Assignment 6

Maksim Nikiforov

Read in the data set

Since we have a comma-separated file, we can read in our data using the function `read_csv()`.

```
concentrations <- read_csv(file = "./concentration.csv")
```

```
## Rows: 400 Columns: 6
```

```
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (1): variety
## dbl (5): Total_lignin, Glucose, Xylose, Arabinose, concentration
```

```
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
concentrations
```

```
## # A tibble: 400 x 6
##    variety     Total_lignin Glucose Xylose Arabinose concentration
##    <chr>              <dbl>   <dbl>  <dbl>     <dbl>         <dbl>
##  1 M.giganteus         17.7    28.0   16.5      23.2             0
##  2 M.giganteus         13.6    21.1   21.0      25.5             0
##  3 M.giganteus         15.8    23.4   21.4      26.9             0
##  4 M.giganteus         11.5    22.7   21.6      29.8             0
##  5 M.giganteus         15.3    19.9   20.1      24.3             0
##  6 M.giganteus         13.9    20.6   23.2      24.4             0
##  7 M.giganteus         14.8    23.8   22.6      27.6             0
##  8 M.giganteus         10.2    23.3   19.8      29.0             0
##  9 M.giganteus         15.7    26.8   21.8      28.0             0
## 10 M.giganteus         18.5    22.0   20.2      27.6             0
## # ... with 390 more rows
```

Implement the bootstrap

Use a ***`for`*** *loop to implement the bootstrap (use the* ***`sample`*** *function from base R or* ***`sample_n`*** *from dplyr for resampling) for fitting a quadratic model using* ***concentration*** *as the predictor and* ***Total_lignin*** *as the response (code for a quadratic model fit is lm(y~x+I(x^2), data = data_set) - you'll need to extract the coefficients from the returned object to get your estimate within each iteration of the loop). Report an estimate of the maximum with a corresponding standard error.*

We set an arbitrary seed (50) to get the same set of numbers every time we run this code. Next, we gather the number of rows in our *concentrations* data set (400) to use within the bootstrap. We store the re-sampled pairs in the *samples* tibble and we fit our quadratic relationship to this tibble. The outcome is stored in the *linear_model* list. From this list, we can calculate a point estimate of the maximum of the curve by extracting β_1 and β_2 using `linear_model$coefficients[2]` and `linear_model$coefficients[3]`, respectively. With a seed of 50, the estimate of the maximum is 42.03 and the estimate of the standard error is 0.78.

```r
# Set seed for repeatability
set.seed(50)

# n for number of observations in the data set
n <- dim(concentrations)[1]

# Initialize vector to store maximums
maximums <- vector()

# Sample with replacement
# Repeat 1000 times
for (i in 1:1000){
  samples <- sample_n(concentrations, n, replace = TRUE)
  linear_model <- lm(Total_lignin ~ concentration + I(concentration^2), data = samples)
  # Extract coefficients from the returned object to get estimate within each iteration
  maximums[i] <- (-1)*linear_model$coefficients[2]/(2*linear_model$coefficients[3])
}

# Report an estimate of the maximum
mean(maximums)
```

```
## [1] 42.02752
```

```r
# Report standard error
sd(maximums)
```

```
## [1] 0.7768011
```

Bootstrap with replicate

Redo the bootstrap analysis for the Total_lignin response but make use of the replicate function instead of a for loop.

Hint: To do this, I created a function called bootFun that essentially did everything within one iteration of a for loop. bootFun took in only the data set the predictor, and the response to use (both variable names in quotes).

Report an estimate of the maximum with a corresponding standard error.

In lieu of a `for()` loop, we create a function called `bootFun`. This function takes in the data set name, the predictor variable, and the response variable and provides a maximum. We use the `paste()` function to fill our quadratic model fit formula with user-provided variables. We then feed this formula to `lm()`. With a seed of 50, this again produces 42.03 as the estimate of the maximum and 0.78 as the estimate of the standard error.

```r
# Set seed for repeatability
set.seed(50)

# Create a function called bootFun with dataset, predictor, and response as options
bootFun <- function(dataset, predictor, response){
  samples <- sample_n(dataset, n, replace = TRUE)
  # Generate formula
  quadraticModelFit <- paste(response, "~", predictor, '+', "I(", predictor, "^2)")
  # Feed above formula to lm function
  linear_model <- lm(formula = quadraticModelFit, data = samples)
  # Generate maximums
  maximums <- (-1)*linear_model$coefficients[2]/(2*linear_model$coefficients[3])
  return(maximums[[1]])
}

# Use replicate to run bootFun function 1000 times
max_estimate <- replicate(1000, bootFun(dataset = concentrations,
                        predictor = "concentration", response = "Total_lignin"))

# Report mean and standard error
mean(max_estimate)
```

```
## [1] 42.02752
```

```
sd(max_estimate)
```

```
## [1] 0.7768011
```

Create wrapper for `replicate`

Create a wrapper function for replicate that will return the standard deviation of the bootstrapped estimates. (A wrapper function is just a function that calls another function.) Hint: I created a function called seBootFun that takes in resp, pred, B, and data and returns the standard deviation of the bootstrapped estimates.

Apply this function using Total_lignin as the response. Apply this function using Glucose as the response.

Our `seBootFun` wrapper function calls `replicate` and supplies it with `resp`, `pred`, `B`, and `data` as options. The standard deviation with Total_lignin as the response is 0.78. The standard deviation with glucosG as the response is 1.09.

```
# Set seed for repeatability
set.seed(50)

# Create a function called seBootFun that takes in resp, pred, B, and data

seBootFun <- function(resp, pred, B = 5000, data){
  max_estimate <- replicate(B, bootFun(dataset = data,
                          predictor = pred, response = resp))
  # Returns the standard deviation of the bootstrapped estimates.
  return(sd(max_estimate))
}

# Apply this function using Total_lignin as the response.
totalLigninSD <- seBootFun("Total_lignin", "concentration", 1000, concentrations)
totalLigninSD
```

```
## [1] 0.7768011
```

```
# Apply this function using Glucose as the response.
glucoseSD <- seBootFun("Glucose", "concentration", 1000, concentrations)
glucoseSD
```

```
## [1] 1.085409
```

Use `lapply`

Create a vector with the response variable names. Use lapply to apply your seBootFun to this vector (you should get back four standard error estimates!).

Following the directions and setting the seed to 50, we find that the following standard error estimates:

- Total_lignin: 0.78
- Glucose: 1.09
- Xylose: 0.41
- Arabinose: 0.26

```
# Set seed for repeatability
set.seed(50)

# Create a vector with the response variable names.
dataVars <- c("Total_lignin", "Glucose", "Xylose", "Arabinose")

# Use lapply to apply your seBootFun to this vector.
lapply(X = dataVars, FUN = seBootFun, pred = "concentration", B = 1000, data = concentrations)
```

```
## [[1]]
## [1] 0.7768011
##
## [[2]]
## [1] 1.085409
##
## [[3]]
## [1] 0.4054824
##
## [[4]]
## [1] 0.2605531
```

Parallelize

Now we want to find the estimate of the standard error for each of our possible response variables (Total_lignin, Glucose, Xylose, Arabinose) using parallel computing. Let's use parallel computing to send each of the four bootstrap standard error computations (one for each response) to a different core (if you only have a dual core, use two cores).

Use the code in the notes to translate what you've done above to be done in parallel.

We initiate the `parallel` library to make the `parLapply` function available to us. We then detect the number of cores using `detectCores()` and set our cluster to a total of 4 cores (one for each response variable). FInally, we run `parLapply` as a parallel-enabled alternative to `lapply`.

```
# Initiate library for parallel computations
library(parallel)

# Detect cores on local machine
cores <- detectCores()
cores
```

```
## [1] 8
```

```
# "On Mac/Linux you have the option of using makeCluster(no_core, type="FORK") that automatically conta
# Source: https://www.r-bloggers.com/2015/02/how-to-go-parallel-in-r-basics-tips/
cluster <- makeCluster(cores - 4, type="FORK")
cluster
```

```
## socket cluster with 4 nodes on host 'localhost'
```

```
# Replace lapply with parLapply for parallel computing
resultsPar <- parLapply(cluster, X = dataVars, fun = seBootFun, pred = "concentration", B = 5000, data =
# Clean up cluster to free up cores
stopCluster(cluster)

# Display results of parallel computation
str(resultsPar)
```

```
## List of 4
##  $ : num 0.795
##  $ : num 1.11
##  $ : num 0.393
##  $ : num 0.259
```

Report estimated maximums

Along with the standard errors you found in the parallel computing section, report the estimated maximum from the linear model fit using the full dataset (rather than a bootstrap sample). There is no need to do anything fancy here, just run the lm function for each of the response variables and find the estimated maximums from those models.

Report these estimates and their standard errors in a table.

```
# Initiate vectors for estimated maximums (estMax) and estimated standard error (estSE)
estMax <- vector()
estSE <- vector()

# Apply linear model to all four response variables, stored in dataVars (created earlier)
for (i in 1:length(dataVars)){
  quadraticModelFit <- paste(dataVars[i], "~", "concentration", '+', "I(", "concentration", "^2)")
  linModel <- lm(formula = quadraticModelFit, data = concentrations)
  estMax[i] <- (-1)*linModel$coefficients[2]/(2*linModel$coefficients[3])
  estSE[i] <- resultsPar[[i]]
}

# Create vector of all four response variables (removing the underscore in Total_lignin)
responseVariables <- c("Lignin", "Glucose", "Xylose", "Arabinose")

# Create tibble with estimated values and rename columns
finalNumbers <- tibble(responseVariables, estMax, estSE)
names(finalNumbers) <- c("", "Max", "SE")

# Report these estimates and their standard errors in a table.
finalNumbers
```

```
## # A tibble: 4 x 3
##   ``          Max    SE
##   <chr>     <dbl> <dbl>
## 1 Lignin     42.0 0.795
## 2 Glucose    30.4 1.11
## 3 Xylose     40.3 0.393
## 4 Arabinose  30.1 0.259
```

Benchmark (optional)

We can run benchmarks against the `lapply` and the `parLapply` functions to estimate differences in the compute time. We see that the mean time for running `lapply` is roughly double the time needed to run tasks in parallel (using `parLapply`).

```r
# Benchmark lapply
parTime <- microbenchmark({lapply(X = dataVars, FUN = seBootFun, pred = "concentration", B = 50, data =
  }, times = 100, unit = "s")
```

```
## Warning in microbenchmark({: less accurate nanosecond times to avoid potential
## integer overflows
```

```r
# Re-allocate 4 cores for parallel computing
cluster <- makeCluster(cores - 4, type="FORK")
cluster
```

```
## socket cluster with 4 nodes on host 'localhost'
```

```r
# Benchmark parLapply
straightTime <- microbenchmark({parLapply(cluster, X = dataVars, fun = seBootFun, pred = "concentration
}, times = 100, unit = "s")

# Free up cores
stopCluster(cluster)

# Output benchmark results
parTime
```

```
## Unit: seconds
##
##  {     lapply(X = dataVars, FUN = seBootFun, pred = "concentration",          B = 50, data = concent
##        min        lq      mean    median        uq       max neval
##  0.2207334 0.2238513 0.2269127 0.2253097 0.2264608 0.2658056   100
```

```r
straightTime
```

```
## Unit: seconds
##
##  {     parLapply(cluster, X = dataVars, fun = seBootFun, pred = "concentration",          B = 50, da
##         min         lq       mean     median         uq       max neval
##  0.07413878 0.07747643 0.08027595 0.07928951 0.08040881 0.1720672   100
```