
    origData  = data
  )
}

#' Re-run bootstrap for a bootstrapWaldCI object
```

```
#' @param object A bootstrapWaldCI object.
#' @return New bootstrapWaldCI object with fresh bootstrap sample.
setGeneric(
  "rebootstrap",
  function(object) standardGeneric("rebootstrap")
)
```

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

```
setMethod(
  "rebootstrap",
  signature(object = "bootstrapWaldCI"),
  function(object) {
    makeBootstrapCI(
      fun     = object@statFun,
      data    = object@origData,
      reps    = object@reps,
      level   = object@level,
      compute = object@compute
    )
  }
)
```

Part b

```
t_serial <- system.time({
  ci1_serial <- makeBootstrapCI(
    fun     = function(x) mean(x$y),
    data    = ggplot2::diamonds,
    reps    = 1000L,
    level   = 0.95,
    compute = "serial"
  )
})

t_parallel <- system.time({
  ci1_parallel <- makeBootstrapCI(
    fun     = function(x) mean(x$y),
    data    = ggplot2::diamonds,
```

```
    reps    = 1000L,
    level   = 0.95,
    compute = "parallel"
  )
})

#' Print estimate and CI for a bootstrapWaldCI object
#' @param obj A bootstrapWaldCI object.
printBootstrapCI <- function(obj) {
  cat("estimate:", obj@estimate, "\n")
  cat("se      :", obj@se, "\n")
  cat("level   :", obj@level, "\n")
  cat("CI      : [", obj@lower, ", ", obj@upper, "]\n", sep = "")
}

## show short summaries instead of full 1000-length slot
printBootstrapCI(ci1_serial)
```

```
estimate: 5.734526
se      : 0.004624965
level   : 0.95
CI      : [5.725461, 5.743591]
```

```
printBootstrapCI(rebootstrap(ci1_serial))
```

```
estimate: 5.734526
se      : 0.004920948
level   : 0.95
CI      : [5.724881, 5.744171]
```

```
printBootstrapCI(ci1_parallel)
```

```
estimate: 5.734526
se      : 0.004776003
level   : 0.95
CI      : [5.725165, 5.743887]
```

```
printBootstrapCI(rebootstrap(ci1_parallel))
```

```
estimate: 5.734526
se      : 0.004909933
level   : 0.95
CI      : [5.724903, 5.744149]
```

```
t_serial
```

```
   user  system elapsed
  2.792   0.186   2.979
```

```
t_parallel
```

```
   user  system elapsed
  3.753   1.652   1.048
```

For repeated calls to `rebootstrap`, the point estimate stays identical while the bootstrap standard errors and Wald CIs fluctuate only slightly, indicating a stable bootstrap estimate at `reps = 1000`. Comparing timing, the parallel implementation dramatically reduces elapsed time (≈ 0.54s vs ≈ 2.68s), showing clear speedup over the serial version despite slightly higher total CPU (user) time.

Part c

```r
#' Extract coefficient of disp from linear model
#' @param data Data frame similar to mtcars.
#' @return Numeric coefficient for disp.
dispCoef <- function(data) {
  fit <- lm(mpg ~ cyl + disp + wt, data = data)
  unname(coef(fit)["disp"])
}

t2_serial <- system.time({
  ci2_serial <- makeBootstrapCI(
    fun     = dispCoef,
    data    = mtcars,
    reps    = 1000L,
    level   = 0.95,
    compute = "serial"
  )
```

```
})

t2_parallel <- system.time({
  ci2_parallel <- makeBootstrapCI(
    fun     = dispCoef,
    data    = mtcars,
    reps    = 1000L,
    level   = 0.95,
    compute = "parallel"
  )
})

printBootstrapCI(ci2_serial)
```

```
estimate: 0.007472925
se      : 0.008977356
level   : 0.95
CI      : [-0.01012237, 0.02506822]
```

```
printBootstrapCI(rebootstrap(ci2_serial))
```

```
estimate: 0.007472925
se      : 0.009242444
level   : 0.95
CI      : [-0.01064193, 0.02558778]
```

```
printBootstrapCI(ci2_parallel)
```

```
estimate: 0.007472925
se      : 0.009188475
level   : 0.95
CI      : [-0.01053616, 0.02548201]
```

```
printBootstrapCI(rebootstrap(ci2_parallel))
```

```
estimate: 0.007472925
se      : 0.009225174
level   : 0.95
CI      : [-0.01060808, 0.02555393]
```

```
t2_serial
```

```
   user  system elapsed
  0.255   0.004   0.261
```

```
t2_parallel
```

```
   user  system elapsed
  0.278   0.135   0.103
```

For the disp coefficient in the mtcars regression, both the serial and parallel bootstrap methods produce essentially the same statistical conclusions. The point estimate is identical ($ 0.00747) *across all runs, and the bootstrap standard error sand* 95 $0.11s vs $ $0.34s) even though its total CPU (user) time is slightly higher, reflecting the overhead and concurrency of parallel processing.

Question 3

```
source("./data.R")
```

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

Part a

```
# ensure df$country is a factor with the 6 levels
df$country <- factor(df$country)

countries <- levels(df$country)

fits_slow   <- vector("list", length(countries))
names(fits_slow) <- countries
times_slow  <- vector("list", length(countries))
names(times_slow) <- countries

for (i in seq_along(countries)) {
  ctry <- countries[i]
```

```
  message("Fitting model for country: ", ctry)

  # subset data for this country
  df_ctry <- df[df$country == ctry, ]

  # standardize predictors within this country
  df_ctry$prior_gpa_z      <- as.numeric(scale(df_ctry$prior_gpa))
  df_ctry$forum_posts_z    <- as.numeric(scale(df_ctry$forum_posts))
  df_ctry$quiz_attempts_z  <- as.numeric(scale(df_ctry$quiz_attempts))

  # time each model fit
  times_slow[[ctry]] <- system.time({
    fits_slow[[ctry]] <- glmer(
      completed_course ~ prior_gpa_z + forum_posts_z + quiz_attempts_z +
        (1 | device_type),
      data   = df_ctry,
      family = binomial()
    )
  })
}
```

```
Fitting model for country: Germany

Fitting model for country: India

Fitting model for country: Lithuania

Fitting model for country: Nigeria

Fitting model for country: Other

Fitting model for country: US
```

```
# collect timing results for each of the 6 models
time_table <- do.call(rbind, times_slow)
time_table
```

```
           user.self sys.self elapsed user.child sys.child
Germany       18.883    1.479  20.368          0         0
India         29.933    2.259  32.197          0         0
Lithuania      0.528    0.236   0.765          0         0
Nigeria        1.186    0.148   1.335          0         0
Other         62.007    6.544  68.561          0         0
US            37.608    2.217  39.839          0         0
```

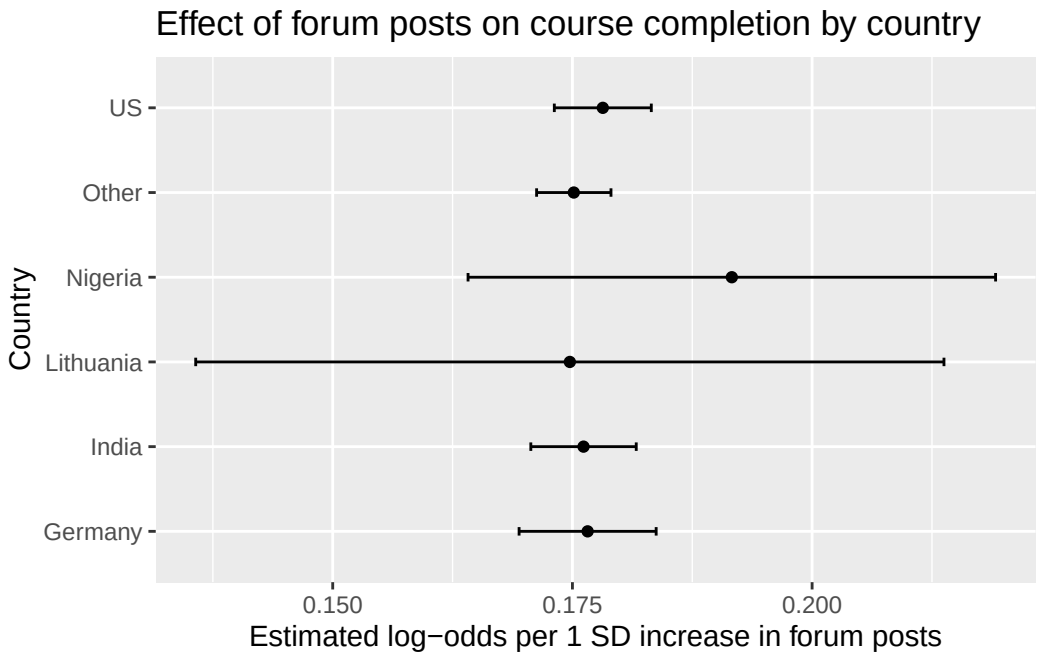
```r
# extract coefficient for forum_posts_z from each model
coef_posts_slow <- do.call(
  rbind,
  lapply(names(fits_slow), function(ctry) {
    sm <- summary(fits_slow[[ctry]])$coefficients
    beta <- sm["forum_posts_z", "Estimate"]
    se   <- sm["forum_posts_z", "Std. Error"]
    data.frame(
      country  = ctry,
      estimate = beta,
      lower    = beta - 1.96 * se,
      upper    = beta + 1.96 * se
    )
  })
)

coef_posts_slow
```

```
    country  estimate     lower     upper
1   Germany 0.1765868 0.1694397 0.1837340
2     India 0.1761593 0.1706619 0.1816568
3 Lithuania 0.1747323 0.1357115 0.2137532
4   Nigeria 0.1916268 0.1641164 0.2191373
5     Other 0.1751442 0.1712663 0.1790220
6        US 0.1781707 0.1731131 0.1832284
```

```r
# visualization of forum_posts effect by country
ggplot(coef_posts_slow,
       aes(x = country, y = estimate)) +
  geom_point() +
  geom_errorbar(aes(ymin = lower, ymax = upper), width = 0.1) +
  coord_flip() +
  labs(
    x = "Country",
```

```
    y = "Estimated log-odds per 1 SD increase in forum posts",
    title = "Effect of forum posts on course completion by country"
  )
```



Part b

Use `parallel::mclapply` to fit six models in parallel, which is expected to reduce the elapsed time for the script.

```
countries <- levels(df$country)

# choose number of cores
ncores <- min(detectCores(), length(countries))

# run everything (standardization + model fits + coefficient extraction) in parallel
time_fast <- system.time({
  fits_fast <- mclapply(
    countries,
    function(ctry) {
      df_ctry <- df[df$country == ctry, ]

      # standardize predictors within this country
```

```
      df_ctry$prior_gpa_z      <- as.numeric(scale(df_ctry$prior_gpa))
      df_ctry$forum_posts_z    <- as.numeric(scale(df_ctry$forum_posts))
      df_ctry$quiz_attempts_z  <- as.numeric(scale(df_ctry$quiz_attempts))

      glmer(
        completed_course ~ prior_gpa_z + forum_posts_z + quiz_attempts_z +
          (1 | device_type),
        data   = df_ctry,
        family = binomial()
      )
    },
    mc.cores = ncores
  )

  names(fits_fast) <- countries
})

time_fast  # total runtime for all 6 models in parallel
```

```
   user  system elapsed
117.448   8.170  83.982
```

```
# extract forum_posts_z coefficient from slow (part a) fits
beta_slow <- setNames(
  coef_posts_slow$estimate,
  coef_posts_slow$country
)

# extract forum_posts_z coefficient from fast (parallel) fits
beta_fast <- sapply(countries, function(ctry) {
  sm <- summary(fits_fast[[ctry]])$coefficients
  sm["forum_posts_z", "Estimate"]
})

beta_slow
```

```
  Germany     India Lithuania   Nigeria     Other        US
0.1765868 0.1761593 0.1747323 0.1916268 0.1751442 0.1781707
```

```
beta_fast
```

```
  Germany     India Lithuania   Nigeria     Other        US
0.1765868 0.1761593 0.1747323 0.1916268 0.1751442 0.1781707
```

```
# check that they match (up to numerical tolerance)
all.equal(unname(beta_fast[countries]), unname(beta_slow[countries]))
```

```
[1] TRUE
```

Question 4

Part a

```
# Load Dataset
url <- "https://raw.githubusercontent.com/JeffSackmann/tennis_atp/refs/heads/master/atp_matc
tennis <- fread(url)

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

cat("Number of tournaments that took place in 2019: ", dim(tourneys)[1], "\n")
```

```
Number of tournaments that took place in 2019:  69
```

Part b

```
winners <- tennis[
  round == "F",
  .(n_tournaments = uniqueN(tourney_id)),
  by = .(winner_id)
][ order(-n_tournaments) ]

multi_winners <- winners[ n_tournaments > 1 ]

cat("Number of players who won more than one tournament: ", nrow(multi_winners), "\n")
```

```
Number of players who won more than one tournament:  12
```

```
cat("Number of tournaments that the most winning player win: ",
    winners[1, n_tournaments], "\n")
```

```
Number of tournaments that the most winning player win:  5
```

Part c

We use Bootstrap to build a 95% confidence interval for the mean difference without assuming normality, and the hypothesis is:

$$H_0 : ace_{winner} - ace_{loser} > 0, \quad H_1 : ace_{winner} - ace_{loser} \leq 0$$

```
# 1) Per-match ace difference (winner - loser)
ace_diff <- tennis[ , .(diff = as.numeric(w_ace) - as.numeric(l_ace)) ][ !is.na(diff) ]

# 2) Observed mean
obs_mean <- ace_diff[ , mean(diff) ]

# 3) Bootstrap (5000 resamples with replacement)
set.seed(506)
B <- 5000L
n <- nrow(ace_diff)

boot_stat <- replicate(B, {
  idx <- sample.int(n, n, replace = TRUE)
  mean(ace_diff$diff[idx])
})

# 4) One-sided percentile confidence intervals
lower_bound <- unname(quantile(boot_stat, probs = 0.05))

# Output and brief explanation
cat("Observed mean difference (winner - loser): ", obs_mean, "\n")
```

```
Observed mean difference (winner - loser):  1.7049
```

```
cat("One-sided 95% LOWER CI (percentile): [", lower_bound, ", Inf)\n", sep = "")
```

```
One-sided 95% LOWER CI (percentile): [1.483296, Inf)
```

Part d

```
# Long form: one row per player per match, with win indicator
long_players <- rbindlist(
  list(
    tennis[ , .(player_id = winner_id, player_name = winner_name, win = 1L) ],
    tennis[ , .(player_id = loser_id,  player_name = loser_name,  win = 0L) ]
  ),
  use.names = TRUE
)

# Summarise to matches, wins, and win_rate per player
win_rate <- long_players[ ,
  .(matches = .N, wins = sum(win)),
  by = .(player_id, player_name)
][
  , win_rate := wins / matches
][
  matches >= 5
][
  order(-win_rate, -wins, -matches)
]

# Show the table
print(win_rate)
```

```
     player_id     player_name matches  wins  win_rate
         <int>           <char>   <int> <int>     <num>
  1:    104745    Rafael Nadal      69    60 0.8695652
  2:    104925  Novak Djokovic      69    58 0.8405797
  3:    103819   Roger Federer      66    55 0.8333333
  4:    106421 Daniil Medvedev      80    59 0.7375000
  5:    104731  Kevin Anderson      15    11 0.7333333
 ---
163:    106058       Jack Sock       5     1 0.2000000
164:    104810          Zhe Li       7     1 0.1428571
165:    111200      Elias Ymer       8     1 0.1250000
166:    106075   Jozef Kovalik      10     1 0.1000000
167:    105155     Pedro Sousa       9     0 0.0000000
```

```r
# Text output for the top player
cat("The player with the highest win-rate is: ", win_rate$player_name[1], "\n")
```

```
The player with the highest win-rate is:  Rafael Nadal
```