hw2-stats506-lijiabao

JiabaoLi

link of github: https://github.com/lijiabao203/stats506__rwork

Problem 1 - Dice Game

a.

First, construct an error test function to make sure the input number is a positive integer:

```r
errorread_pi <- function(inpu){
  # input is the dice numbers
  # will cause stop if the input is not a positive integer
  number = suppressWarnings(as.integer(inpu))
  if (number != inpu){
    stop("Error input, please input an integer.")
  }else if(is.na(number)){
    stop("Error input, input is NA")
  }else if(number <= 0) {
    stop("Error input, input is not positive.")
  }
  return(TRUE)
}
```

For all these versions of functions, the input is the number of dice to roll. and the output is total winnings

- Version 1: Implement this game using a loop.

```r
play_dice1 <- function(dice_times){
  errorread_pi(dice_times)

  winning = 0
  # use function sample to generate random value of the outcome of the dice
  dice_numbers = sample(1:6, dice_times, replace = TRUE)
  for (dic_num in dice_numbers){
    if (dic_num == 3 || dic_num == 5){
      winning = winning + 2 * dic_num
    }else{
      winning = winning - 2
    }
  }
  return(winning)
}
```

- Version 2: Implement this game using built-in R vectorized functions.

```r
play_dice_sub <- function(dic_number){
  # this is a built-in R vectorized functions
  # the input is a vector or a number, which should be the values of dices
  # the output should be the winnings for each dice
  for(i in seq_along(dic_number)){
    if(dic_number[i] == 3 || dic_number[i] == 5){
      dic_number[i] = 2 * dic_number[i]
    }else{
      dic_number[i] = -2
    }
  }

  return(dic_number)
}

play_dice2 <- function(dice_times){
  errorread_pi(dice_times)

  # generate random values of dices via function sample
  dic_numbers = sample(1:6, dice_times, replace = TRUE)
  winning = play_dice_sub(dic_numbers)

  return(sum(winning))
}
```

- Version 3: Implement this by rolling all the dice into one and collapsing the die rolls into a single `table()`. (Hint: Be careful indexing the table - what happens if you make a table of a single dice roll? You may need to look to other resources for how to solve this.)

```r
play_dice3 <- function(dice_times){
  errorread_pi(dice_times)
  dice_numbers = sample(1:6, dice_times, replace = TRUE)

  dice_numbers_table = table(dice_numbers)
  # Calculate winnings using cal_winnings to show them of items in table
  cal_winnings = ifelse(as.numeric(names(dice_numbers_table)) %in% c(3, 5),
                        2 * as.numeric(names(dice_numbers_table)), -2)
  # use winnings of specific dice values times
  # times these values occur to get the answer
  winnings = sum(cal_winnings * dice_numbers_table)

  return(winnings)
}
```

- Version 4: Implement this game by using one of the "`apply`" functions.

```r
play_dice4 <- function(dice_times){
  errorread_pi(dice_times)

  dic_numbers = sample(1:6, dice_times, replace = TRUE)
  # apply values to the function like play_dice_sub in version 2
  winning = sum(sapply(dic_numbers, function(dic_number){
    if(dic_number == 3 || dic_number == 5){
      return(2*dic_number)
    }else{
      return(-2)
    }
  }))

  return(winning)
}
```

b. Demonstrate that all versions work. Do so by running each a few times, once with an input a 3, and once with an input of 3,000.

Check the function with input: 3, 30, 300, 3000

```
c(play_dice1(3), play_dice1(30), play_dice1(300), play_dice1(3000))
```

```
[1]   10   32  336 4100
```

```
c(play_dice1(3), play_dice1(30), play_dice1(300), play_dice1(3000))
```

```
[1]    2   32  280 4076
```

```
c(play_dice1(3), play_dice1(30), play_dice1(300), play_dice1(3000))
```

```
[1]   -6  -12  516 3952
```

```
c(play_dice1(3), play_dice1(30), play_dice1(300), play_dice1(3000))
```

```
[1]   18   52  348 4100
```

c. Demonstrate that the four versions give the same result. Test with inputs 3 and 3,000. (You will need to add a way to control the randomization.)

Use set.seed to check if the out put are same.

```
set.seed(2024)
play_dice1(3)
```

```
[1] 18
```

```
set.seed(2024)
play_dice1(3000)
```

```
[1] 4140
```

```
set.seed(2024)
play_dice2(3)
```

```
[1] 18
```

```r
set.seed(2024)
play_dice2(3000)
```

```
[1] 4140
```

```r
set.seed(2024)
play_dice3(3)
```

```
[1] 18
```

```r
set.seed(2024)
play_dice3(3000)
```

```
[1] 4140
```

```r
set.seed(2024)
play_dice4(3)
```

```
[1] 18
```

```r
set.seed(2024)
play_dice4(3000)
```

```
[1] 4140
```

They are same, so they give the same result.

d. Use the *microbenchmark* package to clearly demonstrate the speed of the implementations. Compare performance with a low input (1,000) and a large input (100,000). Discuss the results

The function using loop is the fastest, the function using vector is second fastest, the function using table is third fastest and the function using apply function is the slowest. It's apparent based on the table of summary of the run time test of 100 times test.

I think the reason is that easier struct is more efficient when solving problems. But methods like using apply function can solve more complex problems, which is still useful and easy to use and understand.

```r
library(microbenchmark)
# use package to judge the speed.
benchmark_low_input_results = microbenchmark(
  Loop = play_dice1(1000),
  Vector = play_dice2(1000),
  Table = play_dice3(1000),
  Apply = play_dice4(1000)
)
benchmark_large_input_results = microbenchmark(
  Loop = play_dice1(100000),
  Vector = play_dice2(100000),
  Table = play_dice3(100000),
  Apply = play_dice4(100000)
)
print(benchmark_low_input_results)
```

```
Unit: microseconds
   expr   min     lq    mean median     uq    max neval cld
   Loop  72.8  76.05  79.302  77.45  79.65  191.5   100 a  
 Vector  92.4  97.05 102.397  99.25 102.15  219.3   100 ab 
  Table 111.0 126.10 144.853 135.85 151.80  327.9   100  b 
  Apply 427.8 436.40 504.587 454.15 483.00 2253.4   100   c
```

```r
print(benchmark_large_input_results)
```

```
Unit: milliseconds
   expr     min       lq      mean   median       uq      max neval cld
   Loop  6.9080  7.19860  7.472671  7.35425  7.60485   9.1701   100  a 
 Vector  8.7685  9.19075  9.562933  9.38720  9.73740  11.7239   100  a 
  Table  6.3439  7.18375  7.815549  7.57535  8.04060  15.2721   100  a 
  Apply 49.6296 57.29545 72.139638 68.69630 80.62685 126.7979   100   b
```

e. Do you think this is a fair game? Defend your decision with evidence based upon a Monte Carlo simulation.

From the expectation, $E(totalwinnings) = -2 \times \frac{2}{3} + 6 \times \frac{1}{6} + 10 \times \frac{1}{6} = \frac{4}{3}$, which means this game is good for player because they can earn money from the statistical perspective.

Here is the Monte Carlo simulation: compute 1000000 times play_dice(1) and compute the mean and std.

```
MC = replicate(1000000, play_dice1(1))
c("mean"=mean(MC), "std"=sd(MC))
```

```
    mean      std 
1.338208 4.854458 
```

From the outcome of the MC simulation, this is not a fair game while players are more likely to get benefit.

Problem 2 - Linear Regression

Download the cars data set available at https://corgis-edu.github.io/corgis/csv/cars/. The goal is to examine the relationship between torque and highway gas mileage.

a. The names of the variables in this data are way too long. Rename the columns of the data to more reasonable lengths.

```
data_cars = read.csv("cars.csv")
names(data_cars) = c("H", "L", "W", "DriveLine", "EngType", "Hybrid",
                     "Gears", "Transmission", "City", "FuelType",
                     "Highway", "class", "ID", "Make", "ModelYear",
                     "Year", "EngHorsepower", "Torque")
names(data_cars)
```

```
 [1] "H"             "L"             "W"             "DriveLine"    
 [5] "EngType"       "Hybrid"        "Gears"         "Transmission" 
 [9] "City"          "FuelType"      "Highway"       "class"        
[13] "ID"            "Make"          "ModelYear"     "Year"         
[17] "EngHorsepower" "Torque"       
```

b. Restrict the data to cars whose Fuel Type is "Gasoline".

Show the number of rows to identify I successfully constrain it.

```
data_cars_gasoline = data_cars[which(data_cars$FuelType == "Gasoline"),]
print(c(nrow(data_cars), nrow(data_cars_gasoline)))
```

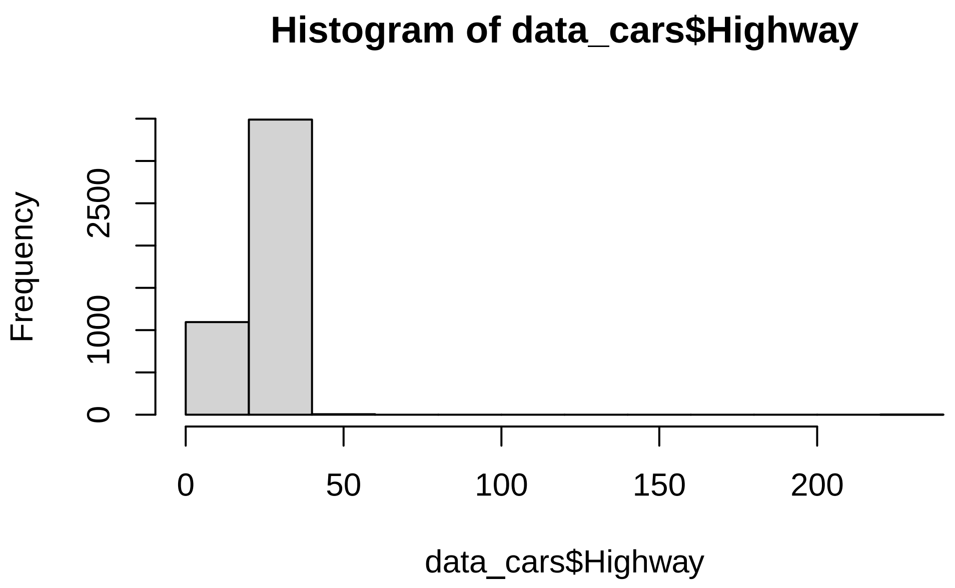
```
[1] 5076 4591
```

```
data_cars = data_cars_gasoline
```

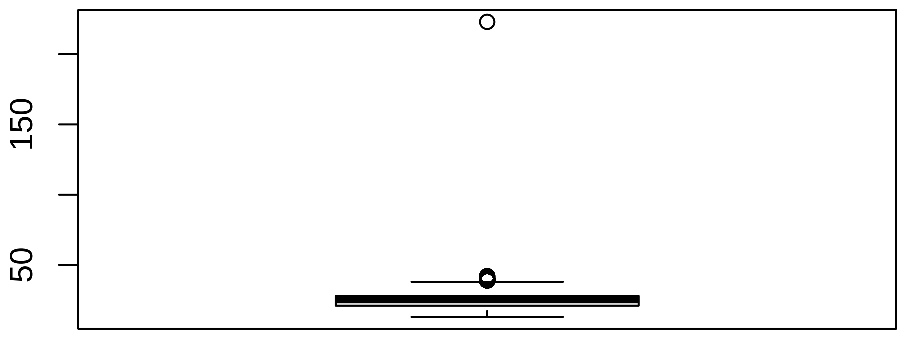
c. Examine the distribution of highway gas mileage. Consider whether a transformation could be used. If so, generate the transformed variable and *use this variable going forward*. If not, provide a short justification.

From the graph generated with function "hist" and "boxplot", there is a big extreme number. And from the summary with the skewness value which is really bigger than 0 and the kurtosis value which is really bigger than 3, more values are at the lower side, which is roughly from 13 to 21, with fewer observations at higher side. It might be difficult to solve the extreme value using transformation, but for the right skewness problem, we can solve it using log transformation.

```
library(moments)
hist(data_cars$Highway)
```



```
boxplot(data_cars$Highway)
```



```r
head(sort(data_cars$Highway, decreasing = TRUE), 10)
```

```
 [1] 223  42  42  42  41  41  41  40  40  40
```

```r
table(data_cars$Highway)
```

```

 13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 
  9  10  22  50 138 262 281 323 274 235 371 267 405 323 302 259 167 160 161  99 
 33  34  35  36  37  38  39  40  41  42 223 
105 125  92  75  16  14  12  27   3   3   1 
```

```r
summary(data_cars$Highway)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
  13.00   21.00   25.00   24.97   28.00  223.00 
```

```r
skewness(data_cars$Highway)
```

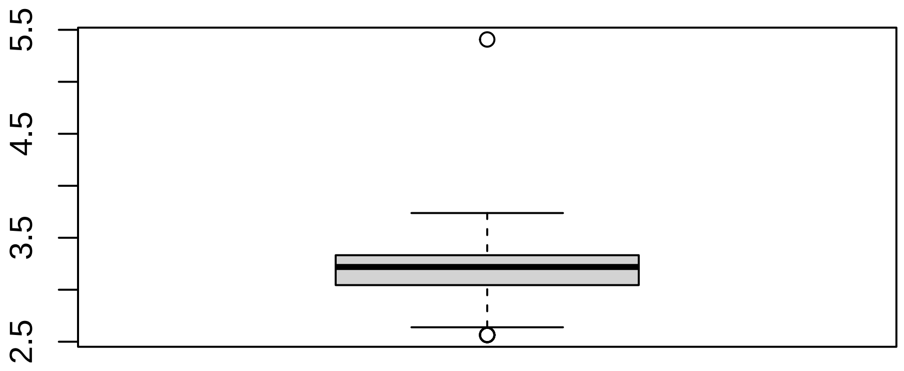
```
[1] 7.993507
```

```r
kurtosis(data_cars$Highway)
```

```
[1] 254.4671
```

Plot again and it seems better now because the absolute value of the skewness is lower than 0.5.

```r
data_cars$Highway = sapply(data_cars$Highway, log)
boxplot(data_cars$Highway)
```



```r
skewness(data_cars$Highway)
```

```
[1] 0.230209
```

```r
kurtosis(data_cars$Highway)
```

```
[1] 4.876462
```

d. Fit a linear regression model predicting MPG on the highway. The predictor of interest is torque. Control for:

```
-   The horsepower of the engine

-   All three dimensions of the car

-   The year the car was released, as a categorical variable.

Briefly discuss the estimated relationship between torque and highway MPG. Be precise about
```

The coefficient for torque means how much highway MPG changes with a one-unit increase in torque, when other variables in the model are stable. In this model, while the coefficient is -3.307053e-05 , it means that for every additional unit of torque, the highway MPG decreases by 3.307053e-05 , while all else variables are not changed.

```r
lm_model = lm(Highway ~ Torque + EngHorsepower + H + L + W +
                factor(ModelYear), data = data_cars)
lm_model[["coefficients"]]["Torque"]
```

```
       Torque
-3.307053e-05
```

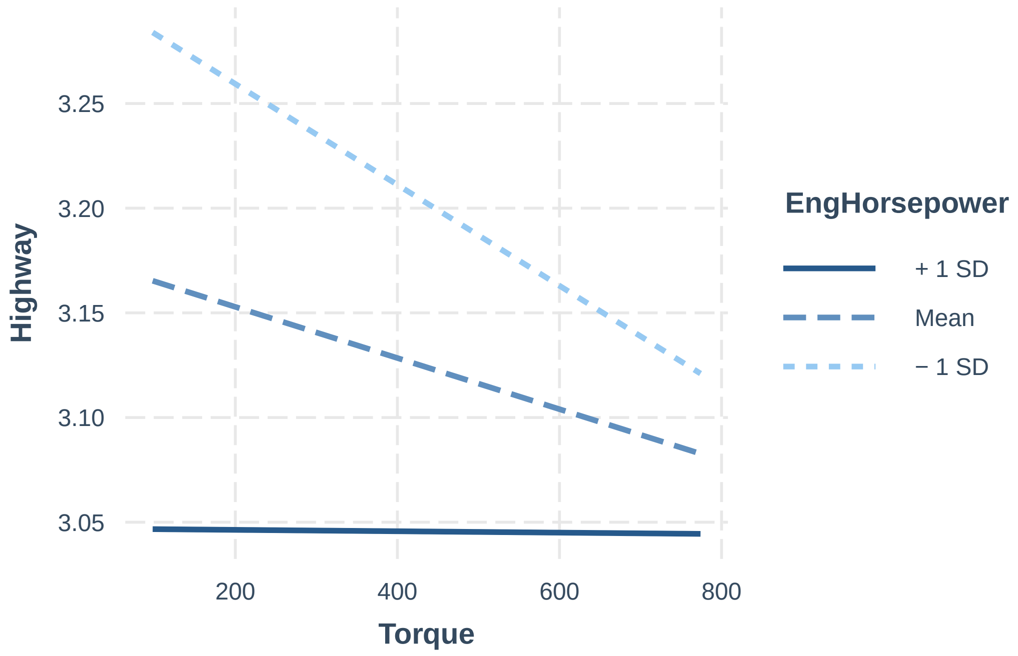
e. Refit the model (with `lm`) and generate an interaction plot

Firstly, use lm function to refit the model:

```r
interaction_model = lm(Highway ~ Torque * EngHorsepower + H + L + W +
                         factor(ModelYear), data = data_cars)
```

Secondly, use interactions package to fit, and choose year as “2012 Volvo XC90”

```r
library(interactions)
# Choose a reasonable year, which has max number of cars data.
interact_plot(interaction_model, pred = Torque, modx = EngHorsepower,
              at = list(ModelYear = "2012 Volvo XC90"))
```



Additionally, if we need to choose the value of EngHorsepower, besides Mean value and +1 SD and -1 SD, 1st Qu and mean and 3rd Qu is also reasonable.

f. Calculate $\hat{\beta}$ from d manually.

First, the design matrix X is:

```
X = model.matrix(~ Torque + EngHorsepower + H + L + W + factor(ModelYear),
                  data = data_cars)
```

So, we have: $\hat{\beta} = (X^T X)^{-1} X^T Y$

```
y = data_cars$Highway

hat_matrix = solve(t(X) %*% X) %*% t(X) %*% y
```

Second, we need to compare the estimated coefficient computed with design matrix and coefficients computed from lm function. The function all.equal is a good choice:

```
all.equal(c(hat_matrix), c(as.matrix(lm_model$coefficient)))
```

```
[1] TRUE
```

These values are the same. So we successfully compute the $\hat{\beta}$ without using lm function.