Essential R: Data Analysis, Visualization, and Modeling

Anju Sambasiavn

2024-08-03

Contents

1 Part-1

- Programming Language: Designed for statistical computing and data analysis.
- Free Software: Open-source and free to use.
- Statistical Tools: Includes a wide range of statistical and graphical techniques.
- Data Visualization: Excellent for creating plots and charts.
- Extensible: Allows users to add custom functions and packages.
- Popular in Academia: Widely used in research and education.
- Community Support: Strong user community and numerous online resources.

1.1 getwd(): Get info of current working directory.

```
getwd()
```

1.2 Types of variable

- Characters
- Numeric(real number)
- Logical
- Integer
- Factor
- Complex

1.2.1 Characters:

- Character variables store text data.

```
name <- "Anju"
city <- "Christchurch"
name
city
```

1.2.2 Numeric:

- Numeric variables store numerical data such as integers or decimals.

```
age <- 30
temperature <- 25.5
age
temperature
```

1.2.3 Logical:

- Logical variables store boolean values, which can be either TRUE or FALSE.

```
is_student <- TRUE
has_car <- FALSE
is_student
has_car
```

1.2.4 Integer:

- Integer variables store whole numbers.

```
count <- 10L # L suffix indicates integer type
count
```

1.2.5 Factor:

- Factor variables are used to represent categorical data with levels.

```
gender <- factor(c("Male", "Female", "Male", "Female"))
gender
```

1.2.6 Complex:

- Complex variables store complex numbers with real and imaginary parts.

```
z <- 3 + 2i
z
```

1.3 Variable Assignment

- In R, variables are assigned using the <- operator (though "=" can also be used).

```
z <- 5
z
```

1.4 Checking Variable Types

- We can check the type of a variable using the class() function.

```
class(age)
class(name)
class(is_student)
```

1.5 Operators in R

1. Arithmetic operators
2. Relational operators
3. Logical operators
4. Assignment operators
5. Miscellaneous operators

1.5.1 Arithmetic Operator:

- Arithmetic operators are used to perform basic mathematical operations.

```
a <- 5
b <- 3

add <- a + b
cat("Addition: ", add, "\n")

sub <- a - b
cat("Subtraction: ", sub, "\n")
```

```r
mult <- a * b
cat("Multiplication: ", mult, "\n")


div <- a / b
cat("Division: ", div, "\n")


exp <- a ^ b
cat("Exponentiation: ", exp, "\n")


mod <- a %% b
cat("Modulus: ", mod, "\n")


intdiv <- a %/% b
cat("Integer Division: ", intdiv, "\n")
```

1.5.2 Relational Operators:

- Relational operators compare values and return logical values (TRUE or FALSE).

```r
a <- 5
b <- 3


equal_result <- a == b
cat("Equal to (a == b):", equal_result, "\n")


not_equal_result <- a != b
cat("Not equal to (a != b):", not_equal_result, "\n")


greater_than_result <- a > b
cat("Greater than (a > b):", greater_than_result, "\n")


less_than_result <- a < b
cat("Less than (a < b):", less_than_result, "\n")


greater_than_or_equal_result <- a >= b
cat("Greater than or equal to (a >= b):", greater_than_or_equal_result, "\n")


less_than_or_equal_result <- a <= b
cat("Less than or equal to (a <= b):", less_than_or_equal_result, "\n")
```

1.5.3 Logical Operators:

- Logical operators are used to combine multiple conditions.

```r
x <- TRUE
y <- FALSE


and_result <- x & y
cat("Logical AND (x & y):", and_result, "\n")


or_result <- x | y
cat("Logical OR (x | y):", or_result, "\n")


not_result <- !x
cat("Logical NOT (!x):", not_result, "\n")


xor_result <- xor(x, y)
cat("Logical XOR (xor(x, y)):", xor_result, "\n")


and_multiple_result <- (x & !y) & (x | y)
cat("Logical AND with multiple conditions:", and_multiple_result, "\n")
```

1.5.4 Assignment Operators:

- Assignment operators are used to assign values to variables.

```r
a <- 10
b <- 5

a <- b
cat("After b assigned to a: a =", a, "\n")

my_function <- function(x = 5) {
  return(x)
}

print(my_function())
print(my_function(10))
```

1.5.5 Miscellaneous Operators

```r
df <- data.frame(
  ID = 1:5,
  Name = c("Alice", "Bob", "Charlie", "David", "Eve"),
  Age = c(25, 30, 35, 40, 45)
)

# 1. Colon Operator (:)
# Creates a sequence of numbers from 1 to 10
sequence <- 1:10
cat("Sequence created using colon operator (1:10):\n")
print(sequence)
```

```r
# 2. Membership (%in%)
# Check if elements are in a vector
vector <- c(1, 3, 5, 7, 9)
membership_check <- c(2, 3, 4) %in% vector
cat("Membership check (c(2, 3, 4) %in% vector):\n")
print(membership_check)



# 3. Concatenation (c())
# Combine elements into a vector
combined_vector <- c(1, 2, 3, 4, 5)
cat("Concatenated vector (c(1, 2, 3, 4, 5)):\n")
print(combined_vector)




# 4. Subset using $ (extract a column from a data frame)
name_column <- df$Name
cat("Extracted Name column using $:\n")
print(name_column)




# 5. Subset using [ (extract rows and columns from a data frame)
subset_rows <- df[1:3, ]
cat("Subset of first 3 rows using [:\n")
print(subset_rows)




# 6. Subset using [[ (extract a single element or list element)
age_column <- df[["Age"]]
cat("Extracted Age column using [[:\n")
print(age_column)
```

1.6 Sequence Control

1. Conditional statements
    a. if
    b. if…else
    c. if…else if…else
2. Loops
    a. for
    b. while

c. repeat

3. Control statements

    a. break

    b. next

1.6.1 Conditional Statements

1.6.1.1 if Statement:

- It evaluates a condition and executes a block of code if the condition is TRUE.

```
x <- 10
if(x >5){
  print("x is greater than 5")
}
```

1.6.1.2 if…else:

- It allows you to execute one block of code if the condition is TRUE and another block if it is FALSE.

```
x <- 3
if (x > 5) {
  print("x is greater than 5")
} else {
  print("x is not greater than 5")
}
```

1.6.1.3 if…else if…else Statement:

- It allows you to check multiple conditions sequentially.

```
x <- 7
if (x <5){
  print("x is less than 5")
} else if(x>=5 & x <10){
  print("x is between 5 and 9")
} else {
  print("x is 10 or greater")
}
```

1.6.2 Loops

- Loops are used to repeat a block of code multiple times until a specified condition is met.

1.6.2.1 for:

- It iterates over a sequence (e.g., a vector or a sequence of numbers) and executes a block of code for each element.

```
for (i in 1:5){
  print(i)
}
```

1.6.2.2 while:

- It repeats a block of code as long as a specified condition is TRUE.

```
x <- 1
while(x <= 5){
  print(x)
  x <- x + 1
}
```

1.6.2.3 repeat:

- It repeatedly executes a block of code until a break statement is encountered.

```
x <- 1
repeat {
  print(x)
  x <- x + 1
  if (x >5){
    # Exit the loop when x > 5
    break
  }
}
```

1.6.3 Control Statements

1.6.3.1 break:

- This is used to exit a loop prematurely.

```
for(i in 1:10){
  if (i>5) {
    break # Exit the loop when i >5
  }
  print(i)
}
```

1.6.3.2 next:

- It skips the current iteration of a loop and continues with the next iteration.

```
for(i in 1:5) {
  if(i == 3) {
    next # Skip iteration when i=3
  }
  print(i)
}
```

1.6.3.3 return:

- It is used to exit a function and return a value.

```
my_function <- function(x) {
  if(x <0){
    return("Input is negative")
  } else{
    return("Input is positive")
  }
}
```

```r
result <- my_function(-5)
print(result)
```

2 Part-2

2.1 Matrix

- A matrix is a two-dimensional array that holds elements arranged in rows and columns.
- Matrices are essentially a collection of vectors arranged in a grid.

1. Dimension: Two-dimensional (rows and columns).
2. Type: Homogeneous (all elements must be of the same type).
3. Indexing: Accessed by two indices (row and column).

2.2 Creating Matrices

- Create a matrix using the matrix() function.
- The matrix() function takes a vector of elements and organizes them into a matrix of specified dimensions.

```
# Create a 3x3 matrix with numbers 1 to 9
# By default, matrices are filled column-wise.
m <- matrix(1:9, nrow = 3, ncol = 3)
print(m)
```

```
# Column wise
my_data <- 1:20
A <- matrix(my_data, 4, 5)
```

```
# Creates a 4x5 matrix with numbers 1 to 20. Filled column-wise by default.
my.data <- 1:20
A <- matrix(my.data, 4, 5)
print(A)
```

2.2.1 Total Elements = row*col

```
# Total elements = 3 * 4 = 12
data <- 1:12
matrix_data <- matrix(data, nrow = 3, ncol = 4)
print(matrix_data)
```

2.2.2 Recycle the data

```
# Total elements = 3 * 4 = 12, but data has only 8 elements
# This will recycle the data.
data <- 1:8
matrix_data <- matrix(data, nrow = 3, ncol = 4)
print(matrix_data)
```

2.3 Filling by Row

```
#Create a 3x3 matrix filled by row
m_byrow <- matrix(1:9, nrow = 3, byrow = TRUE)
print(m_byrow)
```

2.4 Accessing Matrix Elements

```
# Single Element: Access a specific element using row and column indices.
m <- matrix(2:13, nrow = 4, ncol = 3)
print(m)
element <- m[2, 3]
print(element)
```

```
# Row wise
A <- matrix(my_data, 4, 5, byrow = TRUE)


# Accesses the element in the 2nd row, 3rd column: 8
a1 <- A[2, 3]
a1

# Accesses the entire 2nd row: 6 7 8 9 10
a2 <- A[2, ]
a2


# Accesses the entire 3rd column: 3 8 13 18
a3 <- A[, 3]
a3
```

2.5 Matrix operations

Matrix operations are fundamental in linear algebra and data manipulation, especially in programming and data science.

- Matrix Addition
- Element-wise Multiplication
- Matrix Multiplication
- Transpose

2.5.1 Matrix Addition:

- It involves adding corresponding elements of two matrices of the same dimensions.

```
# Matrix Addition
m1 <- matrix(1:8, nrow = 4, byrow = TRUE)
m2 <- matrix(8:15, nrow = 4)
result_add <- m1 + m2
print(result_add)
```

Explanation:

- m1 and m2 are matrices of the same size (4x2 in this case).
- Addition is performed element-wise: (m1[1, 1] + m2[1, 1]) for the element in the first row, first column ,and so on.

2.5.2 Element-wise Multiplication

Element-wise Multiplication multiplies corresponding elements of two matrices of the same dimensions.

```r
# Element-wise Multiplication
result_mult <- m1 * m2
print(result_mult)
```

Explanation:

- m1 * m2 multiplies each element of m1 by the corresponding element in m2.

2.5.3 Matrix Multiplication

Matrix Multiplication (also known as matrix product) involves a more complex operation than element-wise multiplication. It requires that the number of columns in the first matrix be equal to the number of rows in the second matrix.

```r
# Matrix Multiplication
m3 <- matrix(1:8, nrow = 4, byrow = TRUE)
m4 <- matrix(8:15, nrow = 2, ncol = 4)
result_mat_mul <- m3 %*% m4
print(result_mat_mul)
```

Explanation:

- %*% performs matrix multiplication.
- For two matrices A (dimensions m X n) and B (dimensions n X p), the resulting matrix C will have dimensions m X p.
- c[i, j] is computed as the sum of the products of the elements of the i-th row of A and the j-th column of B.

2.5.4 Transpose

Transpose of a matrix swaps its rows with columns.

```r
# Transpose
m_transpose <- t(result_mat_mul)
print(m_transpose)
```

Explanation:

- t() function swaps rows and columns of the matrix.

2.5.5 Summary of Matrix Operations:

1. **Addition (`+`)**: Adds corresponding elements of two matrices of the same size.
2. **Element-wise Multiplication (`*`)**: Multiplies corresponding elements of two matrices of the same size.
3. **Matrix Multiplication (`%*%`)**: Multiplies two matrices where the number of columns in the first matrix matches the number of rows in the second matrix.
4. **Transpose (`t()`)**: Swaps rows and columns of a matrix.

2.6 Matrix Functions

2.6.1 `dim()`

Purpose: Returns the dimensions of a matrix.

```r
dimensions <- dim(m_transpose)
print(dimensions)
```

Explanation:

- dim() function returns a vector with two elements: The number of rows and the number of columns of the matrix.
- Example output could be (4, 4) if m_transpose is a 4x4 matrix.

2.6.2 `sum()`

Purpose: Calculates the sum of all elements in the matrix.

```r
total_sum <- sum(m_transpose)
print(total_sum)
```

Explanation:

- sum() computes the total sum of all elements in the matrix.
- If m_transpose is a matrix with elements 1, 2, 3, 4, then sum(m_transpose) would be 10.

2.6.3 `rowSums()`

Purpose: Calculates the sum of elements for each row in the matrix.

```r
r_sums <- rowSums(m_transpose)
print(r_sums)
```

Explanation:

- rowSums() returns a vector where each element is the sum of the elements in the corresponding row of the matrix.

2.6.4 `colSums()`

Purpose: Calculates the sum of elements for each column in the matrix.

```r
c_sums <- colSums(m_transpose)
print(c_sums)
```

Explanation:

- colSums() returns a vector where each element is the sum of the elements in the corresponding column of the matrix.

Purpose: Combines multiple vectors or matrices into a single matrix by stacking them as rows.

```r
r1 <- c("hello", "world", "today")
r2 <- c("mon", "tue", "wed")
r3 <- c(3, 4, 5)  # Mixed data types

m1 <- rbind(r1, r2, r3)
print(m1)
```

Explanation:

- rbind() function combines the vectors r1, r2, and r3 into a matrix, stacking them as rows.
- Note: Since r1 and r2 are character vectors and r3 is numeric, all data is coerced to character type to accommodate mixed types.

2.7 Column Binding (cbind)

Purpose: Combines multiple vectors or matrices into a single matrix by stacking them as columns.

```
c1 <- 1:5
c2 <- -2:-6
m2 <- cbind(c1, c2)
print(m2)
```

Explanation:

- cbind() function combines c1 and c2 into a matrix, placing them as columns.

2.8 Naming and Accessing Elements

2.8.1 Naming Vectors

Purpose: Assign names to elements in a vector and access them by name.

```
v1 <- 5:9
names(v1) <- c("a", "b", "c", "d", "e")
print(v1)
v1["d"]
```

Explanation:

- names(v1) <- c("a", "b", "c", "d", "e") assigns names to the elements of the vector v1.
- We can access elements using these names.
- For instance, v1["d"] retrieves the value associated with the name "d".

2.8.2 Removing Names from a Vector

```
names(v1) <- NULL
```

Explanation:

- names(v1) <- NULL removes names from the vector v1.

2.8.3 Matrix Creation and Naming

Purpose: Create a matrix and assign row and column names.

```
# A vector with 3 elements: "a", "B", and "hello"
v1 <- c("a", "B", "hello")

# Repeats the elements of v1 three times:
# "a", "B", "hello", "a", "B", "hello", "a", "B", "hello"
v2 <- rep(v1, 3)

# Repeats each element of v1 three times in sequence:
#"a", "a", "a", "B", "B", "B", "hello", "hello", "hello"
v3 <- rep(v1, each = 3)


mat <- matrix(v3, nrow = 3, ncol = 3)
rownames(mat) <- c("how", "are", "you")
colnames(mat) <- c("apple", "banana", "kiwi")
print(mat)
```

Explanation:

- matrix(v3, nrow = 3, ncol = 3) creates a 3x3 matrix using v3.
- rownames(mat) and colnames(mat) assign names to rows and columns respectively.

2.8.4 Accessing Elements Using Names:

```
mat["how", "kiwi"]
```

Explanation:

- Access elements by specifying row and column names.
- For instance, mat["how", "kiwi"] retrieves the element at the intersection of the row "how" and column "kiwi".

2.8.5 Removing Row and Column Names

```
rownames(mat) <- NULL
colnames(mat) <- NULL
```

Explanation:

- rownames(mat) <- NULL and colnames(mat) <- NULL
- remove row and column names from the matrix.

2.8.6 Summary

- **Matrix Functions:** `dim()`, `sum()`, `rowSums()`, and `colSums()` help in analyzing matrix dimensions and summarizing data.
- **Row and Column Binding:** `rbind()` and `cbind()` are used to combine matrices or vectors by rows or columns.
- **Naming and Accessing Elements:** Assign and access names in vectors and matrices to make data manipulation more intuitive.

2.9 Vector

In R, a vector is a fundamental data structure used to store elements of the same type. Vectors are essential for handling and manipulating data in R because they allow for efficient and convenient operations on data collections.

2.10 Key Characteristics of Vectors in R

- **Homogeneous Elements:** All elements in a vector must be of the same data type. For example, a numeric vector can only contain numbers, a character vector can only contain strings, and so forth.
- **One-Dimensional:** Vectors are one-dimensional arrays, meaning they only have a single axis. They can be thought of as a list or sequence of elements.
- **Indexed:** Elements in a vector are accessed via indices, which start from 1 in R. For example, `v[1]` accesses the first element of the vector `v`.

2.11 Different Methods for Vector Creation

2.11.1 Using `c()` Function

Purpose: The `c()` function combines elements into a vector.

a. Numeric Vector:

```
nums <- c(1, 2, 3, 4, 5)
nums
```

Explanation:

- Creates a numeric vector with elements 1 through 5.

b. Character Vector:

```
chars <- c("apple", "banana", "orange")
chars
```

Explanation:

- Creates a character vector with three fruit names.

c. Logical Vector:

```
logic <- c(TRUE, FALSE, TRUE)
logic
```

Explanation:

- Creates a logical vector with boolean values.

d. Mixed Type:

```
vec <- c("a", 2, 3, "b")
vec
```

Explanation:

- All elements are coerced to character type: c("a", "2", "3", "b").

2.11.2 Other Ways to Create Vectors: `seq()` and `rep()`

a. Sequence of Numbers:

```
vec0 <- 6:12
num_seq <- seq(from = 1, to = 10, by = 2)
vec1 <- seq(1, 15)
vec2 <- seq(1, 15, 2)
num_seq
```

Explanation:

- 6:12 creates a sequence from 6 to 12.
- seq(from = 1, to = 10, by = 2) generates a sequence from 1 to 10 with a step of 2: 1, 3, 5, 7, 9.
- seq(1, 15) generates a sequence from 1 to 15 with a default step of 1.
- seq(1, 15, 2) generates a sequence from 1 to 15 with a step of 2.

b. Repeating Elements:

```r
nums_rep <- rep(1:3, times = 2)
vec3 <- rep(2, 5)
vec4 <- rep("hello", 3)
vec4
```

Explanation:

- rep(1:3, times = 2) repeats the sequence 1, 2, 3 twice: 1, 2, 3, 1, 2, 3.
- rep(2, 5) repeats the number 2 five times: 2, 2, 2, 2, 2.
- rep("hello", 3) repeats the string "hello" three times: "hello", "hello", "hello".

c. Mixed Elements:

```r
v2 <- c("h", "ell", "o")
v2 <- c("h", "ell", "o", 7)
v2
```

Explanation:

- v2 initially contains "h", "ell", "o".
- After adding 7, all elements are coerced to character: c("h", "ell", "o", "7").

```r
vec5 <- c(7, 120)
vec6 <- rep(vec5, 2)
vec5
vec6
```

Explanation:

- vec5 is a vector with elements 7 and 120.
- rep(vec5, 2) repeats the vec5 vector twice: 7, 120, 7, 120.

2.12 Vector Indexing and Subsetting

2.12.1 Indexing with [] Bracket:

Explanation:

- Access or remove elements using indexing.
- Negative indices exclude specified elements.

```r
w <- c(2, 3, 4, 5, 6, 7, 81, 21)
 # First element: 2
w[1]
# Second element: 3
w[2]
# Fifth element: 6
w[5]
# All elements except the first one: 3, 4, 5, 6, 7, 81, 21
w[-1]
# All elements except the third one: 2, 3, 5, 6, 7, 81, 21
w[-3]
# Elements from the first to the third: 2, 3, 4
w[1:3]
 # Elements from the fifth to the seventh: 6, 7, 81
w[5:7]
```

```r
# All elements except the first to third: 5, 6, 7, 81, 21
w[-1:-3]
```

2.12.2 Subset Example:

```r
nums <- c(10, 20, 30, 40, 50)
# Access first element: 10
nums[1]
# Access elements 3 to 5: 30, 40, 50
nums[3:5]
# Access elements 1 and 4: 10, 40
nums[c(1, 4)]
```

```r
vec7 <- c(11, 23, 55, 99, 100, 500, 21, 26)
# Number of elements: 8
length(vec7)
# Access the last element: 26
vec7[length(vec7)]
# View without the first element: 23, 55, 99, 100, 500, 21, 26
vec7[-1]
# Subset from 3rd to 7th: 55, 99, 100, 500, 21
vec7[3:7]
# Up to the second last element: 55, 99, 100, 500, 21
vec7[3:(length(vec7) - 1)]
# Remove first three elements: 99, 100, 500, 21, 26
vec7[-(1:3)]
```

2.13 Vector Operations

2.13.1 Element-wise Operations:

```r
vec8 <- c(12, 34, 56, 77, 78, 86, 223, 100, 45, 10)
vec9 <- c(54, 32, 87, 21, 99)
# Element-wise addition (recycling rule applies)
vec10 <- vec8 + vec9
# Element-wise division
vec11 <- vec8 / vec9
```

Explanation:

- Element-wise operations are performed with recycling if vectors are of different lengths.

2.13.2 Recycling Rule Example:

```r
vec1 <- c(1, 2, 3)
vec2 <- c(4, 5)

vec_sum <- vec1 + vec2
# vec2 is recycled to match length of vec1: [4, 5, 4]
# Result: [5, 7, 7]

vec1 <- c(1, 2, 3)
vec2 <- c(4, 5, 6)
```

```r
# Element-wise addition: [5, 7, 9]
vec_sum <- vec1 + vec2
# Scalar multiplication: [2, 4, 6]
vec_mul <- vec1 * 2
# Logical comparison: [FALSE, FALSE, TRUE]
vec_logical <- vec1 > 2
```

Explanation:

- `vec_sum` performs element-wise addition.
- `vec_mul` multiplies each element by 2.
- `vec_logical` creates a logical vector based on comparison.

```r
vec <- c(1, 2, 3)
# Length of vector: 3
length(vec)
# Check if numeric: TRUE
is.numeric(vec)
# Check if character: FALSE
is.character(vec)
# Check if double: TRUE
is.double(vec)
# Check if integer: FALSE
is.integer(vec)
# Type of vector: "double"
typeof(vec)
```

2.13.3 Summary

- **`c()`**: Combines values into a vector.
- **`seq()`**: Generates sequences of numbers.
- **`rep()`**: Repeats elements of vectors.
- **`is.vector()`**: Checks if an object is a vector.
- **`typeof()`**: Determines the type of an object.

2.14 Application Level

2.14.1 CO2 Data Analysis

```r
co2 <- c(369.55, 371.14, 373.28, 375.80, 377.52, 379.80,
         381.90, 383.79, 385.60, 387.43, 389.90, 391.65,
         393.85, 396.52, 398.65,400.83, 404.21)
year <- c(2000:2016)

# Compute the mean of CO2 values
mean(co2)
# Compute the standard deviation of CO2 values
sd(co2)
# Plot CO2 values against years
plot(year, co2)
```

2.14.2 Basketball Players Data Analysis

The data is form based on the data available at https://data.world/datadavis/nba-salaries

Instructions for this dataset:

Once executed the commands the following objects will be created:

Matrices:

1. FieldGoalAttempts
2. FieldGoals
3. Games
4. MinutesPlayed
5. Salary
6. Points
7. Players
8. Seasons

```
# Comments:
# Seasons are labeled based on the first year in the season
# E.g. the 2012-2013 season is preseneted as simply 2012

#Seasons
Seasons <- c("2005","2006","2007","2008","2009","2010","2011","2012","2013","2014")


# Players
Players <- c("KobeBryant","JoeJohnson","LeBronJames","CarmeloAnthony",
             "DwightHoward","ChrisBosh","ChrisPaul","KevinDurant",
             "DerrickRose","DwayneWade")




# 1. Salaries
KobeBryant_Salary <- c(15946875,17718750,19490625,21262500,
                       23034375,24806250,25244493,27849149,30453805,23500000)
JoeJohnson_Salary <- c(12000000,12744189,13488377,14232567,
                       14976754,16324500,18038573,19752645,21466718,23180790)
LeBronJames_Salary <- c(4621800,5828090,13041250,14410581,15779912
                        ,1450000,16022500,17545000,19067500,20644400)
CarmeloAnthony_Salary <- c(3713640,4694041,13041250,14410581,
                           15779912,17149243,18518574,19450000,22407474,22458000)
DwightHoward_Salary <- c(4493160,4806720,6061274,13758000,
                         15202590,16647180,18091770,19536360,20513178,21436271)
ChrisBosh_Salary <- c(3348000,4235220,12455000,14410581,15779912,
                      14500000,16022500,17545000,19067500,20644400)
ChrisPaul_Salary <- c(3144240,3380160,3615960,4574189,13520500,
                      14940153,16359805,17779458,18668431,20068563)
KevinDurant_Salary <- c(0,0,4171200,4484040,4796880,6053663,
                        15506632,16669630,17832627,18995624)
DerrickRose_Salary <- c(0,0,0,4822800,5184480,5546160,
                        6993708,16402500,17632688,18862875)
DwayneWade_Salary <- c(3031920,3841443,13041250,14410581,15779912,
                       14200000,15691000,17182000,18673000,15000000)

# Matrix-1
```

```r
# Step 1: Create the matrix Salary using rbind()
Salary <- rbind(KobeBryant_Salary, JoeJohnson_Salary, LeBronJames_Salary,
                CarmeloAnthony_Salary, DwightHoward_Salary, ChrisBosh_Salary,
                ChrisPaul_Salary, KevinDurant_Salary, DerrickRose_Salary, DwayneWade_Salary)

# Step 2: Remove individual player salary vectors from memory using rm()
# Purpose of rm(): This command removes the individual salary vectors from the environment,
# freeing up memory, since they have already been combined into the matrix Salary.
rm(KobeBryant_Salary, JoeJohnson_Salary, CarmeloAnthony_Salary,
   DwightHoward_Salary, ChrisBosh_Salary, LeBronJames_Salary,
   ChrisPaul_Salary, DerrickRose_Salary, DwayneWade_Salary, KevinDurant_Salary)

# Step 3: Assign column and row names to the matrix
colnames(Salary) <- Seasons
rownames(Salary) <- Players
print(Salary)





# 2. Games
KobeBryant_G <- c(80,77,82,82,73,82,58,78,6,35)
JoeJohnson_G <- c(82,57,82,79,76,72,60,72,79,80)
LeBronJames_G <- c(79,78,75,81,76,79,62,76,77,69)
CarmeloAnthony_G <- c(80,65,77,66,69,77,55,67,77,40)
DwightHoward_G <- c(82,82,82,79,82,78,54,76,71,41)
ChrisBosh_G <- c(70,69,67,77,70,77,57,74,79,44)
ChrisPaul_G <- c(78,64,80,78,45,80,60,70,62,82)
KevinDurant_G <- c(35,35,80,74,82,78,66,81,81,27)
DerrickRose_G <- c(40,40,40,81,78,81,39,0,10,51)
DwayneWade_G <- c(75,51,51,79,77,76,49,69,54,62)

# Matrix-2
Games <- rbind(KobeBryant_G, JoeJohnson_G, LeBronJames_G,
               CarmeloAnthony_G, DwightHoward_G, ChrisBosh_G,
               ChrisPaul_G, KevinDurant_G, DerrickRose_G, DwayneWade_G)

rm(KobeBryant_G, JoeJohnson_G, CarmeloAnthony_G,
   DwightHoward_G, ChrisBosh_G, LeBronJames_G, ChrisPaul_G,
   DerrickRose_G, DwayneWade_G, KevinDurant_G)

colnames(Games) <- Seasons





# 3. Minutes Played
KobeBryant_MP <- c(3277,3140,3192,2960,2835,2779,2232,3013,177,1207)
JoeJohnson_MP <- c(3340,2359,3343,3124,2886,2554,2127,2642,2575,2791)
LeBronJames_MP <- c(3361,3190,3027,3054,2966,3063,2326,2877,2902,2493)
CarmeloAnthony_MP <- c(2941,2486,2806,2277,2634,2751,1876,2482,2982,1428)
```

```r
DwightHoward_MP <- c(3021,3023,3088,2821,2843,2935,2070,2722,2396,1223)
ChrisBosh_MP <- c(2751,2658,2425,2928,2526,2795,2007,2454,2531,1556)
ChrisPaul_MP <- c(2808,2353,3006,3002,1712,2880,2181,2335,2171,2857)
KevinDurant_MP <- c(1255,1255,2768,2885,3239,3038,2546,3119,3122,913)
DerrickRose_MP <- c(1168,1168,1168,3000,2871,3026,1375,0,311,1530)
DwayneWade_MP <- c(2892,1931,1954,3048,2792,2823,1625,2391,1775,1971)

# Matrix-3
MinutesPlayed <- rbind(KobeBryant_MP, JoeJohnson_MP,
                       LeBronJames_MP, CarmeloAnthony_MP,
                       DwightHoward_MP, ChrisBosh_MP, ChrisPaul_MP,
                       KevinDurant_MP, DerrickRose_MP, DwayneWade_MP)

rm(KobeBryant_MP, JoeJohnson_MP, CarmeloAnthony_MP,
   DwightHoward_MP, ChrisBosh_MP, LeBronJames_MP, ChrisPaul_MP,
   DerrickRose_MP, DwayneWade_MP, KevinDurant_MP)

colnames(MinutesPlayed) <- Seasons
rownames(MinutesPlayed) <- Players




# 4. Field Goals
KobeBryant_FG <- c(978,813,775,800,716,740,574,738,31,266)
JoeJohnson_FG <- c(632,536,647,620,635,514,423,445,462,446)
LeBronJames_FG <- c(875,772,794,789,768,758,621,765,767,624)
CarmeloAnthony_FG <- c(756,691,728,535,688,684,441,669,743,358)
DwightHoward_FG <- c(468,526,583,560,510,619,416,470,473,251)
ChrisBosh_FG <- c(549,543,507,615,600,524,393,485,492,343)
ChrisPaul_FG <- c(407,381,630,631,314,430,425,412,406,568)
KevinDurant_FG <- c(306,306,587,661,794,711,643,731,849,238)
DerrickRose_FG <- c(208,208,208,574,672,711,302,0,58,338)
DwayneWade_FG <- c(699,472,439,854,719,692,416,569,415,509)

# Matrix-4
FieldGoals <- rbind(KobeBryant_FG, JoeJohnson_FG, LeBronJames_FG,
                    CarmeloAnthony_FG, DwightHoward_FG, ChrisBosh_FG,
                    ChrisPaul_FG, KevinDurant_FG, DerrickRose_FG, DwayneWade_FG)

rm(KobeBryant_FG, JoeJohnson_FG, LeBronJames_FG,
   CarmeloAnthony_FG, DwightHoward_FG, ChrisBosh_FG,
   ChrisPaul_FG, KevinDurant_FG, DerrickRose_FG, DwayneWade_FG)

colnames(FieldGoals) <- Seasons
rownames(FieldGoals) <- Players




# 5. Field Goal Attempts
KobeBryant_FGA <- c(2173,1757,1690,1712,1569,1639,1336,1595,73,713)
JoeJohnson_FGA <- c(1395,1139,1497,1420,1386,1161,931,1052,1018,1025)
```

```r
LeBronJames_FGA <- c(1823,1621,1642,1613,1528,1485,1169,1354,1353,1279)
CarmeloAnthony_FGA <- c(1572,1453,1481,1207,1502,1503,1025,1489,1643,806)
DwightHoward_FGA <- c(881,873,974,979,834,1044,726,813,800,423)
ChrisBosh_FGA <- c(1087,1094,1027,1263,1158,1056,807,907,953,745)
ChrisPaul_FGA <- c(947,871,1291,1255,637,928,890,856,870,1170)
KevinDurant_FGA <- c(647,647,1366,1390,1668,1538,1297,1433,1688,467)
DerrickRose_FGA <- c(436,436,436,1208,1373,1597,695,0,164,835)
DwayneWade_FGA <- c(1413,962,937,1739,1511,1384,837,1093,761,1084)

# Matrix-5
FieldGoalAttempts <- rbind(KobeBryant_FGA, JoeJohnson_FGA,
                           LeBronJames_FGA, CarmeloAnthony_FGA,
                           DwightHoward_FGA, ChrisBosh_FGA, ChrisPaul_FGA,
                           KevinDurant_FGA, DerrickRose_FGA, DwayneWade_FGA)

rm(KobeBryant_FGA, JoeJohnson_FGA, LeBronJames_FGA,
   CarmeloAnthony_FGA, DwightHoward_FGA, ChrisBosh_FGA,
   ChrisPaul_FGA, KevinDurant_FGA, DerrickRose_FGA, DwayneWade_FGA)

colnames(FieldGoalAttempts) <- Seasons
rownames(FieldGoalAttempts) <- Players




# 6.Points
KobeBryant_PTS <- c(2832,2430,2323,2201,1970,2078,1616,2133,83,782)
JoeJohnson_PTS <- c(1653,1426,1779,1688,1619,1312,1129,1170,1245,1154)
LeBronJames_PTS <- c(2478,2132,2250,2304,2258,2111,1683,2036,2089,1743)
CarmeloAnthony_PTS <- c(2122,1881,1978,1504,1943,1970,1245,1920,2112,966)
DwightHoward_PTS <- c(1292,1443,1695,1624,1503,1784,1113,1296,1297,646)
ChrisBosh_PTS <- c(1572,1561,1496,1746,1678,1438,1025,1232,1281,928)
ChrisPaul_PTS <- c(1258,1104,1684,1781,841,1268,1189,1186,1185,1564)
KevinDurant_PTS <- c(903,903,1624,1871,2472,2161,1850,2280,2593,686)
DerrickRose_PTS <- c(597,597,597,1361,1619,2026,852,0,159,904)
DwayneWade_PTS <- c(2040,1397,1254,2386,2045,1941,1082,1463,1028,1331)

# Matrix-6
Points <- rbind(KobeBryant_PTS, JoeJohnson_PTS, LeBronJames_PTS,
                CarmeloAnthony_PTS, DwightHoward_PTS, ChrisBosh_PTS,
                ChrisPaul_PTS, KevinDurant_PTS, DerrickRose_PTS, DwayneWade_PTS)

rm(KobeBryant_PTS, JoeJohnson_PTS, LeBronJames_PTS,
   CarmeloAnthony_PTS, DwightHoward_PTS, ChrisBosh_PTS,
   ChrisPaul_PTS, KevinDurant_PTS, DerrickRose_PTS, DwayneWade_PTS)

colnames(Points) <- Seasons
rownames(Points) <- Players
```

2.15 Questions

2.15.1 How many games did ChrisPaul play in 2011?

```
no_of_games_CP = Games["ChrisPaul_G", "2011"]
paste(no_of_games_CP, "no. of games ChrisPaul play in 2011")
```

Explanation:

- `Games` is a matrix where the rows are players and the columns are seasons. To get the number of games Chris Paul played in 2011, you access the element in the row labeled "ChrisPaul" and the column labeled "2011".
- `paste()` combines the result with a string to provide a readable output.

2.15.2 What are the field goals per game for each player?

```
    dim(Games)
    dim(FieldGoals) # check dimension
    FieldGoals_Per_Game = FieldGoals / Games
    print(FieldGoals_Per_Game)
    round(FieldGoals_Per_Game)
    round(FieldGoals_Per_Game, 1) # with one decimal
```

Explanation:

- `FieldGoals` and `Games` are matrices of the same dimensions.
- The division `FieldGoals / Games` computes the field goals per game for each player by dividing the number of field goals by the number of games played.
- `round()` is used to round the results to a specified number of decimal places for better readability.

2.15.3 How many minutes did each player play per game?

```
Min_ply_per_game = MinutesPlayed / Games
print(Min_ply_per_game)
```

Explanation:

- `MinutesPlayed` and `Games` are matrices with the same dimensions. Dividing `MinutesPlayed` by `Games` gives the average number of minutes played per game for each player.

2.15.4 How much is per minute worth for each player?

```
Per_min_worth = Salary / MinutesPlayed
print(Per_min_worth)
```

Explanation:

- `Salary` and `MinutesPlayed` are matrices. Dividing `Salary` by `MinutesPlayed` computes the worth per minute for each player, which tells you how much salary is earned per minute of play.

2.15.5 How accurate is each player?

```
Accurate <- FieldGoals / FieldGoalAttempts
print(Accurate)
```

Explanation:

- `FieldGoals` and `FieldGoalAttempts` are matrices. Dividing `FieldGoals` by `FieldGoalAttempts` calculates the shooting accuracy (or field goal percentage) for each player, which measures how often a player makes a field goal attempt.

2.15.6 Who is good at 3-pointers?

```
# Calculate total points for each player
r_sum <- matrix(rowSums(Points), nrow = length(Players), dimnames = list(Players, "Total Points"))
print(r_sum)

# Sort players by total points and print the sorted results
sorted_r_sum <- sort(r_sum[, "Total Points"], decreasing = TRUE)
print(sorted_r_sum)

# Print the highest ranked player
highest_ranked_player <- names(sorted_r_sum)[1]
highest_ranked_points <- sorted_r_sum[1]
cat("The highest ranked player based on total points is:",
    highest_ranked_player, "with",
    highest_ranked_points, "total points.\n")
```

- `rowSums(Points)`: This calculates the sum of points scored by each player across all seasons.
- `matrix(..., nrow = length(Players), dimnames = list(Players, "Total Points"))`: This converts the total points into a matrix. Each row corresponds to a player, and the column is named "Total Points".
- `print(r_sum)`: This prints the matrix `r_sum` to the console, showing each player and their total points.
- `names(sorted_r_sum)[1]`: This extracts the name of the player with the highest total points (the first element in `sorted_r_sum`).
- `sorted_r_sum[1]`: This extracts the highest total points value.
- `cat(...)`: This concatenates and prints a message to the console stating who the highest-ranked player is and their total points.

3 Part-3

3.1 Read the Data from a CSV File

```r
mydata <- read.csv("datasets/DemographicsData.csv")
```

- **`read.csv("datasets/DemographicsData.csv")`**: Reads the CSV file into a data frame named **`mydata`**.

```r
mydata
```

- **`mydata`**: Displays the entire data frame.

3.2 Explore and Understand the Data

```r
# Number of rows in the data frame
nrow(mydata)
# Number of columns in the data frame
ncol(mydata)
# Dimensions of the data frame (rows, columns)
dim(mydata)
# Number of rows
dim(mydata)[1]
```

- **`nrow(mydata)`**: Returns the number of rows.
- **`ncol(mydata)`**: Returns the number of columns.
- **`dim(mydata)`**: Returns a vector with the number of rows and columns.
- **`dim(mydata)[1]`**: Extracts the number of rows from the dimensions vector.

```r
# View the first 6 rows of the data frame
head(mydata)
# View the last 6 rows of the data frame
tail(mydata)
```

- **`head(mydata)`**: Shows the first 6 rows.
- **`tail(mydata)`**: Shows the last 6 rows.

```r
# Structure of the data frame, showing column types and a preview
str(mydata)
# Summary statistics for each column
summary(mydata)
```

- **`str(mydata)`**: Provides the structure of the data frame, including data types and a preview of the data.
- **`summary(mydata)`**: Provides summary statistics for each column, such as min, max, mean, median, etc.

3.3 Subsetting the Data Frame

```r
# Extract the first row as a data frame
mydata[1, ]
# Check if the result is still a data frame
is.data.frame(mydata[1, ])
```

- **`mydata[1, ]`**: Extracts the first row of **`mydata`** as a data frame.
- **`is.data.frame(mydata[1, ])`**: Checks if the extracted row is a data frame (it is).

```r
# Extract the first column as a vector
mydata[, 1]
```

```r
# Check if the result is a data frame (it's not)
is.data.frame(mydata[, 1])
# Confirm the result is a vector
is.vector(mydata[, 1])
# Extract the first column as a data frame (TRUE)
is.data.frame(mydata[, 1, drop = F])
```

- **`mydata[, 1]`**: Extracts the first column as a vector.
- **`is.data.frame(mydata[, 1])`**: Checks if the extracted column is a data frame (it's not; it's a vector).
- **`is.vector(mydata[, 1])`**: Confirms that the result is a vector.
- **`mydata[, 1, drop = F]`**: Extracts the first column as a data frame (using `drop = FALSE` to preserve the data frame structure).
- **Without `drop = FALSE`**: If you simply use `mydata[, 1]`, you get a vector of the first column if `mydata` has only one column selected.
- **With `drop = FALSE`**: Using `mydata[, 1, drop = FALSE]` ensures that the result is a data frame with one column, even if only a single column is extracted.

3.4 Accessing Columns Using $

```r
# Get 'Country.Name' column
country_name = mydata$Country.Name
head(country_name, 10)
```

```r
# First 10 rows
mydata[1:10,]
```

```r
# Extract the 5th and 99th rows from the dataframe
mydata[c(5, 99), ]
```

```r
# Extract the 3rd and 55th rows from the dataframe
mydata[c(3, 55), ]
```

Explanation:

- `mydata$Country.Name`: Extracts the 'Country.Name' column as a vector.
- `mydata[1:10, ]`: Displays the first 10 rows of the dataframe.
- `mydata[c(5, 99), ]`: Shows rows 5 and 99.
- `mydata[c(3, 55), ]`: Shows rows 3 and 55.

3.5 Create an Additional Attribute (Add a Column)

```r
# Add 'dummy' column as product of 'Birth.rate' and 'Internet.users'
mydata$dummy <- mydata$Birth.rate * mydata$Internet.users
head(mydata, 5)
```

```r
# Remove 'dummy' column
mydata$dummy <- NULL
```

Explanation:

- `mydata$dummy <- mydata$Birth.rate * mydata$Internet.users`: Adds a new column 'dummy' which is the product of 'Birth.rate' and 'Internet.users'.
- `mydata$dummy <- NULL`: Removes the 'dummy' column from the dataframe.

3.6 Filter the Data

```
# Filter rows where 'Internet.users' < 2
myfilter1 <- mydata$Internet.users < 2
```

```
# Show filtered rows
mydata[myfilter1, ]
```

```
# Count filtered rows
nrow(mydata[myfilter1, ])
```

```
# Filter rows where 'Internet.users' < 4 and 'Birth.rate' > 40
myfilter2 <- mydata$Internet.users < 4 & mydata$Birth.rate > 40

# Show filtered rows
mydata[myfilter2, ]
```

```
# Get row for 'New Zealand'
myfilter3 <- mydata$Country.Name == "New Zealand"

# Show row for 'New Zealand'
mydata[myfilter3, ]
```

```
# Another way to get row for 'New Zealand'
mydata[mydata$Country.Name == "New Zealand", ]
```

Explanation:

- `mydata$Internet.users < 2`: Creates a logical vector for rows where 'Internet.users' is less than 2.
- `mydata[myfilter1, ]`: Shows rows where the condition is true.
- `nrow(mydata[myfilter1, ])`: Counts how many rows meet the condition.
- `mydata$Internet.users < 4 & mydata$Birth.rate > 40`: Creates a logical vector for rows where 'Internet.users' is less than 4 and 'Birth.rate' is greater than 40.
- `mydata[myfilter2, ]`: Shows rows where both conditions are true.
- `mydata$Country.Name == "New Zealand"`: Creates a logical vector to filter rows where 'Country.Name' is "New Zealand".
- `mydata[myfilter3, ]`: Shows the row for "New Zealand".
- `mydata[mydata$Country.Name == "New Zealand", ]`: Another way to filter rows for "New Zealand".

3.7 Creating a New Attribute Based on Conditions

```
# Add 'InternetLevel' column with default "Low"
mydata$InternetLevel <- "Low"
```

```
# Set "High" for 'Internet.users' >= 70
mydata[mydata$Internet.users >= 70, "InternetLevel"] <- "High"
```

```r
# Set "Medium" for 'Internet.users' between 40 and 69
mydata[mydata$Internet.users < 70 & mydata$Internet.users >= 40, "InternetLevel"] <- "Medium"
```

```r
# Set "Low" for 'Internet.users' < 40
mydata[mydata$Internet.users < 40, "InternetLevel"] <- "Low"
```

```r
head(mydata,5)
```

Explanation:

- `mydata$InternetLevel <- "Low"`: Adds a new column 'InternetLevel' with the default value "Low".
- `mydata[mydata$Internet.users >= 70, "InternetLevel"] <- "High"`: Sets 'InternetLevel' to "High" where 'Internet.users' is 70 or more.
- `mydata[mydata$Internet.users < 70 & mydata$Internet.users >= 40, "InternetLevel"] <- "Medium"`: Sets 'InternetLevel' to "Medium" where 'Internet.users' is between 40 and 69.
- `mydata[mydata$Internet.users < 40, "InternetLevel"] <- "Low"`: Ensures 'InternetLevel' is "Low" where 'Internet.users' is less than 40.

3.8 Quick Plotting with qplot

```r
# Load ggplot2
library(ggplot2)

# Load data again
mydata <- read.csv("datasets/DemographicsData.csv")

# Histogram of 'Internet.users'
qplot(data = mydata, x = Internet.users)
```

```r
# Scatter plot of 'Internet.users' vs 'Birth.rate'
qplot(data = mydata, x = Internet.users, y = Birth.rate)
```

```r
# Scatter plot with size
qplot(data = mydata, x = Internet.users, y = Birth.rate, size = I(5))
```

```r
# Scatter plot with color
qplot(data = mydata, x = Internet.users, y = Birth.rate, size = I(5), color = I("brown"))
```

```r
# Scatter plot with shape
qplot(data = mydata, x = Internet.users, y = Birth.rate,
      size = I(5), color = I("green"), pch = I(17))
```

```r
# Scatter plot with transparency
qplot(data = mydata, x = Internet.users, y = Birth.rate,
      size = I(5), color = I("brown"),
      pch = I(17), alpha = I(0.5))
```

```r
# Scatter plot with color by 'Income.Group'
qplot(data = mydata, x = Internet.users, y = Birth.rate,
      size = I(5), color = Income.Group,
      pch = I(19), alpha = I(0.5))
```

Explanation:

- `library(ggplot2)`: Loads the ggplot2 library for plotting.

- `?qplot()`: Shows help for the `qplot` function.
- `mydata <- read.csv("DemographicsData.csv")`: Reloads the data.
- `qplot(data = mydata, x = Internet.users)`: Creates a histogram of the 'Internet.users' variable.
- `qplot(data = mydata, x = Internet.users, y = Birth.rate)`: Creates a scatter plot of 'Internet.users' versus 'Birth.rate'.
- `qplot(data = mydata, x = Internet.users, y = Birth.rate, size = I(5))`: Adjusts the size of the plot points.
- `qplot(data = mydata, x = Internet.users, y = Birth.rate, size = I(5), color = I("brown"))`: Changes the color of the points.
- `qplot(data = mydata, x = Internet.users, y = Birth.rate, size = I(5), color = I("green"), pch = I(17))`: Changes the shape of the points.
- `qplot(data = mydata, x = Internet.users, y = Birth.rate, size = I(5), color = I("brown"), pch = I(17), alpha = I(0.5))`: Adjusts the transparency of the points.
- `qplot(data = mydata, x = Internet.users, y = Birth.rate, size = I(5), color = Income.Group, pch = I(19), alpha = I(0.5))`: Colors points based on 'Income.Group' and adjusts transparency.

4 Part-4

4.1 Reading the CSV Files

```r
mydf1 <- read.csv("datasets/DemographicsData.csv")
mydf2 <- read.csv("datasets/CountryRegion.csv")
```

- **`read.csv("DemographicsData.csv")`**: Reads the CSV file named `"DemographicsData.csv"` into a dataframe `mydf1`.
- **`read.csv("CountryRegion.csv")`**: Reads the CSV file named `"CountryRegion.csv"` into a dataframe `mydf2`.

4.2 Exploring the Dataframes

```r
# Check column names of the first dataframe
colnames(mydf1)
```

```r
# Check column names of the second dataframe
colnames(mydf2)
```

```r
# Merge the two dataframes by matching 'Country.Code'
# in mydf1 with 'Codes_2021_Dataset' in mydf2
mymerg <- merge(mydf1, mydf2, by.x = "Country.Code", by.y = "Codes_2021_Dataset")
head(mymerg, 5)
```

```r
# Remove the unnecessary column 'Countries_2021_Dataset' from the merged dataframe
mymerg$Countries_2021_Dataset <- NULL
```

```r
# Save the merged dataframe to a new CSV file without row names
write.csv(mymerg, "merged.csv", row.names = FALSE)
```

4.2.1 Explanation:

- **`mydf1 <- read.csv("DemographicsData.csv")`**: Loads the first dataset from the CSV file into the dataframe `mydf1`.
- **`mydf2 <- read.csv("CountryRegion.csv")`**: Loads the second dataset from the CSV file into the dataframe `mydf2`.
- **`colnames(mydf1)`**: Displays the column names of `mydf1`.
- **`colnames(mydf2)`**: Displays the column names of `mydf2`.
- **`mymerg <- merge(mydf1, mydf2, by.x = "Country.Code", by.y = "Codes_2021_Dataset")`**: Merges `mydf1` and `mydf2` on the specified columns.
- **`mymerg$Countries_2021_Dataset <- NULL`**: Deletes the column `Countries_2021_Dataset` from `mymerg`.
- **`write.csv(mymerg, "merged.csv", row.names = FALSE)`**: Writes the cleaned merged dataframe to a new CSV file called `"merged.csv"` without including row numbers.

4.3 Load Data

```r
# Load the movie ratings data from a CSV file
mymov <- read.csv("datasets/MovieRatings.csv")
```

```r
# Check column names of the dataframe
colnames(mymov)
```

Explanation:

- **`read.csv("MovieRatings.csv")`**: Reads the movie ratings data from a CSV file into a dataframe called `mymov`.
- **`colnames(mymov)`**: Displays the names of the columns in the dataframe.

4.4 Data Preprocessing

```r
# Rename columns for clarity
colnames(mymov) <- c("Film", "Genre", "CRating", "ARating", "BudMils", "Year")

# Verify new column names
colnames(mymov)
```

```r
# Get a summary of the data
summary(mymov)
```

```r
# Get the structure of the data
str(mymov)
```

```r
# Convert 'Genre' column to a factor (categorical variable)
mymov$Genre <- as.factor(mymov$Genre)
```

```r
# Verify the structure again to ensure 'Genre' is a factor
str(mymov)
```

Explanation:

- **`colnames(mymov) <- c(...)`**: Renames the columns of the dataframe for better readability.
- **`summary(mymov)`**: Provides a statistical summary of each column in the dataframe.
- **`str(mymov)`**: Shows the structure of the dataframe, including data types and sample data.
- **`as.factor(mymov$Genre)`**: Converts the `Genre` column to a factor, useful for categorical data analysis

4.5 Data Visualization with ggplot2

```r
# Load the ggplot2 package, which is used for creating plots
library(ggplot2)
```

- **Loads ggplot2 package** for data visualization.
- **Required for** creating plots in R.

```r
# Create a basic scatter plot with CRating on the x-axis and ARating on the y-axis
ggplot(data=mymov, aes(x=CRating, y=ARating)) + geom_point()
```

Creates a scatter plot:

- `CRating` on the x-axis.
- `ARating` on the y-axis.
- **Points represent movies** in the plot.

```
# Create a scatter plot where points are colored by Genre
ggplot(data=mymov, aes(x=CRating, y=ARating, colour=Genre)) + geom_point()
```

Scatter plot with color mapping:

- Points are **colored by `Genre`**.
- **Distinguishes genres** visually.

```
# Create a scatter plot where point size represents the Budget (BudMils)
ggplot(data=mymov, aes(x=CRating, y=ARating, colour=Genre, size=BudMils)) + geom_point()
```

Scatter plot with size mapping:

- **Point size represents** `BudMils` (budget in millions).
- **Visualizes budget** alongside ratings and genres.

```
# Create a scatter plot with transparent points (alpha = 0.5)
ggplot(data=mymov, aes(x=CRating, y=ARating, colour=Genre, size=BudMils)) + geom_point(alpha = 0.5)
```

- **Adds transparency** to points (alpha = 0.5).
- **Helps with overlapping points**, making the plot clearer.

```
# Create a base plot object for future customization and layering
mybase1 <- ggplot(data=mymov, aes(x=CRating, y=ARating, colour=Genre, size=BudMils))
```

- **Creates a base plot object** (`mybase1`).
- **Includes mappings** for `CRating`, `ARating`, `Genre`, and `BudMils`.
- **Can be reused** and customized with additional layers.

```
# Add points to the base plot with transparency (alpha = 0.5)
mybase1 + geom_point(alpha=0.5)
```

- **Adds points with transparency** to the base plot (`mybase1`).
- **Uses predefined mappings** in the base plot.

```
# Add points and lines to the base plot; lines may connect
# points but might not be meaningful here
mybase1 + geom_point(alpha=0.5) + geom_line()
```

- **Adds points and lines** to the base plot (`mybase1`).
- **Lines connect points** but may not be meaningful in a scatter plot.

```
# Override the point size mapping to use CRating instead of BudMils
mybase1 + geom_point(aes(size=CRating)) + labs(size="CRating")
```

- **Overrides size mapping**:

Size now represents `CRating` instead of `BudMils`.

- **Updates the legend** title to "CRating".

```
# Override the point color mapping to use BudMils instead of Genre
mybase1 + geom_point(aes(colour=BudMils)) + labs(colour="Budget in Millions")
```

- **Overrides color mapping**:

Color now represents `BudMils` instead of `Genre`.

- **Updates the legend** title to "Budget in Millions".

4.6 Settings vs. Mappings

```
# Create another base plot without specific mappings, for comparison
mybase2 <- ggplot(data=mymov, aes(x=CRating, y=ARating))
```

- **Creates a simpler base plot** (`mybase2`).
- **No color or size mappings** are applied, only x and y.

```
# Color points by Genre using mapping
mybase2 + geom_point(aes(colour = Genre))
```

- **Adds points colored by `Genre`** to the base plot (`mybase2`).
- **Color mapping helps distinguish** between genres.

```
# Set a fixed color for all points without mapping
mybase2 + geom_point(colour = "#9633ff")
```

- **Sets a fixed color** for all points (`#9633ff` - a shade of purple).
- **No color mapping** to variables, all points are the same color.

```
# Incorrect: Attempting to set a fixed color inside aes(), all points will be colored blue
mybase2 + geom_point(aes(colour = "blue"))
```

- **Incorrectly attempts to set a fixed color** (`"blue"`) inside `aes()`.
- **Incorrect usage**: Trying to set a fixed color inside `aes()` (which should be used for mappings).
- **Correct usage**: Fixed colors should typically be set outside of `aes()` unless mapping is involved.
- **Results in all points being different color** this method is not recommended.

```
# Fixed color using a hexadecimal value, but still inside aes()
mybase2 + geom_point(aes(colour = "#9033ff"))
```

- **Incorrectly uses `aes()`** for a fixed color (`#9033ff`).
- **All points will have the same color**, but using `aes()` here is unnecessary.

4.7 Geometric and Statistical Plots

```
# Create a base plot for Budget (BudMils)
mybase3 <- ggplot(data=mymov, aes(x=BudMils))

# Create a histogram with 15 bins
mybase3 + geom_histogram(bins=15)
```

- **Create a base plot** (`mybase3`) for visualizing the distribution of `BudMils`.
- **Add a histogram** with 15 bins to visualize the distribution of movie budgets.

```
# Set color for histogram outlines
mybase3 + geom_histogram(bins=10, colour="blue")

# Fill histogram bars with color
mybase3 + geom_histogram(bins=10, colour="white", fill="blue")
```

- **Add color to histogram outlines**.
- **Fill histogram bars** with a specified color.

```r
# Map fill color to Genre
mybase3 + geom_histogram(bins=10, colour="black", aes(fill=Genre))
```

- **Map fill color to `Genre`**, allowing different genres to have different colors within the histogram.

```r
# Create a density plot for BudMils
mybase3 + geom_density()
```

```r
# Improve density plot by adding fill color and transparency
mybase3 + geom_density(aes(fill=Genre), alpha=0.5)
```

- **Create a density plot** to visualize the distribution of `BudMils`.
- **Enhance the plot** by adding fill colors for different genres and making the plot semi-transparent.

```r
# Stack density plots by Genre with transparency
mybase3 + geom_density(aes(fill=Genre), position="stack", alpha=0.5)
```

- **Stack the density plots** by `Genre` and add transparency to better visualize the overlapping distributions.

4.8 Exercises and Comparisons

```r
# Create a histogram for ARating
mybase4 <- ggplot(data=mymov, aes(x=ARating))
mybase4 + geom_histogram(bins=15)
```

```r
# Customize the histogram
mybase4 + geom_histogram(bins=10, colour="white", fill="#ff33f0")
```

```r
# Map fill color to Genre
mybase4 + geom_histogram(bins=10, colour="black", aes(fill=Genre))
```

- **Create and customize histograms** for `ARating`.
- **Map fill colors to `Genre`** for better differentiation.

```r
# Create a histogram for CRating
mybase5 <- ggplot(data=mymov, aes(x=CRating))
mybase5 + geom_histogram(bins=15)
```

```r
# Customize the histogram
mybase5 + geom_histogram(bins=10, colour="white", fill="#3396ff")
```

```r
# Map fill color to Genre
mybase5 + geom_histogram(bins=10, colour="black", aes(fill=Genre))
```

- **Create and customize histograms** for `CRating`.
- **Map fill colors to `Genre`** for better visualization.

```r
# Create a scatter plot with a trend line (smoothing)
mybase4 <- ggplot(data=mymov, aes(x=CRating, y=ARating))
mybase4 + geom_point() + geom_smooth()
```

- **Create a scatter plot** with a trend line to visualize the relationship between `CRating` and `ARating`.

```r
# Create a boxplot for ARating by Genre
mybase5 <- ggplot(data=mymov, aes(x=Genre, y=ARating, colour = Genre))
mybase5 + geom_boxplot() + geom_point()
```

```r
# Create a boxplot with jitter to show individual points
mybase5 + geom_boxplot() + geom_jitter()
```

- **Create a boxplot** to compare `ARating` across different genres.
- **Add jittered points** to the boxplot to show individual data points.

```r
# Improve boxplot by filling colors by Genre and adding transparency
mybase5 + geom_boxplot(aes(fill=Genre), alpha=0.5)
```

```r
# Stack boxplots by Genre with customized settings
mybase5 + geom_boxplot(aes(fill=Genre), size=2, alpha=0.5)
```

- **Enhance boxplots** by filling them with genre-based colors and adding transparency.
- **Stack and customize boxplots** for better visualization.

4.9 Loading the Data and Renaming Columns

```r
library(ggplot2)
mymov <- read.csv("datasets/MovieRatings.csv")
colnames(mymov) <- c("Film", "Genre", "CRating", "ARating", "BudMils", "Year")
```

- **Load the `ggplot2` library**: This library is used for creating plots in R.
- **Read the CSV file** `MovieRatings.csv` into a dataframe called `mymov`.
- **Rename the columns** of the dataframe to more meaningful names:
- `Film`: Title of the movie.
- `Genre`: Genre of the movie (e.g., Action, Drama).
- `CRating`: Critic rating.
- `ARating`: Audience rating.
- `BudMils`: Budget in millions.
- `Year`: Release year.

4.10 Creating a Histogram with a Specific Color

```r
m <- ggplot(data=mymov, aes(x=BudMils, fill=Genre))
m + geom_histogram(bins = 15, colour = "blue")
```

- **Create a base plot (`m`)**: The x-axis is `BudMils` (budget in millions), and the fill color of the bars is mapped to `Genre`.
- **Add a histogram layer**:
- The histogram is divided into 15 bins.
- The **outline color of the bars** is set to blue (`colour = "blue"`).
- The **bars are filled** based on the `Genre` of the movie.

4.11 Focusing on a Specific Range Using `coord_cartesian()`

```r
# Focus on the x(0, 50) and y(0,50)
# Here xlim, ylim cut out important points
```

```r
# m+geom_histogram(bins = 15, colour = "blue") + ylim(0,50)
m + geom_histogram(bins=15,colour="blue") + ylim(0,50)
```

- Effect of ylim(0, 50): The ylim() function restricts the visible y-range to between 0 and 50.
- Any part of the data with y-values higher than 50 will be removed from the plot. Also removed the starting below 50 but its ends outside the 50 limit.
- Unlike coord_cartesian(), which zooms in while keeping all data points intact, ylim() discards data outside the specified limits.

```r
m + geom_histogram(bins = 15, colour = "blue") +
  coord_cartesian(ylim=c(0,50))
```

- **First commented-out code**:
- The code suggests using `ylim=c(0,50)` to limit the y-axis between 0 and 50, but this approach **cuts off data points** that fall outside this range.
- The comment highlights that `xlim` and `ylim` can remove important data points from the plot.
- **Using `coord_cartesian()`**:
- Instead of using `ylim`, the `coord_cartesian(ylim=c(0,50))` function is used.
- This approach **zooms in** on the y-axis between 0 and 50 without removing any data points outside this range. It just restricts the viewable area of the plot, preserving all data points. "Without removing any data points outside this range": it still exists in the plot's structure, but you won't see it if you zoom in to only show the 0-50 range.
- Zoom in on a specific range (e.g., y-axis from 0 to 50) using coord_cartesian().
- No data is removed; points outside this range still exist.
- Only the visible area is restricted to the specified range.
- Data points outside the range remain in the plot but are not shown.

4.12 Creating a Scatter Plot and Applying Axis Limits

```r
# In scatterplot we can use xlim ylim no problem
n <- ggplot(data=mymov, aes(x=CRating, y=ARating, colour=Genre))
n + geom_point(size=3)
```

```r
n + geom_point(size=3) + xlim(0,50) + ylim(0,50)
```

- **Create a scatter plot (`n`)**:
- `CRating` is mapped to the x-axis and `ARating` to the y-axis.
- Points are colored by `Genre`, and the size of the points is set to 3.
- **Apply axis limits**:
- In scatter plots, using `xlim(0,50)` and `ylim(0,50)` is generally fine.
- These functions **limit the x and y axes** to the specified ranges.
- This can help focus on a specific area of interest, but it might **cut off some data points** outside these ranges, which is why a warning is issued.

4.12.1 Warning Message Explanation

```
# we got warning message because we cut the datapoint
```

Explanation of the warning:

- The warning appears because `xlim` and `ylim` can exclude data points that fall outside the specified range.
- R notifies you that some points have been removed from the plot due to these limits, which could potentially exclude important information from your visualization.

4.13 Summary

- **Histograms**: `coord_cartesian()` is used to focus on a specific y-axis range without losing data, whereas `ylim` may remove points outside the range.
- **Scatter plots**: `xlim` and `ylim` work well but can cut off data, which triggers a warning. This technique should be used carefully depending on whether the full data range is needed.

4.14 Loading Data and Preparing the Base Plot

```
# Load ggplot2 for creating plots
# Read the CSV file into a dataframe named 'mymov'
# Rename columns for clarity
library(ggplot2)
mymov <- read.csv("datasets/MovieRatings.csv")
colnames(mymov) <- c("Film", "Genre", "CRating", "ARating", "BudMils", "Year")
```

- **Load the `ggplot2` library** for creating visualizations.
- **Read the CSV file** `MovieRatings.csv` into a dataframe named `mymov`.
- **Rename the columns** of the dataframe to more meaningful names, making it easier to work with the data.

4.15 Facets of Histograms

4.15.1 Creating a Basic Histogram

```
# Base plot with 'BudMils' on x-axis, bars filled by 'Genre'
m <- ggplot(data=mymov, aes(x=BudMils, fill=Genre))


# Add histogram with 15 bins and blue borders around the bars
m + geom_histogram(bins = 15, colour = "blue")
```

- **Create a base plot (`m`)** where `BudMils` (budget in millions) is on the x-axis and the bars are filled based on `Genre`.
- **Add a histogram layer** with 15 bins and blue borders around the bars.

4.15.2 Adding Facets

```
m + geom_histogram(bins=15, colour="blue") +
  facet_grid(Genre~.)
```

- **Add facets** to the histogram:

- **Facet by Genre**, creating a separate subplot (subfigure) for each genre.
- The `Genre~.` syntax places subfigures in rows.

4.15.3 Customizing Facet Scales

```
m + geom_histogram(bins=15, colour="blue") +
  facet_grid(Genre~., scale = "free")
```

Use free scales for each facet:

- The `scale = "free"` option allows each subplot to have its own y-axis scale, which is useful when the data varies widely between genres.

4.15.4 Facets in Columns

```
m + geom_histogram(bins=15, colour="blue") +
  facet_grid(.~Genre)
```

Arrange facets in columns:

- The `.~Genre` syntax places subfigures in columns rather than rows, allowing a vertical comparison of genres.

4.16 Facets of Scatterplots

4.16.1 Creating a Base Scatterplot

```
n <- ggplot(data=mymov, aes(x=CRating, y=ARating, colour=Genre))
n + geom_point(size=3)
```

- **Create a scatterplot** (n) where `CRating` is on the x-axis, `ARating` on the y-axis, and points are colored by `Genre`.
- **Set the size of the points** to 3.

4.16.2 Adding Facets to the Scatterplot

```
n + geom_point(size=3) + facet_grid(Genre~., scale ="free")
```

Facet the scatterplot by Genre:

- Each genre gets its own subplot with its own scales (if necessary).

```
n + geom_point(size=3) + facet_grid(Year~.)
```

Facet by Year:

- This creates subplots for each year, showing how critic and audience ratings vary over time.

```
n + geom_point(size=3) + facet_grid(.~Year)
```

Arrange the facets by year in columns:

- This arrangement is useful for comparing ratings across different years vertically.

```
n + geom_point(size=3) + facet_grid(Genre~Year, scale = "free")
```

4.16.2.1 Combining Facets for Both Genre and Year **Combine Genre and Year in a grid**:

- Facets are created for each combination of `Genre` and `Year`, with free scales to accommodate varying range.

4.17 Customizing Plot Themes

4.17.1 Customizing Axis Labels and Titles

```
x <- m + geom_histogram(bins=15, colour="blue")

x + xlab("Cost of the movies") +
  ylab("Number of Movies") +
  theme(axis.title.x = element_text(size=10, colour="blue"),
        axis.title.y = element_text(size=10, colour = "red"))
```

- **Set custom axis labels**:
- `xlab("Cost of the movies")` and `ylab("Number of Movies")` change the axis labels to more descriptive names.
- **Customize axis titles**:
- `element_text(size=30, colour="blue")` and `element_text(size=50, colour="red")` set the font size and color of the axis titles.

4.17.2 Customizing Tick Mark Labels

```
x + xlab("Cost of the movies") +
  ylab("Number of Movies") +
  theme(axis.title.x = element_text(size=10, colour="blue"),
        axis.title.y = element_text(size=10, colour = "red"),
        axis.text.x = element_text(size=10, colour="darkgreen"),
        axis.text.y = element_text(size=10, colour="pink"))
```

- **Customize tick mark labels**:
- `axis.text.x` and `axis.text.y` modify the font size and color of the tick mark labels on the x and y axes, respectively.

4.18 Moving the Legend Inside the Plot Area

```
x + xlab("Cost of the movies") +
  ylab("Number of Movies") +
  theme(axis.title.x = element_text(size=10, colour="blue"),
        axis.title.y = element_text(size=10, colour = "red"),
        axis.text.x = element_text(size=10, colour="darkgreen"),
        axis.text.y = element_text(size=10, colour="pink"),
        legend.position = c(1,1),
        legend.justification = c(1,1),
        legend.title = element_text(size = 10, colour = "orange"),
        legend.text = element_text(size=10 , colour = "blue"))
```

- **Position the legend inside the plot area**:
- `legend.position = c(1,1)` places the legend at the top-right corner of the plot area.
- `legend.justification = c(1,1)` aligns the legend to the top-right corner.
- **Customize the legend**:

- `legend.title` and `legend.text` adjust the font size and color of the legend title and text.

4.19 Adding a Title to the Plot

```
x + xlab("Cost of the movies") +
  ylab("Number of Movies") +
  ggtitle("Budget Distribution of Movies 2007~2011") +
  theme(axis.title.x = element_text(size=10, colour="blue"),
        axis.title.y = element_text(size=10, colour = "red"),
        axis.text.x = element_text(size=10, colour="darkgreen"),
        axis.text.y = element_text(size=10, colour="pink"),
        legend.position = c(1,1),
        legend.justification = c(1,1),
        legend.title = element_text(size = 10, colour = "orange"),
        legend.text = element_text(size=10 , colour = "blue"),
        plot.title = element_text(size = 10, colour="purple", hjust=0))
```

- **Add a title to the plot** using `ggtitle()`:
  - The title "Budget Distribution of Movies 2007~2011" is added at the top of the plot.
- **Customize the plot title**:
  - `plot.title = element_text(size = 20, colour="purple", hjust=0)` sets the font size, color, and horizontal justification (`hjust`) of the title. Setting `hjust=0` aligns the title to the left.

```
x +
  # Set x-axis label
  xlab("Cost of the Movies") +
  # Set y-axis label
  ylab("Number of Movies") +
  ggtitle("Budget Distribution of Movies 2007-2011") +
  # Add a title to the plot
  theme(
    axis.title.x = element_text(size = 11, colour = "blue"),
    # Reduce x-axis title size
    axis.title.y = element_text(size = 10, colour = "red"),
    # Reduce y-axis title size
    axis.text.x = element_text(size = 10, colour = "darkgreen"),
    # Reduce x-axis tick label size
    axis.text.y = element_text(size = 10, colour = "pink"),
    # Reduce y-axis tick label size
    legend.position = c(0.85, 0.85),
    # Move legend inside plot, slightly offset
    legend.justification = c(1, 1),
    # Align legend to top-right
    legend.title = element_text(size = 12, colour = "orange"),
    # Reduce legend title size
    legend.text = element_text(size = 12, colour = "blue"),
    # Reduce legend text size
    plot.title = element_text(size = 14, colour = "purple", hjust = 0.5)
    # Adjust plot title size, center align
  )
```

5 Part-5

5.1 Loading Data and Summarizing

```r
# Load the dataset from 'manheim.csv' into 'carsale'
carsale <- read.csv("datasets/manheim.csv")

# Provide a summary of the dataset
summary(carsale)
```

- **Load the dataset:** The `carsale` dataframe is created by reading data from the CSV file `manheim.csv`.
- **Summary statistics:** `summary(carsale)` provides a basic statistical summary of each column, including min, max, mean, median, and quantiles.

```r
head(carsale, 5)
```

```r
tail(carsale, 5)
```

5.2 Basic Statistical Functions for Price

```r
# Find the minimum price
min(carsale$price)
# Find the maximum price
max(carsale$price)
# Calculate the variance of price
var(carsale$price)
# Calculate the standard deviation of price
sd(carsale$price)
# Find the range of prices
range(carsale$price)
# Calculate the interquartile range (IQR) of price
IQR(carsale$price)
```

- **Minimum and Maximum:** Identify the lowest and highest prices in the dataset.
- **Variance and Standard Deviation:** Measure the spread or variability of the price data.
- **Range:** Gives the difference between the minimum and maximum prices.
- **Interquartile Range (IQR):** Measures the range within which the central 50% of the prices fall.

5.3 Correlation Between Miles and Price for Model X

```r
# Filter the dataset for model "X" cars
saleX <- carsale[carsale$model=="X",]
# Calculate the correlation between price and miles for model "X"
cor.test(saleX$price, saleX$miles)
```

- **Filter by Model:** `saleX` is a subset of the `carsale` dataframe, containing only rows where the model is "X".
- **Correlation Test:** `cor.test` calculates the Pearson correlation between `price` and `miles` for model "X". The result includes the correlation coefficient (r) and p-value.

5.4 Pearson's Product-Moment Correlation Interpretation

5.4.1 Correlation Coefficient (cor = -0.6531409):

The correlation coefficient is **-0.653**.

- This indicates a **moderate negative correlation** between price and miles. As the number of miles increases, the price tends to decrease. A correlation of -1 would be a perfect negative correlation, while 0 would indicate no correlation.

5.4.2 t-statistic (t = -16.067):

- The t-statistic of **-16.067** shows how many standard deviations the sample correlation is away from zero.
- This large negative value suggests a **strong deviation from no correlation (zero)**.

5.4.3 Degrees of Freedom (df = 347):

- The degrees of freedom (df = 347) indicate the sample size minus 2 (**n - 2**).
- This suggests the test was performed on **349 observations (n = 349)**.

5.4.4 p-value (< 2.2e-16):

- The **p-value** is extremely small (**essentially 0**), meaning the result is **highly statistically significant**.
- This provides strong evidence to reject the null hypothesis that the true correlation is 0 (i.e., no linear relationship between price and miles).
- With a p-value this small, there is **overwhelming evidence** that a relationship exists between price and miles.

5.4.5 Confidence Interval (95% CI: -0.709 to -0.589):

- The 95% confidence interval for the true correlation is between **-0.709** and **-0.589**.
- We are **95% confident** that the true correlation in the population lies within this range.
- Since the interval is entirely negative and doesn't include zero, this further confirms a **statistically significant negative correlation** between the two variables.

5.4.6 Alternative Hypothesis:

- The alternative hypothesis is that the true correlation is **not equal to 0**.
- Based on the p-value and the confidence interval, we **reject the null hypothesis** and accept the alternative hypothesis.
- This means that there is a **linear relationship** between price and miles, and it's **negative**.

5.5 Scatter Plot of Miles vs. Price

```r
# Load ggplot2 for plotting
library(ggplot2)
# Create a base plot for miles vs. price
mybase <- ggplot(data=saleX, aes(x=miles, y=price, colour=sale))
# Add points with size 3 and 50% transparency
mybase + geom_point(size=3, alpha = 0.5)
```

- **Create Scatter Plot:** `mybase` sets up a scatter plot with `miles` on the x-axis and `price` on the y-axis.

- **Plot Points:** `geom_point(size=3, alpha=0.5)` adds points to the plot, with a size of 3 and 50% transparency. This visualizes the relationship between miles and price for model "X".

5.6 Simple Linear Regression (SLR)

```
# Fit a linear regression model with price as the response
# and miles as the predictor
mySLR <- lm(data=saleX, price~miles)

# Summary of the regression model
summary(mySLR)
```

- **Fit Linear Model:** `lm` fits a linear model where `price` is predicted by `miles`.
- **Model Summary:** `summary(mySLR)` provides detailed statistics about the regression, including coefficients, R-squared, and p-values.

5.7 Linear Regression Output Explained

5.7.1 Correlation Coefficient (R-squared and Adjusted R-squared)

- **R-squared: 0.4266**: This means that about **43%** of the variation in `price` is explained by `miles`. A higher value would be better because it would mean that more of the changes in `price` can be predicted by `miles`.
- **Adjusted R-squared: 0.4249**: This value is slightly lower than R-squared because it adjusts for the number of variables. It also shows that about **42%** of the variance in `price` is explained by `miles`. Higher is better.
- How It Works:

  If you add a variable that significantly improves the model, Adjusted R-squared will increase. If you add a variable that doesn't help improve the model, Adjusted R-squared will decrease.
- Practical Meaning:

  Higher Adjusted R-squared means the model explains a good amount of the variance, accounting for the number of predictors. It reflects how well the model generalizes to new, unseen data.

  Lower Adjusted R-squared means the model either has too many unnecessary predictors or doesn't explain the variance well.

5.7.2 p-value

- **p-value $<$ 2.2e-16**: This is a very small p-value, which means the relationship between `price` and `miles` is **highly statistically significant**. Lower p-values are better because they show the results are less likely to be due to chance.

5.7.3 t-statistic

- **t = -16.07**: This shows that the relationship between `miles` and `price` is strong. A larger t-value (in absolute terms) is better because it indicates that `miles` is having a significant effect on `price`.

  Higher (in absolute value) is better: A larger t-value (either positive or negative) indicates stronger evidence against the null hypothesis (which assumes no relationship).

  A larger t-value (in absolute terms) means that the coefficient is more significantly different from zero.

5.7.4 Residual Standard Error (RSE)

- **Residual Standard Error: 1370**: This tells us how far off the model's predictions are from the actual prices, on average. Lower is better because it means the model's predictions are closer to the actual prices.

5.7.5 F-statistic

- **F-statistic: 258.2**: This is a measure of how well the model fits the data. Higher is better because it means the model does a better job of explaining the changes in `price` based on `miles`.

  A higher F-statistic means that the model provides a better fit to the data than would be expected by chance

5.7.6 What Does This Mean?

- **R-squared and Adjusted R-squared**: These tell us how much of the changes in `price` can be predicted by `miles`. Higher values would be better, but in this case, about 42-43% is explained.
- **p-value**: This is very low, so the relationship between `miles` and `price` is statistically significant.
- **t-statistic**: The high t-value shows that `miles` has a strong effect on `price`.
- **Residual Standard Error**: This shows the average difference between the predicted price and the actual price. A lower value is better, but here it's 1370.
- **F-statistic**: A high F-statistic means the model does a good job of explaining the relationship between `price` and `miles`.

5.7.7 Summary:

- The model shows that as `miles` increases, the `price` decreases, and this relationship is statistically significant. However, the model explains about 43% of the variability in `price`, meaning there could be other factors affecting the price as well.

1. R-squared and Adjusted R-squared: Higher values are better because they indicate more of the variance in price is explained by miles.
2. p-values: Lower values are better because they indicate stronger statistical significance.
3. t-statistic: A larger t-statistic (in absolute terms) is better, indicating a more significant effect.
4. Residual Standard Error: Lower is better because it indicates the predictions are closer to the actual values.
5. F-statistic: Higher is better, indicating that the model fits the data well.

5.7.8 Predicting Price for Specific Mileage

```r
# Predict price for a car with 62,000 miles
predict(mySLR, data.frame(miles=c(62000)))

# Extract the coefficients from the regression model
mySLR$coefficients
```

- **Predict Price:** Use the fitted model to predict the price of a car with 62,000 miles.
- **data.frame(miles = c(62000)):** This creates a data frame with a column named miles and a value of 62,000. The predict() function needs the new data to be in a data frame format, even if you are only making a single prediction.

- **Model Coefficients:** `mySLR$coefficients` retrieves the intercept and slope from the linear model.

5.7.9 Custom Prediction Function

```
# Manually calculate predicted price
mypredict <- function(mymiles){
  mySLR$coefficients[1] + mySLR$coefficients[2] * mymiles
}

# Use the custom function to predict the price for 62,000 miles
mypredict(62000)
# Compare with the built-in predict function
predict(mySLR, data.frame(miles=c(62000)))
```

- **Custom Prediction Function:** `mypredict` manually calculates the predicted price using the linear regression coefficients.
- **Test Prediction:** Both `mypredict(62000)` and `predict(mySLR, data.frame(miles=c(62000)))` should return the same predicted price.

5.7.10 Confidence Interval for Prediction

```
# Predict with a 95% confidence interval
predict(mySLR, data.frame(miles=c(62000)), interval="confidence", level=0.95)
```

5.7.11 Fit (15,740.56):

- The predicted price for a car with **62,000 miles** is **$15,740.56**.
- This is the best estimate for the price based on the linear model (`mySLR`).

5.7.12 95% Confidence Interval:

- The confidence interval gives a range within which we are **95% confident** that the true mean price for a car with 62,000 miles will fall.
- **Lower bound (lwr): $15,384.15**: We are 95% confident that the price will not be lower than this value.
- **Upper bound (upr): $16,096.96**: We are 95% confident that the price will not be higher than this value.

5.7.13 Interpretation:

- The predicted price is **$15,740.56**, but due to the uncertainty in the data, we can only be **95% confident** that the true mean price lies between **$15,384.15** and **$16,096.96**.
- The confidence interval is fairly narrow, suggesting that the model is relatively certain about the prediction.
- **Confidence Interval:** Predict the price for a car with 62,000 miles, including a 95% confidence interval. This interval provides a range within which the true price is likely to fall with 95% confidence.

5.7.14 Summary:

- **Data Exploration:** Use summary statistics and basic functions to explore the `price` variable.

- **Correlation Analysis:** Determine the relationship between `miles` and `price` for a specific model using correlation and visualization.
- **Linear Regression:** Fit a linear model to predict `price` based on `miles` and evaluate the model using summary statistics.
- **Prediction:** Generate predictions for specific mileages using the regression model, including confidence intervals for more informed decision-making.

5.8 Multiple Linear Regression Model

5.8.1 Loading Data and Fitting a Multiple Linear Regression Model

```r
# Load the 'datarium' package which contains the 'marketing' dataset
library("datarium")

# Load the 'marketing' dataset into 'mydf'
mydf <- marketing
```

- **Load the datarium package:** This package includes various datasets, including the `marketing` dataset.
- **Assign the dataset:** The `marketing` dataset is assigned to the variable `mydf` for easier access.

```r
str(mydf)
```

```r
head(mydf)
```

5.9 Code for Splitting Data and Fitting Linear Regression:

```r
# install.packages("caTools")
library(caTools)

# Assume your dataset is named 'mydf'
set.seed(123)  # Set seed for reproducibility

# Split the data into 80% training and 20% testing
split <- sample.split(mydf$sales, SplitRatio = 0.8)

# Create the training and testing sets
train <- subset(mydf, split == TRUE)  # 80% for training
test <- subset(mydf, split == FALSE)  # 20% for testing

# Fit the linear regression model without the 'newspaper' variable
myMLR <- lm(sales ~ youtube + facebook, data = train)

# View the summary of the model
summary(myMLR)

# Make predictions on the test set
predictions <- predict(myMLR, newdata = test)

# Print the first few predictions
head(predictions)
```

5.10 Code for without Splitting Data and Fitting Linear Regression:

```
# Fit a multiple linear regression model with sales as the response variable
myMLR <- lm(data=mydf, sales ~ youtube + facebook + newspaper)

# Summarize the model to get detailed statistics
summary(myMLR)
```

5.11 Linear Regression Model Interpretation

This model shows how spending on **YouTube**, **Facebook**, and **Newspaper** ads affects **sales**. Here's what the results mean:

5.11.1 Residuals:

- **Min (-10.5932)**: The model overestimated sales by this amount for the worst prediction.
- **Max (3.3951)**: The model underestimated sales by this amount for the worst case.
- **Median (0.2902)**: The middle value of the residuals is close to zero, meaning the model generally predicts sales well.

5.11.2 Coefficients:

- **Intercept (3.526667)**: If no money is spent on ads (YouTube, Facebook, or Newspaper), the predicted sales would be about **3.53 units**.
- **YouTube (0.045765)**: For every additional unit spent on YouTube ads, sales increase by **0.0458 units**. This effect is highly significant, meaning it has a strong impact on sales.
- **Facebook (0.188530)**: For every additional unit spent on Facebook ads, sales increase by **0.1885 units**. This effect is also highly significant and has a bigger impact on sales than YouTube ads.
- **Newspaper (-0.001037)**: Newspaper ads do not have a significant effect on sales. The small and negative coefficient suggests that spending on newspaper ads does not boost sales.

5.11.3 R-squared and Adjusted R-squared:

- **R-squared (0.8972)**: This means that **89.72%** of the variation in sales can be explained by spending on YouTube, Facebook, and Newspaper ads combined.
- **Adjusted R-squared (0.8956)**: This adjusts for the number of predictors and still shows that about **89.56%** of the variation in sales is explained by the model, meaning the model is a good fit.

5.11.4 F-statistic:

- The **F-statistic** is very high, indicating that the overall model is statistically significant.
- The p-value is extremely small, meaning the model as a whole has a strong ability to explain sales.
- **Fit the Multiple Linear Regression (MLR) model:** `lm()` fits a model where `sales` is predicted by three independent variables: `youtube`, `facebook`, and `newspaper`.
- **Model Summary:** `summary(myMLR)` provides detailed statistics about the model, including the coefficients, p-values, R-squared value, and more.

```
# Extract residuals from the model
residuals <- resid(myMLR)

# Plot the residuals histogram
```

```r
hist(residuals,
     main = "Histogram of Residuals",
     xlab = "Residuals",
     col = "lightblue",
     border = "black")
```

5.12 Interpretation of the Residuals Histogram

5.12.1 Symmetry/Shape:

- The histogram is **slightly right-skewed**. Most of the residuals are concentrated around zero, but there are a few residuals that extend into the negative values.
- Ideally, residuals should be **normally distributed** (bell-shaped), with most values near zero if the linear regression model is a good fit for the data. The slight skewness suggests some **minor departure from normality**.

5.12.2 Center:

- The residuals are **centered around zero**, which is expected in a good model. This indicates that, on average, the model does not systematically over-predict or under-predict sales.

5.12.3 Spread:

- The spread of the residuals shows that most residuals are between **-5 and 5**. A few outliers extend below **-10**.
- A wider spread or outliers (as seen on the left side) may indicate that the model's predictions are not perfect for all data points, especially those that deviate more from the average.

5.12.4 Frequency:

- The highest frequency (the tallest bar) is around residuals close to **zero**, meaning that for most data points, the predicted sales are very close to the actual sales.

5.12.5 Outliers:

- The few residuals on the far left (around **-10**) indicate **potential outliers** or observations where the model over-predicted sales by a large margin.

5.12.6 Conclusion:

- The histogram shows that the residuals are generally close to zero, which is a **good sign**. However, the slight skewness and the presence of some larger negative residuals suggest that there may be **some non-normality or outliers** in the data.
- It might be worth investigating those **negative outliers** to see if they represent unusual data points or if the model could be improved to better handle them.

5.12.7 Coefficients Interpretation

```r
# Extract the coefficients from the regression model
myMLR$coefficients
```

- **Model Coefficients:** `myMLR$coefficients` retrieves the coefficients for each predictor variable, including the intercept. These coefficients indicate how much the `sales` are expected to change with a one-unit change in each predictor while holding other predictors constant.

5.12.8 Interpreting the Regression Equation

```
# Regression equation based on coefficients:
# sales = 3.526667243 + 0.04576464*youtube + 0.188530017*facebook - 0.001037493*newspaper
```

Regression Equation: The equation derived from the coefficients:

- **Intercept (3.526667243):** The base sales when all predictors are 0.
- **Youtube (0.04576464):** For every additional unit spent on YouTube, sales are expected to increase by approximately 0.046 units, assuming other factors remain constant.
- **Facebook (0.188530017):** For every additional unit spent on Facebook, sales are expected to increase by approximately 0.189 units.
- **Newspaper (-0.001037493):** For every additional unit spent on newspapers, sales are expected to decrease by approximately 0.001 units.

5.12.9 Assessing the Importance of Predictors Using p-values

```
# Negative coefficient for 'newspaper' suggests
# a slight decrease in sales with increased newspaper spending.
# p-value for 'newspaper' is 0.86, which is much greater than 0.05,
# indicating it is not statistically significant.
```

- **Negative Coefficient for Newspaper:** The negative coefficient suggests that increasing spending on newspapers might slightly decrease sales, but this effect is very small.
- **P-value Significance:** The p-value for the `newspaper` variable is 0.86, which is much greater than the typical significance level of 0.05. This high p-value suggests that the newspaper variable does not significantly contribute to the prediction of sales and may not be necessary in the model.

5.12.10 Removing Insignificant Predictors and Refitting the Model

```
# Refit the model without the 'newspaper' variable
myMLR <- lm(data=mydf, sales ~ youtube + facebook)

# Summarize the new model
summary(myMLR)
```

- **Refit the Model Without Newspaper:** The model is refit by excluding the `newspaper` variable, which had a high p-value and was not significant.
- **New Model Summary:** The new `summary(myMLR)` provides the updated model statistics, which likely show a better fit or more accurate coefficients for the significant predictors (`youtube` and `facebook`).

5.12.11 Summary:

- **Initial Model:** A multiple linear regression model was fitted to predict `sales` using `youtube`, `facebook`, and `newspaper` as predictors.
- **Coefficient Interpretation:** The coefficients from the model provide insights into how each predictor affects sales.
- **Importance of Predictors:** The p-value for `newspaper` was high, indicating it is not a significant predictor of sales.

5.13 Logistic Regression

In logistic regression, we model the relationship between one or more predictor variables (independent variables) and a binary outcome (dependent variable). The goal of logistic regression is to estimate the probability that a certain event will occur, given the values of the predictor variables.

5.14 Loading Libraries and Data

```
# install.packages("caTools")
# Load caTools for data splitting
library(caTools)
# Load the dataset from 'binary.csv'
mydata <- read.csv("datasets/binary.csv")
```

- **Load caTools library:** This library provides functions like `sample.split` for splitting datasets into training and testing sets.
- **Load the dataset:** The `binary.csv` file is loaded into the `mydata` dataframe.

```
head(mydata)
```

- `admit`: This is the target variable, likely representing whether a student was admitted (1) or not (0) to a program.
- `gre`: This column contains the GRE scores of the applicants.
- `gpa`: This column shows the GPA (Grade Point Average) of the applicants.
- `rank`: This column represents the prestige or ranking of the undergraduate institution of the applicants.

5.15 Splitting the Data

```
split <- sample.split(mydata$admit, SplitRatio=0.8)
# Split data into 80% training and 20% testing
# admit column is the target variable. It shows whether someone was admitted to the program or not:
# 1 means the person was admitted.
# 0 means the person was not admitted.
```

- **Data splitting:** `sample.split` splits the data based on the `admit` variable with an 80/20 ratio, creating a logical vector (`TRUE` for training, `FALSE` for testing).

```
# Create the training set (80% of data)
train <- mydata[split == TRUE,]

# Create the testing set (20% of data)
test <- mydata[split == FALSE,]
```

- **Create training and testing sets:** The data is divided into `train` and `test` based on the logical vector generated by `sample.split`

5.16 Data Wrangling and Model Fitting

```
str(mydata)
```

```
# Convert 'admit' to a factor
mydata$admit <- factor(mydata$admit)
# Convert 'rank' to a factor
mydata$rank <- factor(mydata$rank)
```

- **Convert to factors:** Both `admit` (binary response) and `rank` (categorical predictor) are converted to factors, which is necessary for logistic regression.

```
# Fit logistic regression model
lmodel <- glm(admit ~ gre + rank, data=train, family = binomial)

# Summarize the model
summary(lmodel)
```

- **Fit logistic regression model:** `glm()` fits a logistic regression model predicting `admit` based on `gre` and `rank`, using the `train` dataset.
- **Model summary:** `summary(lmodel)` provides the coefficients, standard errors, z-values, and p-values for each predictor.

5.17 Interpretation:

- **Intercept (-1.793159)**:
- The intercept represents the log-odds of being admitted when both `gre` and `rank` are zero. The p-value is **0.023622**, which means the intercept is statistically significant at the 5% level.
- **`gre` (0.003744)**:
- For each additional point in **GRE scores**, the log-odds of being admitted increase by **0.003744**. Since the p-value is **0.001075**, this effect is highly significant ($p < 0.01$). Therefore, **higher GRE scores** increase the chances of being admitted.
- **`rank` (-0.499054)**:
- For each one-unit increase in the **rank** (which indicates a worse rank), the log-odds of being admitted decrease by **0.499054**. The p-value is **0.000313**, meaning this effect is highly significant ($p < 0.001$). This indicates that applicants from **higher-ranked institutions** (lower values of `rank`) are more likely to be admitted.

5.17.1 Deviance:

- **Null deviance: 400.59** on 319 degrees of freedom: This is the deviance of a model with only the intercept and no predictors.
- **Residual deviance: 371.77** on 317 degrees of freedom: This is the deviance of the model after including the predictors (`gre` and `rank`). The decrease in deviance indicates that the predictors improve the model fit.

5.17.2 AIC:

- **AIC: 377.77**: The Akaike Information Criterion (AIC) is used to compare different models. Lower AIC values indicate a better fit. This AIC value can be compared with other models to determine which one fits the data best.

5.17.3 Significance Codes:

- `***`: p-value < 0.001 (highly significant)
- `**`: p-value < 0.01 (significant)
- `*`: p-value < 0.05 (marginally significant)

5.17.4 Conclusion:

- **GRE scores** and **rank** both significantly influence the chances of being admitted. Higher GRE scores increase the likelihood of admission, while higher ranks (worse-ranked schools) reduce the chances of admission.
- The model significantly improves the fit compared to a null model (only the intercept).

5.18 Key Points About Logistic Regression

5.18.1 Binary Outcome:

- Logistic regression is used when the outcome (dependent variable) is **binary**, meaning there are only two possible outcomes, typically coded as 0 or 1.
- Example:
  - **0 = Not admitted**, **1 = Admitted** (in your case).
  - **0 = No** (the event didn't happen), **1 = Yes** (the event happened).

5.18.2 Predicting Probabilities:

- Logistic regression doesn't predict the outcome directly as 0 or 1.
- Instead, it predicts the **probability** of the outcome occurring. For example, the probability of being admitted to a program.
- The result of logistic regression is a **value between 0 and 1** that represents this probability.

5.18.3 Link Function: Log-Odds:

- Logistic regression uses the **log-odds** (logarithm of the odds) to model the probability of the outcome.
- Log-odds can range from **-infinity to +infinity**, which are then transformed into probabilities using the logistic function.

5.18.4 Logistic Function:

- The **logistic function** (also called the **sigmoid function**) converts the log-odds into a **probability** between **0 and 1**.

5.18.5 Coefficients Interpretation:

- The coefficients in logistic regression represent the **change in log-odds** of the outcome for each unit increase in the predictor variable.
- Example: If the coefficient for `gre` is **0.0037**, it means that for each additional point in the GRE score, the **log-odds** of being admitted increase by **0.0037**.

5.19 Making Predictions and Evaluating the Model

```r
# Predict probabilities for the test set
res <- predict(lmodel, test, type="response")
res
```

5.19.1 What does `type = "response"` mean in `predict()`?

In logistic regression, the `predict()` function can return different types of predictions. Here's what it means when you use `type = "response"`.

5.19.2 type = "response":

- This argument tells R to return the **predicted probabilities** of the outcome, rather than the **log-odds**.
- Logistic regression predicts values that can either be:
  - **Log-odds**: If you use `type = "link"`, R will return the log-odds (the logarithm of the odds of the event happening).
  - **Probabilities**: If you use `type = "response"`, R will return the probability of the event happening (for example, the probability of being admitted).

Since `type = "response"` is used, R returns the **probability** of the outcome being 1 (e.g., being admitted).

- **Predict probabilities:** `predict()` is used to generate predicted probabilities of admission (`admit = 1`) for the test set.

```
# Create a confusion matrix
t1 <- table(ActVal = test$admit, PreVal = res > 0.5)

# Display the confusion matrix
print(t1)
```

5.19.3 What does res > 0.5 mean in Logistic Regression?

5.19.4 Predicted Probabilities:

- `res` contains the **predicted probabilities** for the test set, which were generated from the logistic regression model.
- These probabilities range between **0 and 1**, representing the likelihood of the outcome being `1` (for example, being admitted).

5.19.5 res > 0.5:

- The expression `res > 0.5` converts those probabilities into **binary predictions**:
  - If the predicted probability is **greater than 0.5**, it predicts the outcome as **1** (e.g., admitted).
  - If the predicted probability is **less than or equal to 0.5**, it predicts the outcome as **0** (e.g., not admitted).

5.20 Confusion matrix:

- **Confusion matrix:** A table comparing actual values (`test$admit`) with predicted values (`res > 0.5`). The threshold of 0.5 is used to classify probabilities into binary outcomes (admit or not).

```
cm_df <- as.data.frame(t1)

# Simple square confusion matrix
ggplot(cm_df, aes(x = PreVal, y = ActVal, fill = Freq)) +
  # Simple black-bordered tiles
  geom_tile(color = "black") +
  # Adds frequency counts
  geom_text(aes(label = Freq)) +
  theme_minimal() +
  labs(title = "Confusion Matrix", x = "Predicted", y = "Actual") +
  # Simple white-to-blue color fill
  scale_fill_gradient(low = "white", high = "steelblue") +
```

```r
  # Keeps the tiles square
  coord_fixed()
```

5.20.1 What Each Cell Represents:

- **Top left (23): False Negatives (FN)**
  - The model predicted `0` (not admitted), but the actual value was `1` (admitted).
  - There are **23 cases** where the model missed the correct admission.
- **Top right (2): True Positives (TP)**
  - The model predicted `1` (admitted), and the actual value was also `1` (admitted).
  - There are **2 correct predictions** for admitted students.
- **Bottom left (49): True Negatives (TN)**
  - The model predicted `0` (not admitted), and the actual value was also `0` (not admitted).
  - There are **49 correct predictions** for students who were not admitted.
- **Bottom right (6): False Positives (FP)**
  - The model predicted `1` (admitted), but the actual value was `0` (not admitted).
  - There are **6 incorrect predictions** for admitted students.

5.20.2 Summary:

- The model is better at predicting **not admitted** (49 correct predictions) than predicting **admitted** (only 2 correct predictions).

```r
accu <- (t1[1,1] + t1[2,2]) / sum(t1)
# Calculate the accuracy
accu
```

- **Calculate accuracy:** The accuracy is calculated by summing the true positives (`t1[1,1]`) and true negatives (`t1[2,2]`) and dividing by the total number of cases.

6 Part-6

6.1 Neural Network (NN)

6.2 Creating a Training Dataset

```
# Test scores
TKS <- c(20, 10, 30, 20, 80, 30)
# Course scores
CSS <- c(90, 20, 40, 50, 50, 80)
# Binary outcome (placed or not)
Placed <- c(1, 0, 0, 0, 1, 1)
# Combine into a dataframe
df <- data.frame(TKS, CSS, Placed)
```

- **Create features and labels:** `TKS` and `CSS` are predictor variables, while `Placed` is the binary response variable.
- **Combine into a dataframe:** The data is organized into `df`, which will be used for training the neural network.

6.3 Fitting the Neural Network

```
library(neuralnet)
# Load neuralnet package for training neural networks
nn <- neuralnet(Placed ~ TKS + CSS, data = df, hidden = 3,
                act.fct = "logistic", linear.output = FALSE)
# Plot the neural network
plot(nn)
```

- **Fit the neural network:** `neuralnet()` fits a neural network model with 3 hidden nodes in one layer, using the logistic activation function.
- **Plot the network:** `plot(nn)` visualizes the structure of the neural network.

6.4 Predicting New Data

```
# Test scores for new data
TKS <- c(30, 40, 85)
# Course scores for new data
CSS <- c(85, 50, 40)
# Create a test dataset
test <- data.frame(TKS, CSS)
```

- **Create test data:** New test data is created to predict whether students will be placed.

```
# Predict using the trained neural network
Predict <- compute(nn, test)
# Extract probabilities
prob <- Predict$net.result
# Convert probabilities to binary outcomes
pred <- ifelse(prob > 0.5, 1, 0)
# Display predicted results
pred
```

- **Compute predictions:** `compute()` generates predicted probabilities for the test dataset.

- **Convert to binary outcomes:** Probabilities are converted to binary predictions (`1` if greater than 0.5, otherwise `0`).

6.5 Exercise - Neural Network Model on Dataset

6.5.1 Exercise - Neural Network Model on `binary.csv`

6.5.1.1 Steps for the Exercise:

1. **80% Training, 20% Testing:**
   - Use the `sample.split()` function to divide `binary.csv` into 80% training and 20% testing datasets.
2. **Hidden Layer with 3 Nodes:**
   - Fit a neural network model with 3 hidden nodes using the `neuralnet()` function.
3. **Confusion Matrix & Accuracy:**
   - Predict the outcomes for the test set, create a confusion matrix, and calculate the accuracy of the model.
4. **Change Hidden Nodes (4, 5, 6):**
   - Repeat the model fitting process with 4, 5, and 6 hidden nodes. Compare the performance (accuracy) of the models and draw conclusions.

6.5.2 Summary:

- **Logistic Regression:** The logistic regression model is fitted on training data to predict admission (`admit`), followed by evaluating its performance using a confusion matrix and accuracy calculation.
- **Neural Network:** A simple neural network is trained with manually created data, followed by predictions and binary classification.
- **Exercise:** The provided steps guide you to apply neural networks to the `binary.csv` dataset, experimenting with different model configurations and evaluating their performance.

```r
# Load the dataset
mydata <- read.csv("datasets/binary.csv")
```

```r
# Split the data into training (80%) and testing (20%) sets
set.seed(123)  # For reproducibility
split <- sample.split(mydata$admit, SplitRatio = 0.8)
train <- subset(mydata, split == TRUE)
test <- subset(mydata, split == FALSE)
```

```r
# Fit the neural network model with 3 hidden nodes
nn_3 <- neuralnet(admit ~ gre + gpa + rank, data = train, hidden = 3,
                  act.fct = "logistic", linear.output = FALSE)

# Plot the neural network
plot(nn_3)
```

```r
# Predict the outcomes for the test set
predict_3 <- compute(nn_3, test[,c("gre", "gpa", "rank")])
prob_3 <- predict_3$net.result
pred_3 <- ifelse(prob_3 > 0.5, 1, 0)
```

```r
# Create a confusion matrix
confusion_matrix_3 <- table(Actual = test$admit, Predicted = pred_3)

# Calculate accuracy
accuracy_3 <- sum(diag(confusion_matrix_3)) / sum(confusion_matrix_3)
print(paste("Accuracy with 3 hidden nodes:", accuracy_3))
```

```r
# Fit the model with 4 hidden nodes
nn_4 <- neuralnet(admit ~ gre + gpa + rank, data = train, hidden = 4,
                  act.fct = "logistic", linear.output = FALSE)
plot(nn_4)

# Predict, create confusion matrix, and calculate accuracy
predict_4 <- compute(nn_4, test[,c("gre", "gpa", "rank")])
prob_4 <- predict_4$net.result
pred_4 <- ifelse(prob_4 > 0.5, 1, 0)
confusion_matrix_4 <- table(Actual = test$admit, Predicted = pred_4)
accuracy_4 <- sum(diag(confusion_matrix_4)) / sum(confusion_matrix_4)
print(paste("Accuracy with 4 hidden nodes:", accuracy_4))
```