# 506 Problem Set 1

Xiaohan Liu

## Table of contents

GitHub repository: https://github.com/EmiiilyLiu/STATS_506

## Problem 1

### (a)

```r
## Import data into a data frame
data <- read.csv("F:/Desktop/STATS 506/PS1/wine/wine.data", header = FALSE)
data <- as.data.frame(data)

## Add column names
colnames(data) <- c('class', 'Alcohol', 'Malic acid', 'Ash', 'Alcalinity of ash',
                    'Magnesium', 'Total phenols', 'Flavanoids', 'Nonflavanoid phenols',
                    'Proanthocyanins','Color intensity', 'Hue',
                    'OD280/OD315 of diluted wines', 'Proline' )
```

**(b)**

```
# Ensure that the number of wines within each class is correct
cat("The number of wines in class 1 is", sum(data$class==1), "\n")
```

```
The number of wines in class 1 is 59
```

```
cat("The number of wines in class 2 is", sum(data$class==2), "\n")
```

```
The number of wines in class 2 is 71
```

```
cat("The number of wines in class 3 is", sum(data$class==3), "\n")
```

```
The number of wines in class 3 is 48
```

```
ifelse(sum(data$class==1)==59 & sum(data$class==2)==71 & sum(data$class==3)==48,
       'Correct', 'Not correct')
```

```
[1] "Correct"
```

**(c)**

```
# 1
data$class[which.max(data$Alcohol)]
```

```
[1] 1
```

```
# 2
data$class[which.min(data$Alcohol)]
```

```
[1] 2
```

```r
# 3
average_magnesium <- 114
sum(data$Magnesium > average_magnesium)
```

```
[1] 26
```

```r
# 4
num_class <- length(unique(data$class))
for(i in 1:num_class){
  cat('Class', i, 'has', sum(data$Magnesium[data$class==i]>average_magnesium),
      'wines with higher levels of magnesium than average German beer\n')
}
```

```
Class 1 has 15 wines with higher levels of magnesium than average German beer
Class 2 has 6 wines with higher levels of magnesium than average German beer
Class 3 has 5 wines with higher levels of magnesium than average German beer
```

**(d)**

```r
## Calculate average of each class
class_avg <- aggregate(data[ ,-1], list(data$class), mean)
class_avg$Group.1 <- paste("Class", class_avg$Group.1)
rownames(class_avg) <- class_avg$Group.1

## Calculate overall average
overall_avg <- colMeans(data[,-1])
overall_avg <- as.data.frame(t(overall_avg), row.names = 'Overall')

## Combine into a single table
df_avg <- rbind(overall_avg, class_avg[,-1])
print(df_avg)
```

```
         Alcohol Malic acid      Ash Alcalinity of ash Magnesium Total phenols
Overall 13.00062   2.336348 2.366517          19.49494  99.74157      2.295112
Class 1 13.74475   2.010678 2.455593          17.03729 106.33898      2.840169
Class 2 12.27873   1.932676 2.244789          20.23803  94.54930      2.258873
Class 3 13.15375   3.333750 2.437083          21.41667  99.31250      1.678750
        Flavanoids Nonflavanoid phenols Proanthocyanins Color intensity
```

```
Overall  2.0292697           0.3618539        1.590899        5.058090
Class 1  2.9823729           0.2900000        1.899322        5.528305
Class 2  2.0808451           0.3636620        1.630282        3.086620
Class 3  0.7814583           0.4475000        1.153542        7.396250
              Hue OD280/OD315 of diluted wines   Proline
Overall 0.9574494                     2.611685  746.8933
Class 1 1.0620339                     3.157797 1115.7119
Class 2 1.0562817                     2.785352  519.5070
Class 3 0.6827083                     1.683542  629.8958
```

## (e)

*3* tests will be used.

1. $H_0 : Ash_{\text{Class 1}} = Ash_{\text{Class 2}}$ VS $H_1 : Ash_{\text{Class 1}} \neq Ash_{\text{Class 2}}$

```
Ash_1 <- data$Ash[data$class==1]
Ash_2 <- data$Ash[data$class==2]
Ash_3 <- data$Ash[data$class==3]

t.test(Ash_1, Ash_2)
```

```
    Welch Two Sample t-test

data:  Ash_1 and Ash_2
t = 4.4184, df = 125.59, p-value = 2.124e-05
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 0.116383 0.305226
sample estimates:
mean of x mean of y
 2.455593  2.244789
```

Given that the $p-value = 2.124e-05$ less than 0.05, we can reject $H_0$ and conclude that the level of Ash differs between Class 1 and Class 2.

2. $H_0 : Ash_{\text{Class 1}} = Ash_{\text{Class 3}}$ VS $H_1 : Ash_{\text{Class 1}} \neq Ash_{\text{Class 3}}$

```
t.test(Ash_1, Ash_3)
```

```
    Welch Two Sample t-test

data:  Ash_1 and Ash_3
t = 0.46489, df = 105, p-value = 0.643
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -0.06043717  0.09745695
sample estimates:
mean of x mean of y 
 2.455593  2.437083 
```

Given that the $p-value = 0.643$ larger than 0.05, we have no enough evidence to reject $H_0$ conclude that the level of Ash differs between Class 1 and Class 3.

3. $H_0 : Ash_{\text{Class 2}} = Ash_{\text{Class 3}}$ VS $H_1 : Ash_{\text{Class 2}} \neq Ash_{\text{Class 3}}$

```r
t.test(Ash_2, Ash_3)
```

```
    Welch Two Sample t-test

data:  Ash_2 and Ash_3
t = -4.184, df = 114.96, p-value = 5.627e-05
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -0.2833328 -0.1012564
sample estimates:
mean of x mean of y 
 2.244789  2.437083 
```

Given that the $p-value = 5.627e-05$ less than 0.05, we can reject $H_0$ and conclude that the level of Ash differs between Class 2 and Class 3.

Manually, construct a function to calculate $p-value$:

```r
p_value <- function(l1, l2){
  n1 <- length(l1)
  n2 <- length(l2)
  mean1 <- mean(l1)
  mean2 <- mean(l2)
  sd1 <- sd(l1)
  sd2 <- sd(l2)
  se <- sqrt((sd1^2/n1) + (sd2^2/n2))
  t <- (mean1 - mean2) / se

  df <- ((sd1^2/n1 + sd2^2/n2)^2) / ((sd1^2/n1)^2/(n1-1) + (sd2^2/n2)^2/(n2-1))

  p_value <- 2 * (1 - pt(abs(t), df))

  return(p_value)
}

p_value(Ash_1,Ash_2)
```

```
[1] 2.124341e-05
```

```
p_value(Ash_1,Ash_3)
```

```
[1] 0.642973
```

```
p_value(Ash_2,Ash_3)
```

```
[1] 5.626653e-05
```

## Problem 2

### (a)

```
isPerfectPower <- function(num, power){
  if(num==0 || power<0){
    return("Invalid input")
  }else if(num<0){
    root <- (-num)^(1/power)
  }else{
    root <- num^(1/power)
  }

  if (abs(root - round(root)) < 1e-10) {
    if(num<0 & (power%%2==1)){
```

```r
        return(list(isPerfect = TRUE, root = -root))
      }else if(num>0){
        return(list(isPerfect = TRUE, root = root))
      }else{
        return(list(isPerfect = FALSE, root = NULL))
      }
    }
    else {
      return(list(isPerfect = FALSE, root = NULL))
    }
  }
```

**(b)**

```r
findRootPower <- function(num){
  if(num==0){
    return(paste0(num, " is not a valid input."))
  }

  flag=F

  for (i in 2:round(log2(abs(num)))){
    res=isPerfectPower(num,i)
    if(res$isPerfect){
      flag=T
      if(res$root>0){
        return(paste0(num,'=',res$root,'^',i))
      }else{
        return(paste0(num,'=(',res$root,')^',i))
      }
    }
  }
  if(!flag){
    return(paste0(num,' is not a perfect power'))
  }
}

findRootPower(27)
```

```
[1] "27=3^3"
```

```r
findRootPower(-27)
```

```
[1] "-27=(-3)^3"
```

```r
findRootPower(13060694016)
```

```
[1] "13060694016=6^13"
```

```r
findRootPower(7776)
```

```
[1] "7776=6^5"
```

```r
findRootPower(170859375)
```

```
[1] "170859375=15^7"
```

```r
findRootPower(58247422)
```

```
[1] "58247422 is not a perfect power"
```

```r
findRootPower(94143178827)
```

```
[1] "94143178827=3^23"
```

## Problem 3

### (a)

Here are two R functions: one to determine the name of a 5-card stud poker hand from a given set of suits and ranks, and another to simulate dealing a round of cards in a poker game with a specified number of players:

```r
# Function to determine the name of a poker hand
get_poker_hand_name <- function(suits, ranks) {
  if (!is.vector(suits) || !is.vector(ranks)
      || length(suits) != 5 || length(ranks) != 5) {
    stop("Input vectors must have exactly 5 elements.")
  }

  # Sort ranks in descending order
  sorted_ranks <- sort(ranks, decreasing = TRUE)

  # Check for a straight flush
  if (all(diff(sorted_ranks) == 1) && length(unique(suits)) == 1) {
    return("Straight Flush")
  }

  # Check for four of a kind
  if (any(table(ranks) == 4)) {
    return("Four of a Kind")
  }

  # Check for a full house
  if (all(table(ranks) %in% c(2, 3))) {
    return("Full House")
  }

  # Check for a flush
  if (length(unique(suits)) == 1) {
    return("Flush")
  }

  # Check for a straight
  if (all(diff(sorted_ranks) == 1)) {
    return("Straight")
  }

  # Check for three of a kind
  if (any(table(ranks) == 3)) {
    return("Three of a Kind")
  }

  # Check for two pair
  if (sum(table(ranks) == 2) == 4) {
    return("Two Pair")
  }

  # Check for one pair
  if (any(table(ranks) == 2)) {
    return("One Pair")
  }

  # If none of the above, it's a high card hand
  return("High Card")
}

# Function to simulate dealing a round of cards in poker
deal_poker_round <- function(num_players) {
  if (num_players < 2 || num_players > 8) {
    stop("Number of players must be between 2 and 8.")
  }

  # Define the deck of cards
  suits <- c("Hearts", "Diamonds", "Clubs", "Spades")
  ranks <- c("2", "3", "4", "5", "6", "7", "8", "9", "10",
             "Jack", "Queen", "King", "Ace")

  # Create a deck by combining suits and ranks
  deck <- expand.grid(Suit = suits, Rank = ranks)

  # Shuffle the deck
  set.seed(123)  # For reproducibility
  shuffled_deck <- deck[sample(nrow(deck)), ]

  # Deal 5 cards to each player
  player_hands <- matrix("", nrow = num_players, ncol = 5)
  for (i in 1:num_players) {
    player_hands[i, ] <- paste(shuffled_deck[((i - 1) * 5 + 1):(i * 5), ],
                               collapse = " of ")
  }

  # Determine and display the name of each player's hand
  hand_names <- sapply(1:num_players, function(i) {
    get_poker_hand_name(suits = shuffled_deck[((i - 1) * 5 + 1):(i * 5), "Suit"],
                              ranks = shuffled_deck[((i - 1) * 5 + 1):(i * 5), "Rank"])
  })

  cat("Player Hands:\n")
  for (i in 1:num_players) {
    cat(sprintf("Player %d: %s - %s\n", i,
                paste(player_hands[i, ], collapse = ", "),
                hand_names[i]))
  }
}

# Example usage:
# To simulate dealing a round of poker with 4 players:
# deal_poker_round(4)
```

You can call the deal_poker_round function with the desired number of players to simulate a round of poker and display the hands along with their names determined by the get_poker_hand_name function.

## (b)

Two functions both generate some errors.

```
suits <- c("Hearts", "Diamonds", "Clubs", "Spades", "Hearts")
ranks <- c("Ace", "King", "Queen", "Jack", "10")
#get_poker_hand_name(suits, ranks)
```

Error in r[i1] - r[-length(r):-(length(r) - lag + 1L)] : non-numeric argument to binary operator

Error in base::try(findRootPower, silent = TRUE) : object 'findRootPower' not found

```
#deal_poker_round(4)
```

Error in get_poker_hand_name(suits = shuffled_deck[((i - 1) * 5 + 1):(i * : Input vectors must have exactly 5 elements.

The first function, (I delete the comments given by ChatGPT, and put the explanation of the fixed part in comments)

```
get_poker_hand_name <- function(suits, ranks) {
  if (!is.vector(suits) || !is.vector(ranks) ||
      length(suits) != 5 || length(ranks) != 5) {
    stop("Input vectors must have exactly 5 elements.")
  }

## Since some functions used below only accept numeric input,
## but there exists non-numeric input, so I use *match* function
## to transfer into numeric values
  rank_values <- match(ranks, c("2", "3", "4", "5", "6", "7", "8", "9", "10",
                                "Jack", "Queen", "King", "Ace"))

  sorted_ranks <- sort(rank_values, decreasing = TRUE)

## According to the website, the code given by ChatGPT
## misses Royal Flush, so I add it here
  if (all(sorted_ranks == c(13, 12, 11, 10, 9)) && length(unique(suits)) == 1) {
    return("Royal Flush")
  }

### Change all ranks into sorted_ranks
  if (all(diff(sorted_ranks) == 1) && length(unique(suits)) == 1) {
    return("Straight Flush")
  }

  if (any(table(sorted_ranks) == 4)) {
    return("Four of a Kind")
  }

  if (all(table(sorted_ranks) %in% c(2, 3))) {
    return("Full House")
  }

  if (length(unique(suits)) == 1) {
    return("Flush")
  }

  if (all(diff(sorted_ranks) == 1)) {
    return("Straight")
  }
```

```r
  if (any(table(sorted_ranks) == 3)) {
    return("Three of a Kind")
  }

  if (sum(table(sorted_ranks) == 2) == 4) {
    return("Two Pair")
  }

  if (any(table(sorted_ranks) == 2)) {
    return("One Pair")
  }

  return("High Card")
}

suits <- c("Hearts", "Diamonds", "Clubs", "Spades", "Hearts")
ranks <- c("Ace", "King", "Queen", "Jack", "10")
get_poker_hand_name(suits, ranks)
```

```
[1] "High Card"
```

```r
suits <- c("Hearts", "Hearts", "Hearts", "Hearts", "Hearts")
ranks <- c("Ace", "King", "Queen", "Jack", "10")
get_poker_hand_name(suits, ranks)
```

```
[1] "Royal Flush"
```

```r
suits <- c("Hearts", "Clubs", "Diamonds", "Spades", "Hearts")
ranks <- c("4", "4", "4", "4", "King")
get_poker_hand_name(suits, ranks)
```

```
[1] "Four of a Kind"
```

The second function,

```r
deal_poker_round <- function(num_players) {
  if (num_players < 2 || num_players > 8) {
    stop("Number of players must be between 2 and 8.")
  }

  suits <- c("Hearts", "Diamonds", "Clubs", "Spades")
  ranks <- c("2", "3", "4", "5", "6", "7", "8", "9", "10",
             "Jack", "Queen", "King", "Ace")

  deck <- expand.grid(Suit = suits, Rank = ranks)

  set.seed(123)
  shuffled_deck <- deck[sample(nrow(deck)), ]

  player_hands <- matrix("", nrow = num_players, ncol = 5)
  for (i in 1:num_players) {
    #player_hands[i, ] <- paste(shuffled_deck[((i - 1) * 5 + 1):(i * 5), ]
    #, collapse = " of ")
    player_hands[i, ] <- apply(shuffled_deck[((i - 1) * 5 + 1):(i * 5),], 1,
                             function(x)paste(x,collapse = " of "))
  }

  hand_names <- sapply(1:num_players, function(i) {
    get_poker_hand_name(suits = as.vector(shuffled_deck[((i - 1) * 5 + 1):(i * 5), "Suit"]
                        ranks = as.vector(shuffled_deck[((i - 1) * 5 + 1):(i * 5), "Rank"]
  })

  cat("Player Hands:\n")
  for (i in 1:num_players) {
    cat(sprintf("Player %d: %s - %s\n", i, paste(player_hands[i, ],
                                        collapse = ", "),hand_names[i]))
  }
}

deal_poker_round(4)
```

```
Player Hands:
Player 1: Clubs of 9, Clubs of 5, Diamonds of 5, Clubs of 2, Diamonds of Queen - One Pair
Player 2: Clubs of Queen, Hearts of Jack, Diamonds of Ace, Hearts of 8, Diamonds of 8 - One
Player 3: Clubs of 8, Hearts of 3, Spades of King, Spades of 8, Hearts of 4 - One Pair
Player 4: Hearts of 9, Clubs of 10, Spades of 3, Clubs of King, Clubs of 3 - One Pair
```

From the definition of *hand_names*, we know that the expected output of *player_hands* should be a $5 * 5$ matrix, with element in the form of *"(a certain suit) of (a certain rank)"*:

```r
## expected output
num_players <- 4
suits <- c("Hearts", "Diamonds", "Clubs", "Spades")
ranks <- c("2", "3", "4", "5", "6", "7", "8", "9", "10",
           "Jack", "Queen", "King", "Ace")

deck <- expand.grid(Suit = suits, Rank = ranks)

set.seed(123)
shuffled_deck <- deck[sample(nrow(deck)), ]

player_hands <- matrix("", nrow = num_players, ncol = 5)
for (i in 1:num_players) {
  player_hands[i, ] <- apply(shuffled_deck[((i - 1) * 5 + 1):(i * 5),], 1,
                               function(x)paste(x,collapse = " of "))
}

player_hands
```

```
     [,1]              [,2]              [,3]               [,4]
[1,] "Clubs of 9"      "Clubs of 5"      "Diamonds of 5"    "Clubs of 2"
[2,] "Clubs of Queen"  "Hearts of Jack"  "Diamonds of Ace"  "Hearts of 8"
[3,] "Clubs of 8"      "Hearts of 3"     "Spades of King"   "Spades of 8"
[4,] "Hearts of 9"     "Clubs of 10"     "Spades of 3"      "Clubs of King"
     [,5]
[1,] "Diamonds of Queen"
[2,] "Diamonds of 8"
[3,] "Hearts of 4"
[4,] "Clubs of 3"
```

But the code given by ChatGPT cannot get the expected output (see below), which need adjusted.

```r
## Output of player_hands given by ChatGPT
num_players <- 4
suits <- c("Hearts", "Diamonds", "Clubs", "Spades")
ranks <- c("2", "3", "4", "5", "6", "7", "8", "9", "10",
           "Jack", "Queen", "King", "Ace")

deck <- expand.grid(Suit = suits, Rank = ranks)
```

```r
set.seed(123)
shuffled_deck <- deck[sample(nrow(deck)), ]

player_hands <- matrix("", nrow = num_players, ncol = 5)
for (i in 1:num_players) {
  player_hands[i, ] <- paste(shuffled_deck[((i - 1) * 5 + 1):(i * 5), ],
                             collapse = " of ")

}

player_hands
```

```
     [,1]
[1,] "c(3, 3, 2, 3, 2) of c(8, 4, 4, 1, 11)"
[2,] "c(3, 1, 2, 1, 2) of c(11, 10, 13, 7, 7)"
[3,] "c(3, 1, 4, 4, 1) of c(7, 2, 12, 7, 3)"
[4,] "c(1, 3, 4, 3, 3) of c(8, 9, 2, 12, 2)"
     [,2]
[1,] "c(3, 3, 2, 3, 2) of c(8, 4, 4, 1, 11)"
[2,] "c(3, 1, 2, 1, 2) of c(11, 10, 13, 7, 7)"
[3,] "c(3, 1, 4, 4, 1) of c(7, 2, 12, 7, 3)"
[4,] "c(1, 3, 4, 3, 3) of c(8, 9, 2, 12, 2)"
     [,3]
[1,] "c(3, 3, 2, 3, 2) of c(8, 4, 4, 1, 11)"
[2,] "c(3, 1, 2, 1, 2) of c(11, 10, 13, 7, 7)"
[3,] "c(3, 1, 4, 4, 1) of c(7, 2, 12, 7, 3)"
[4,] "c(1, 3, 4, 3, 3) of c(8, 9, 2, 12, 2)"
     [,4]
[1,] "c(3, 3, 2, 3, 2) of c(8, 4, 4, 1, 11)"
[2,] "c(3, 1, 2, 1, 2) of c(11, 10, 13, 7, 7)"
[3,] "c(3, 1, 4, 4, 1) of c(7, 2, 12, 7, 3)"
[4,] "c(1, 3, 4, 3, 3) of c(8, 9, 2, 12, 2)"
     [,5]
[1,] "c(3, 3, 2, 3, 2) of c(8, 4, 4, 1, 11)"
[2,] "c(3, 1, 2, 1, 2) of c(11, 10, 13, 7, 7)"
[3,] "c(3, 1, 4, 4, 1) of c(7, 2, 12, 7, 3)"
[4,] "c(1, 3, 4, 3, 3) of c(8, 9, 2, 12, 2)"
```

**(c)**

```r
# Define a function named get_poker_hand_name with two inputs: suits and ranks
get_poker_hand_name <- function(suits, ranks) {

  # Check if the inputs are vectors and if their lengths are 5.
  # Otherwise, raise an error
  if (!is.vector(suits) || !is.vector(ranks) ||
      length(suits) != 5 || length(ranks) != 5) {
    stop("Invalid input")
  }

  # Since the rank in poker games are not all numerical values,
  # so use match function to convert into numerical values$
  rank_values <- match(ranks, c("2", "3", "4", "5", "6", "7", "8", "9", "10",
                                "Jack", "Queen", "King", "Ace"))

  # Sort ranks in descending order
  sorted_ranks <- sort(rank_values, decreasing = TRUE)

  # Check if inputs satisfy Royal Flush:
  # ranks after *match* equal to 14, 13, 12, 11, 10,
  # corresponding to (Ace, King, Queen, Jack, 10) respectively,
  # and they have the same suit
  if (all(sorted_ranks == c(13, 12, 11, 10, 9)) && length(unique(suits)) == 1) {
    return("Royal Flush")
  }

  # If Royal Flush is not satisfied,
  # check if inputs satisfy Straight Flush:
  # All of the neighboring values in *sorted_ranks*
  # are different by 1 (there is a error, since we order ranks
  # in descending direction,
  # so it should be diff(sorted_ranks) == -1),
  # and they have the same suit
  if (all(diff(sorted_ranks) == 1) && length(unique(suits)) == 1) {
    return("Straight Flush")
  }

  # If the above are both unsatisfied,
  # check if inputs satisfy Four of a kind:
```

```
# there are four cards with the same rank
if (any(table(sorted_ranks) == 4)) {
  return("Four of a Kind")
}

# If the above are all unsatisfied,
# check if inputs satisfy Full House:
# any three of the five have the same rank,
# and the remaining two have the same rank of another kind
if (all(table(ranks) %in% c(2, 3))) {
  return("Full House")
}

# If the above are all unsatisfied,
# check if inputs satisfy Flush: 5 cards have the same suit
if (length(unique(suits)) == 1) {
  return("Flush")
}

# If the above are all unsatisfied,
# check if inputs satisfy Flush: 5 cards have the same suit
if (all(diff(sorted_ranks) == 1)) {
  return("Straight")
}

# If the above are all unsatisfied,
# check if inputs satisfy Three of a Kind:
# any three of the five have the same rank
if (any(table(ranks) == 3)) {
  return("Three of a Kind")
}

# If the above are all unsatisfied,
# check if inputs satisfy Two Pair:
# two of the five cards have the same rank,
# other two of the remaining three cards have the same rank.
if (sum(table(ranks) == 2) == 4) {
  return("Two Pair")
}

# If the above are all unsatisfied,
```

```r
  # check if inputs satisfy One Pair:
  # two of the five cards have the same rank
  if (any(table(ranks) == 2)) {
    return("One Pair")
  }

  # None conditions are satisfied, just return High Card
  return("High Card")
}
```

```r
# Define a function named deal_poker_round with a input: num_players
deal_poker_round <- function(num_players) {
  # The num_player only accept value larger or equal to 2 and smaller or equal to 8,
  # otherwise, raise an error message
  if (num_players < 2 || num_players > 8) {
    stop("Number of players must be between 2 and 8.")
  }

  # Create a vector variable named suits,
  # containing all kinds of suits in a Poker game
  suits <- c("Hearts", "Diamonds", "Clubs", "Spades")

  # Create a vector variable named ranks,
  # containing all kinds of ranks in a Poker game
  ranks <- c("2", "3", "4", "5", "6", "7", "8", "9", "10",
             "Jack", "Queen", "King", "Ace")

  # Create a data frame named deck,
  # containing all combinations (52*2) of suit and rank
  deck <- expand.grid(Suit = suits, Rank = ranks)

  # Set a seed in order for the random process reproducible
  set.seed(123)

  # Random sampling with frequency equal to the number of rows of deck
  # without replacement, in order to shuffle the above combinations randomly
  shuffled_deck <- deck[sample(nrow(deck)), ]

  # Create a matrix, whose elements are all empty strings,
  # named 'player_hands', with the number of rows equal to num_players
  # and 5 columns (to store the 5 cards for each player).
  player_hands <- matrix("", nrow = num_players, ncol = 5)

  # Deal cards to each player using a for loop
  for (i in 1:num_players) {

    # In ith iteration, take 5 rows from shuffled_deck
    # (shuffled_deck[((i - 1) * 5 + 1):(i * 5),])
    # and distribute to the i-th player.
    # By apply function with a custom function,
    # format each card with the form "(rank) of (suit)".
    # Store the strings in the i-th row of player_hands
    player_hands[i, ] <- apply(shuffled_deck[((i - 1) * 5 + 1):(i * 5),], 1,
                               function(x)paste(x,collapse = " of "))
  }

  # Using sapply function to iterate every player
  # and get the hand of each player by calling function get_poker_hand_name.
  # Store the result in a vector hand_names
  hand_names <- sapply(1:num_players, function(i) {
    get_poker_hand_name(suits = as.vector(shuffled_deck[((i - 1) * 5 + 1)
                                                         :(i * 5), "Suit"]),
                        ranks = as.vector(shuffled_deck[((i - 1) * 5 + 1)
                                                         :(i * 5), "Rank"]))
  })

  # Print the result
  cat("Player Hands:\n")
  # Use a loop to print the card of each player
  for (i in 1:num_players) {
    # Use cat to indicate the form of output
    cat(sprintf("Player %d: %s - %s\n", i, paste(player_hands[i, ],
                                             collapse = ", "),hand_names[i]))
  }
}
```

## (d)

1. Input and output are as required. *get_poker_hand_name* inputs two vectors and outputs the name of the hand. However, according to the given website, Royal Flush is missing. The second inputs the number of players and output each player's hand and the corresponding name.

2. All the hands are valid through random sampling without replacement and distribute the cards in order.
3. Not all hands are named correctly.

```r
# Expected Output: Royal Flush
suits <- c("Hearts", "Hearts", "Hearts", "Hearts", "Hearts")
ranks <- c("Ace", "King", "Queen", "Jack", "10")
get_poker_hand_name(suits, ranks)

# Expected Output: Straight Flush. WRONG!
suits <- c("Hearts", "Hearts", "Hearts", "Hearts", "Hearts")
ranks <- c("9", "8", "7", "6", "5")
get_poker_hand_name(suits, ranks)

# Expected Output: Four of a Kind
suits <- c("Hearts", "Clubs", "Diamonds", "Spades", "Hearts")
ranks <- c("4", "4", "4", "4", "King")
get_poker_hand_name(suits, ranks)

# Expected Output: Full House
suits <- c("Hearts", "Clubs", "Diamonds", "Spades", "Hearts")
ranks <- c("4", "4", "4", "King", "King")
get_poker_hand_name(suits, ranks)

# Expected Output: Flush. WRONG!
suits <- c("Hearts", "Hearts", "Hearts", "Hearts", "Hearts")
ranks <- c("Ace", "King", "10", "8", "5")
get_poker_hand_name(suits, ranks)

# Expected Output: Straight. WRONG!
suits <- c("Hearts", "Clubs", "Diamonds", "Spades", "Hearts")
ranks <- c("6", "5", "4", "3", "2")
get_poker_hand_name(suits, ranks)

# Expected Output: Three of a Kind
suits <- c("Hearts", "Clubs", "Diamonds", "Spades", "Hearts")
ranks <- c("Queen", "Queen", "Queen", "King", "Ace")
get_poker_hand_name(suits, ranks)

# Expected Output: Two Pair. WRONG!
suits <- c("Hearts", "Clubs", "Diamonds", "Spades", "Hearts")
```

```r
ranks <- c("2", "2", "King", "King", "Ace")
get_poker_hand_name(suits, ranks)

# Expected Output: One Pair
suits <- c("Hearts", "Clubs", "Diamonds", "Spades", "Hearts")
ranks <- c("10", "10", "King", "Queen", "Ace")
get_poker_hand_name(suits, ranks)

# Expected Output: High Card
suits <- c("Hearts", "Clubs", "Diamonds", "Spades", "Hearts")
ranks <- c("Ace", "King", "Queen", "Jack", "9")
get_poker_hand_name(suits, ranks)
```

The outputs: [1] “Royal Flush” [1] “High Card” [1] “Four of a Kind” [1] “Full House” [1] “High Card” [1] “High Card” [1] “Three of a Kind” [1] “One Pair” [1] “One Pair” [1] “High Card”

Fix:

```r
get_poker_hand_name <- function(suits, ranks) {
  if (!is.vector(suits) || !is.vector(ranks) ||
      length(suits) != 5 || length(ranks) != 5) {
    stop("Input vectors must have exactly 5 elements.")
  }

  rank_values <- match(ranks, c("2", "3", "4", "5", "6", "7", "8", "9", "10",
                                "Jack", "Queen", "King", "Ace"))

  sorted_ranks <- sort(rank_values, decreasing = TRUE)

  if (all(sorted_ranks == c(13, 12, 11, 10, 9)) && length(unique(suits)) == 1) {
    return("Royal Flush")
  }

  ## diff(sorted_ranks) == -1 instead of 1
  if (all(diff(sorted_ranks) == -1) && length(unique(suits)) == 1) {
    return("Straight Flush")
  }

  if (any(table(sorted_ranks) == 4)) {
    return("Four of a Kind")
  }
  if (all(table(sorted_ranks) %in% c(2, 3))) {
    return("Full House")
  }

  if (length(unique(suits)) == 1) {
    return("Flush")
  }

  ## diff(sorted_ranks) == -1 instead of 1
  if (all(diff(sorted_ranks) == -1)) {
    return("Straight")
  }

  if (any(table(sorted_ranks) == 3)) {
    return("Three of a Kind")
  }

  # sum(table(sorted_ranks) == 2) == 2 instead of 4
  if (sum(table(sorted_ranks) == 2) == 2) {
    return("Two Pair")
  }

  if (any(table(sorted_ranks) == 2)) {
    return("One Pair")
  }

  return("High Card")
}
```

```
# Expected Output: Straight Flush. WRONG!
suits <- c("Hearts", "Hearts", "Hearts", "Hearts", "Hearts")
ranks <- c("9", "8", "7", "6", "5")
get_poker_hand_name(suits, ranks)
```

```
[1] "Straight Flush"
```

```
# Expected Output: Straight. WRONG!
suits <- c("Hearts", "Clubs", "Diamonds", "Spades", "Hearts")
ranks <- c("6", "5", "4", "3", "2")
get_poker_hand_name(suits, ranks)
```

```
[1] "Straight"
```

```
# Expected Output: Two Pair. WRONG!
suits <- c("Hearts", "Clubs", "Diamonds", "Spades", "Hearts")
ranks <- c("2", "2", "King", "King", "Ace")
get_poker_hand_name(suits, ranks)
```

```
[1] "Two Pair"
```

4. It ensures no duplicates in cards. If you ask for more than 10 hands to be dealt, an error will be raised since the function are restricted $2 <= num_players <= 8$. In theory, a deck of cards can guarantee up to ten without duplicates.