SRM Institute of Science and Technology

Department of Computer Applications

Delhi – Meerut Road, Sikri Kalan, Ghaziabad, Uttar Pradesh – 201204

Circular – 2020-21

MCA 2nd semester
DATA ANALYSIS USING R (PCA20S02J)

Lab Manual

Lab 1: Implementation of how to install R program and import packages

Title

Installation of R and Package Import

Aim

To understand the process of installing R and RStudio, and to learn how to install and load R packages.

Procedure

1. **Download and Install R:** Navigate to the official CRAN (Comprehensive R Archive Network) website and download the appropriate R installer for your operating system (Windows, macOS, Linux). Follow the installation prompts.
2. **Download and Install RStudio:** Go to the RStudio website and download RStudio Desktop (Open Source Edition). Install it following the on-screen instructions. RStudio is an Integrated Development Environment (IDE) that makes working with R much easier.
3. **Open RStudio:** Launch RStudio. You will see a console, script editor, environment pane, and plots/packages pane.
4. **Install a Package:** In the RStudio Console, use the `install.packages()` function to download and install a package from CRAN. For example, to install `ggplot2` (a popular visualization package), type `install.packages("ggplot2")` and press Enter.
5. **Load a Package:** After installation, use the `library()` function to load the package into your current R session, making its functions available. For `ggplot2`, type `library(ggplot2)` and press Enter.
6. **Verify Installation:** You can verify that the package is loaded by checking its version or trying to use one of its functions.

Source Code

```
# Step 4: Install a package (e.g., ggplot2)
# This command downloads and installs the package from CRAN.
# You only need to run this once per package on your system.
install.packages("ggplot2")

# Step 5: Load the installed package into the current R session
# This command makes the functions within ggplot2 available for use.
# You need to run this every time you start a new R session and want to use the
package.
library(ggplot2)

# Step 6: Verify installation by checking a function or package version
# This line will print the version of the ggplot2 package, confirming it's
loaded.
packageVersion("ggplot2")

# You can also try to access a function from the package, e.g.,
# If no error, it means the package is loaded.
# ?ggplot
```

Input

N/A (The commands are executed directly in the R console or script.)

Expected Output

Upon successful execution, you will see messages indicating the package download and installation progress. After loading the package, the `packageVersion()` command will output the version number of `ggplot2`, similar to:

```
'3.4.4'
```

(The exact version number may vary depending on when you install it.)

Lab 2: Implementation of R program - basic

Title

Basic R Program - Arithmetic Operations

Aim

To perform fundamental arithmetic operations and variable assignments in R.

Procedure

1. Open RStudio.
2. Use the R console as a calculator to perform basic arithmetic operations such as addition, subtraction, multiplication, and division directly.
3. Assign numeric values to variables using the `<-` (assignment operator) or `=` operator.
4. Perform arithmetic operations using these variables.
5. Use the `print()` function to display the results of variable operations.

Source Code

```
# Step 2: Basic arithmetic operations directly in the console
# Addition
2 + 3

# Subtraction
10 - 5

# Multiplication
4 * 6

# Division
20 / 4

# Step 3: Assign values to variables
# Assigning the value 15 to variable 'a'
a <- 15

# Assigning the value 7 to variable 'b'
b <- 7

# Step 4: Perform operations using variables
# Calculate the sum of 'a' and 'b'
sum_ab <- a + b

# Calculate the product of 'a' and 'b'
product_ab <- a * b

# Step 5: Print the results
print(sum_ab)
print(product_ab)
```

Input

N/A (The values are hardcoded within the script.)

Expected Output

The R console will display the results of each operation sequentially:

```
[1] 5
[1] 5
[1] 24
[1] 5
[1] 22
[1] 105
```

Lab 3: Implementation of R program - basic

Title

Basic R Program - Vector Operations

Aim

To understand and perform operations on vectors, which are fundamental data structures in R.

Procedure

1. Open RStudio.
2. Create numeric vectors using the `c()` function (combine function).
3. Create a character vector.
4. Perform element-wise arithmetic operations (e.g., addition, multiplication) on numeric vectors. R performs these operations on corresponding elements.
5. Access specific elements or subsets of a vector using square bracket `[]` indexing. R uses 1-based indexing.

Source Code

```
# Step 2: Create numeric vectors
# vec1 contains five numeric elements
vec1 <- c(10, 20, 30, 40, 50)

# vec2 contains five numeric elements
vec2 <- c(1, 2, 3, 4, 5)

# Step 3: Create a character vector
# fruits contains three character elements (strings)
fruits <- c("apple", "banana", "cherry")

# Step 4: Element-wise operations on numeric vectors
# Element-wise addition: (10+1), (20+2), ..., (50+5)
sum_vec <- vec1 + vec2
print("Element-wise sum:")
print(sum_vec)

# Element-wise multiplication: (10*1), (20*2), ..., (50*5)
prod_vec <- vec1 * vec2
print("Element-wise product:")
print(prod_vec)

# Step 5: Access elements of a vector
# Access the third element of vec1
print("Third element of vec1:")
print(vec1[3])

# Access the first and third elements of fruits
print("First and third elements of fruits:")
print(fruits[c(1, 3)])
```

Input

N/A (The vectors are defined within the script.)

Expected Output

The R console will display the results of vector operations and element access:

```
[1] "Element-wise sum:"
[1] 11 22 33 44 55
[1] "Element-wise product:"
[1]  10  40  90 160 250
[1] "Third element of vec1:"
[1] 30
[1] "First and third elements of fruits:"
[1] "apple"  "cherry"
```

Lab 4: Implementation of data types in R

Title

Understanding Data Types in R

Aim

To explore and demonstrate various fundamental data types in R, including numeric, integer, complex, character, and logical, and to understand type coercion.

Procedure

1. Open RStudio.
2. Declare variables and assign values corresponding to different R data types:
   - **Numeric:** Default for numbers (double-precision floating-point).
   - **Integer:** Explicitly declared by appending `L` to a number.
   - **Complex:** Numbers with an imaginary part.
   - **Character:** Text strings.
   - **Logical:** Boolean values (TRUE/FALSE).
3. Use the `class()` function to check the data type of each variable.
4. Demonstrate type coercion using `as.integer()`, `as.numeric()`, etc., to convert data from one type to another. Observe how R handles conversions, especially for incompatible types.

Source Code

```
# Step 2 & 3: Declare variables of different data types and check their class

# Numeric (default for numbers with decimal points or large numbers)
num_var <- 10.5
print(paste("Numeric:", num_var, "Class:", class(num_var)))

# Integer (explicitly declared by appending 'L' to ensure it's treated as an
integer)
int_var <- 10L
print(paste("Integer:", int_var, "Class:", class(int_var)))

# Complex (numbers with an imaginary component)
comp_var <- 3 + 2i
print(paste("Complex:", comp_var, "Class:", class(comp_var)))

# Character (strings of text)
char_var <- "Hello R Programming"
print(paste("Character:", char_var, "Class:", class(char_var)))

# Logical (boolean values: TRUE or FALSE)
log_var <- TRUE
print(paste("Logical:", log_var, "Class:", class(log_var)))

# Step 4: Demonstrate Type Coercion

# Coercing a numeric to an integer (decimal part is truncated)
coerced_int <- as.integer(num_var)
print(paste("Coerced Numeric to Integer:", coerced_int, "Class:",
class(coerced_int)))

# Coercing a character string that represents a number to numeric
```

```
char_to_num <- as.numeric("123.45")
print(paste("Coerced Character to Numeric:", char_to_num, "Class:",
class(char_to_num)))

# Coercing a character string that cannot be converted to numeric (results in NA
- Not Available)
char_to_num_fail <- as.numeric("abc")
print(paste("Coerced Character to Numeric (Fail):", char_to_num_fail, "Class:",
class(char_to_num_fail)))

# Coercing a logical to numeric (TRUE becomes 1, FALSE becomes 0)
log_to_num <- as.numeric(log_var)
print(paste("Coerced Logical to Numeric:", log_to_num, "Class:",
class(log_to_num)))
```

Input

N/A (Variables are defined and manipulated within the script.)

Expected Output

The R console will display the value and class of each variable before and after coercion:

```
[1] "Numeric: 10.5 Class: numeric"
[1] "Integer: 10 Class: integer"
[1] "Complex: 3+2i Class: complex"
[1] "Character: Hello R Programming Class: character"
[1] "Logical: TRUE Class: logical"
[1] "Coerced Numeric to Integer: 10 Class: integer"
[1] "Coerced Character to Numeric: 123.45 Class: numeric"
[1] "Coerced Character to Numeric (Fail): NA Class: numeric"
[1] "Coerced Logical to Numeric: 1 Class: numeric"
```

Lab 5: Implementation of Control Statements in R and KNN in R

Title

Control Statements and K-Nearest Neighbors (KNN) in R

Aim

To implement conditional logic using `if-else` and `switch` statements, and to perform a basic K-Nearest Neighbors (KNN) classification.

Procedure

1. Open RStudio.
2. **Control Statements:**
   - **If-Else:** Write an `if-else` statement to execute different blocks of code based on a specified condition (e.g., checking if a score is passing).
   - **Switch:** Implement a `switch` statement to select one of many code blocks to be executed based on the value of an expression (e.g., mapping a number to a day of the week).
3. **K-Nearest Neighbors (KNN):**
   - Install and load the `class` package, which provides the `knn()` function.
   - Create a small, simple dataset for demonstration, including features and a categorical label.
   - Define a new data point that you want to classify.
   - Use the `knn()` function, providing the training data features, the new data point's features, the training data labels, and the number of neighbors (`k`).
   - Print the predicted class for the new data point.

Source Code

```
# --- Part 1: Control Statements ---

# If-Else Statement: Checks a condition and executes code accordingly
score <- 75 # Example score
if (score >= 60) {
  print("Passed the exam.")
} else {
  print("Failed the exam.")
}

# Switch Statement: Selects one of many code blocks based on a value
day_num <- 3 # Example day number (1 for Monday, 2 for Tuesday, etc.)
day_name <- switch(day_num,
                   "Monday",
                   "Tuesday",
                   "Wednesday",
                   "Thursday",
                   "Friday",
                   "Saturday",
                   "Sunday")
print(paste("Today is:", day_name))

# --- Part 2: K-Nearest Neighbors (KNN) ---
# Install and load the 'class' package if not already installed
# 'quietly = TRUE' prevents messages from being printed if the package is
already installed
if (!requireNamespace("class", quietly = TRUE)) {
  install.packages("class")
}
library(class)

# Step 3: Sample data for KNN classification
# train_data: A data frame with features (x, y) and a categorical label (A or B)
train_data <- data.frame(
  x = c(1, 2, 3, 4, 5, 6),
  y = c(1, 2, 3, 4, 5, 6),
  label = factor(c("A", "A", "A", "B", "B", "B")) # 'factor' is important for
classification
)

# new_data: The data point we want to classify
new_data <- data.frame(x = 3.5, y = 3.5)

# Step 4: Perform KNN classification
# train: The training data features (excluding the label)
# test: The new data point features
# cl: The true classifications (labels) of the training set
# k: The number of nearest neighbors to consider
knn_prediction <- knn(train = train_data[, c("x", "y")],
                      test = new_data[, c("x", "y")],
                      cl = train_data$label,
                      k = 3) # Using k=3 neighbors

# Step 5: Print the prediction
print(paste("KNN Prediction for new data point:", knn_prediction))
```

Input

N/A (Values for control statements and KNN data are hardcoded within the script.)

Expected Output

The R console will display the results of the control statements and the KNN prediction:

```
[1] "Passed the exam."
[1] "Today is: Wednesday"
[1] "KNN Prediction for new data point: A"
```

(Note: The KNN prediction is 'A' because the new data point (3.5, 3.5) is closer to the 'A' labeled points (1,2,3) than the 'B' labeled points (4,5,6) when considering 3 neighbors in this simple linear example.)

Lab 6: Implementation of Looping Statements

Title

Implementation of Looping Statements in R

Aim

To demonstrate the use of `for`, `while`, and `repeat` loops in R for repetitive tasks.

Procedure

1. Open RStudio.
2. **For Loop:** Write a `for` loop to iterate over a sequence (e.g., numbers from 1 to 5) and perform an action (e.g., print the current number) for each element in the sequence.
3. **While Loop:** Implement a `while` loop that continues to execute a block of code as long as a specified condition remains true. Include a counter variable and an incrementing step to ensure the loop eventually terminates.
4. **Repeat Loop:** Use a `repeat` loop, which executes a block of code indefinitely until an explicit `break` statement is encountered. Include a condition within the loop that triggers the `break` to prevent an infinite loop.

Source Code

```
# --- Part 1: For Loop ---
# A 'for' loop is used to iterate over a sequence (e.g., a vector, list, or
range of numbers).
# In this example, it iterates through numbers from 1 to 5.
print("--- For Loop Output ---")
for (i in 1:5) {
  print(i) # Prints the current value of 'i' in each iteration
}

# --- Part 2: While Loop ---
# A 'while' loop continues to execute as long as a specified condition is TRUE.
# It's important to ensure the condition eventually becomes FALSE to avoid an
infinite loop.
print("--- While Loop Output ---")
count <- 1 # Initialize a counter variable
while (count <= 5) { # Loop continues as long as 'count' is less than or equal
to 5
  print(count)       # Prints the current value of 'count'
  count <- count + 1 # Increment 'count' to move towards the termination
condition
}

# --- Part 3: Repeat Loop ---
# A 'repeat' loop executes its body indefinitely until a 'break' statement is
encountered.
# You must explicitly include a 'break' condition within the loop.
print("--- Repeat Loop Output ---")
j <- 1 # Initialize a counter variable
repeat {
  print(j) # Prints the current value of 'j'
  j <- j + 1 # Increment 'j'
  if (j > 5) { # Condition to break the loop
    break    # Exits the 'repeat' loop when 'j' becomes greater than 5
  }
}
```

Input

N/A (The loops are self-contained with predefined iteration ranges or conditions.)

Expected Output

The R console will display the numbers from 1 to 5, three times, each under its respective loop heading:

```
[1] "--- For Loop Output ---"
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] "--- While Loop Output ---"
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] "--- Repeat Loop Output ---"
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

Lab 7: Implementation of Decision Tree

Title

Implementation of Decision Tree in R

Aim

To build and visualize a classification decision tree model using the `rpart` and `rpart.plot` packages in R, and to evaluate its performance.

Procedure

1. Open RStudio.
2. **Install and Load Packages:** Ensure you have the `rpart` (for building trees) and `rpart.plot` (for visualizing trees) packages installed and loaded.
3. **Load Dataset:** Load a suitable dataset for classification (e.g., the built-in `iris` dataset, which contains measurements of iris flowers and their species).
4. **Data Splitting:** Divide the dataset into training and testing sets. The training set is used to build the model, and the testing set is used to evaluate its performance on unseen data.
5. **Build Decision Tree:** Use the `rpart()` function to construct the decision tree model. Specify the formula (target variable ~ predictor variables) and `method = "class"` for classification trees.
6. **Visualize Tree:** Plot the generated decision tree using `rpart.plot()` for an intuitive graphical representation of the decision rules.
7. **Make Predictions:** Use the `predict()` function to make predictions on the test dataset using the trained model.
8. **Evaluate Model:** Create a confusion matrix to compare the actual species in the test set with the predicted species. Calculate the accuracy of the model.

Source Code

```
# Step 2: Install and load required packages
# rpart: Used for building recursive partitioning and regression trees (decision
trees).
# rpart.plot: Provides enhanced plotting capabilities for rpart objects.
if (!requireNamespace("rpart", quietly = TRUE)) {
  install.packages("rpart")
}
if (!requireNamespace("rpart.plot", quietly = TRUE)) {
  install.packages("rpart.plot")
}
library(rpart)
library(rpart.plot)

# Step 3: Load the iris dataset
# The iris dataset is a classic dataset for classification, containing
measurements
# of sepal length, sepal width, petal length, petal width, and species for 150
iris flowers.
data(iris)

# Set a seed for reproducibility
# This ensures that the random sampling for training/testing split is the same
every time you run the code.
```

```
set.seed(123)

# Step 4: Create training and testing sets
# We'll use 70% of the data for training and 30% for testing.
sample_index <- sample(nrow(iris), 0.7 * nrow(iris)) # Randomly select row
indices for training
train_data <- iris[sample_index, ] # Training data subset
test_data <- iris[-sample_index, ]  # Testing data subset (remaining 30%)

# Step 5: Build the decision tree model
# Formula: Species ~ . means predict 'Species' using all other columns as
predictors.
# method = "class" specifies that this is a classification tree (for categorical
target).
decision_tree_model <- rpart(Species ~ ., data = train_data, method = "class")

# Step 6: Plot the decision tree
# extra = 104: Displays the percentage of correct classification at each node.
# fallen.leaves = TRUE: Arranges the leaf nodes at the bottom of the plot.
# tweak = 1.2: Adjusts the text size for better readability.
print("--- Decision Tree Plot (will appear in RStudio Plots pane) ---")
rpart.plot(decision_tree_model, extra = 104, fallen.leaves = TRUE, tweak = 1.2)

# Step 7: Make predictions on the test data
# type = "class" ensures that the predictions are the predicted class labels
(e.g., "setosa").
predictions <- predict(decision_tree_model, test_data, type = "class")

# Step 8: Evaluate the model using a confusion matrix and accuracy
# A confusion matrix shows the number of correct and incorrect predictions for
each class.
confusion_matrix <- table(Actual = test_data$Species, Predicted = predictions)
print("--- Confusion Matrix ---")
print(confusion_matrix)

# Calculate accuracy: (Number of correct predictions) / (Total number of
predictions)
accuracy <- sum(diag(confusion_matrix)) / sum(confusion_matrix)
print(paste("Accuracy:", round(accuracy, 4))) # Round accuracy to 4 decimal
places
```

Input

N/A (The script uses the built-in `iris` dataset.)

Expected Output

1. A graphical representation of the decision tree will be displayed in the RStudio Plots pane.
2. The R console will output the confusion matrix and the calculated accuracy. The exact values in the confusion matrix might vary slightly due to the random splitting of data, but generally, high accuracy is expected for the `iris` dataset with a decision tree.

Example Console Output:

```
[1] "--- Decision Tree Plot (will appear in RStudio Plots pane) ---"
[1] "--- Confusion Matrix ---"
            Predicted
Actual       setosa versicolor virginica
  setosa         15          0         0
  versicolor      0         14         1
  virginica       0          1        14
[1] "Accuracy: 0.9778"
```

Lab 8: Implementation of Naïve Bayes

Title

Implementation of Naïve Bayes Classifier in R

Aim

To implement and evaluate a Naïve Bayes classification model using the `e1071` package in R.

Procedure

1. Open RStudio.
2. **Install and Load Package:** Ensure the `e1071` package (which contains the `naiveBayes()` function) is installed and loaded.
3. **Load Dataset:** Load a suitable dataset for classification (e.g., the built-in `iris` dataset).
4. **Data Splitting:** Divide the dataset into training and testing sets to train the model and then evaluate its performance on unseen data.
5. **Build Naïve Bayes Model:** Use the `naiveBayes()` function to construct the Naïve Bayes classifier. Specify the formula (target variable ~ predictor variables).
6. **Make Predictions:** Use the `predict()` function to make class predictions on the test dataset.
7. **Evaluate Model:** Create a confusion matrix to compare the actual classes in the test set with the predicted classes. Calculate the accuracy of the model.

Source Code

```
# Step 2: Install and load the 'e1071' package
# This package provides the implementation for various machine learning
algorithms,
# including Naïve Bayes.
if (!requireNamespace("e1071", quietly = TRUE)) {
  install.packages("e1071")
}
library(e1071)

# Step 3: Load the iris dataset
# The iris dataset is commonly used for demonstrating classification algorithms.
data(iris)

# Set a seed for reproducibility
# Ensures that the random split of data into training and testing sets is
consistent.
set.seed(123)

# Step 4: Create training and testing sets
# Splitting the data into 70% for training and 30% for testing.
sample_index <- sample(nrow(iris), 0.7 * nrow(iris)) # Randomly select row
indices
train_data <- iris[sample_index, ] # Training data
test_data <- iris[-sample_index, ]  # Testing data

# Step 5: Build the Naïve Bayes model
# Formula: Species ~ . means predict 'Species' using all other columns as
predictors.
# The target variable 'Species' should be a factor.
```

```
naive_bayes_model <- naiveBayes(Species ~ ., data = train_data)

print("--- Naïve Bayes Model Summary ---")
print(naive_bayes_model) # Prints the model details (prior probabilities,
conditional probabilities)

# Step 6: Make predictions on the test data
# The 'predict' function applies the trained model to the test data.
predictions <- predict(naive_bayes_model, test_data)

# Step 7: Evaluate the model using a confusion matrix and accuracy
# A confusion matrix helps assess the performance of a classification model.
confusion_matrix <- table(Actual = test_data$Species, Predicted = predictions)
print("--- Confusion Matrix ---")
print(confusion_matrix)

# Calculate accuracy: Sum of diagonal elements (correct predictions) divided by
total predictions.
accuracy <- sum(diag(confusion_matrix)) / sum(confusion_matrix)
print(paste("Accuracy:", round(accuracy, 4))) # Round accuracy to 4 decimal
places
```

Input

N/A (The script uses the built-in `iris` dataset.)

Expected Output

The R console will display the summary of the Naïve Bayes model, the confusion matrix, and the calculated accuracy. The exact values in the confusion matrix might vary slightly due to the random splitting of data.

Example Console Output:

```
[1] "--- Naïve Bayes Model Summary ---"

Naive Bayes Classifier for Discrete Predictors

Call:
naiveBayes.default(x = X, y = Y)

A-priori probabilities:
Y
    setosa versicolor  virginica
 0.3333333  0.3333333  0.3333333

Conditional probabilities:
             Sepal.Length
Y                  [,1]      [,2]
  setosa       5.006 0.3524887
  versicolor   5.936 0.5161711
  virginica    6.588 0.6358772
... (truncated for brevity, full output shows all conditional probabilities)

[1] "--- Confusion Matrix ---"
            Predicted
Actual       setosa versicolor virginica
  setosa         15          0         0
  versicolor      0         14         1
  virginica       0          1        14
[1] "Accuracy: 0.9778"
```

Lab 9: Implementation of Random Forest in R

Title

Implementation of Random Forest in R

Aim

To build and evaluate a Random Forest classification model using the `randomForest` package in R.

Procedure

1. Open RStudio.
2. **Install and Load Package:** Ensure the `randomForest` package is installed and loaded.
3. **Load Dataset:** Load a suitable dataset for classification (e.g., the built-in `iris` dataset).
4. **Data Preparation:** Ensure the target variable is a factor, as Random Forest in R expects categorical target variables for classification.
5. **Data Splitting:** Divide the dataset into training and testing sets to train the model and then evaluate its performance on unseen data.
6. **Build Random Forest Model:** Use the `randomForest()` function to construct the Random Forest classifier. Specify the formula (target variable ~ predictor variables), the number of trees (`ntree`), and the number of variables to randomly sample at each split (`mtry`).
7. **Make Predictions:** Use the `predict()` function to make class predictions on the test dataset.
8. **Evaluate Model:** Create a confusion matrix to compare the actual classes in the test set with the predicted classes. Calculate the accuracy of the model.

Source Code

```
# Step 2: Install and load the 'randomForest' package
# This package provides an efficient implementation of the Random Forest
algorithm.
if (!requireNamespace("randomForest", quietly = TRUE)) {
  install.packages("randomForest")
}
library(randomForest)

# Step 3: Load the iris dataset
# The iris dataset is a standard benchmark for classification tasks.
data(iris)

# Step 4: Ensure the target variable 'Species' is a factor
# Random Forest classification requires the target variable to be a factor.
iris$Species <- as.factor(iris$Species)

# Set a seed for reproducibility
# This ensures that the random split of data and the random nature of Random
Forest
# (e.g., bootstrap sampling, feature sampling) are consistent across runs.
set.seed(123)

# Step 5: Create training and testing sets
# Splitting the data into 70% for training and 30% for testing.
sample_index <- sample(nrow(iris), 0.7 * nrow(iris)) # Randomly select row
indices
train_data <- iris[sample_index, ] # Training data
```

```
test_data <- iris[-sample_index, ]  # Testing data

# Step 6: Build the Random Forest model
# Formula: Species ~ . means predict 'Species' using all other columns.
# ntree = 500: Specifies the number of trees to grow in the forest. More trees
generally lead to better performance.
# mtry = sqrt(ncol(train_data) - 1): Specifies the number of variables randomly
sampled as candidates at each split.
# For classification, a common heuristic is sqrt(number of features).
random_forest_model <- randomForest(Species ~ ., data = train_data, ntree = 500,
mtry = sqrt(ncol(train_data) - 1))

print("--- Random Forest Model Summary ---")
print(random_forest_model) # Prints details like OOB (Out-Of-Bag) error estimate
and confusion matrix for training data.

# Step 7: Make predictions on the test data
# The 'predict' function applies the trained Random Forest model to the test
data.
predictions <- predict(random_forest_model, test_data)

# Step 8: Evaluate the model using a confusion matrix and accuracy
# A confusion matrix helps assess the performance of the classifier on unseen
data.
confusion_matrix <- table(Actual = test_data$Species, Predicted = predictions)
print("--- Confusion Matrix (Test Data) ---")
print(confusion_matrix)

# Calculate accuracy: Sum of diagonal elements (correct predictions) divided by
total predictions.
accuracy <- sum(diag(confusion_matrix)) / sum(confusion_matrix)
print(paste("Accuracy:", round(accuracy, 4))) # Round accuracy to 4 decimal
places
```

Input

N/A (The script uses the built-in `iris` dataset.)

Expected Output

The R console will display a summary of the Random Forest model (including its OOB error), the confusion matrix for the test data, and the calculated accuracy. Random Forest typically achieves very high accuracy on the `iris` dataset.

Example Console Output:

```
[1] "--- Random Forest Model Summary ---"

Call:
 randomForest(formula = Species ~ ., data = train_data, ntree = 500,      mtry =
sqrt(ncol(train_data) - 1))
               Type of random forest: classification
                     Number of trees: 500
No. of variables tried at each split: 2

        OOB estimate of  error rate: 3.81%
Confusion matrix:
           setosa versicolor virginica class.error
setosa         35          0         0  0.00000000
versicolor      0         33         1  0.02941176
virginica       0          3        33  0.08333333

[1] "--- Confusion Matrix (Test Data) ---"
```

```
            Predicted
Actual       setosa versicolor virginica
  setosa         15          0         0
  versicolor      0         15         0
  virginica       0          0        15
[1] "Accuracy: 1"
```

(Note: The accuracy of 1 is common for Random Forest on the Iris dataset due to its effectiveness.)

Lab 10: Implementation of K means

Title

Implementation of K-Means Clustering in R

Aim

To perform K-Means clustering on a dataset and visualize the resulting clusters.

Procedure

1. Open RStudio.
2. **Load Dataset:** Load a suitable dataset for clustering (e.g., the built-in `iris` dataset). For clustering, we typically exclude the known class labels.
3. **Select Features:** Choose the numeric features from the dataset that will be used for clustering.
4. **Set Seed:** Set a random seed for reproducibility, as K-Means involves random initialization of centroids.
5. **Perform K-Means:** Use the `kmeans()` function, specifying the data to cluster, the desired number of clusters (`centers`), and `nstart` (number of random sets to choose for initial centroids).
6. **Add Cluster Assignments:** Add the cluster assignments generated by K-Means back to the original dataset for easier analysis and visualization.
7. **Visualize Clusters:** Use `ggplot2` to create a scatter plot of two selected features, coloring the points by their assigned cluster to visually inspect the clustering result.

Source Code

```
# Step 2: Load the iris dataset
# The iris dataset is often used for clustering demonstrations.
data(iris)

# Step 3: Select features for clustering
# For clustering, we typically exclude the known class label ('Species' column)
# as K-Means is an unsupervised learning algorithm.
iris_features <- iris[, -5] # Exclude the 5th column (Species)

# Step 4: Set a seed for reproducibility
# K-Means algorithm involves random initialization of cluster centroids.
# Setting a seed ensures that the results are consistent across multiple runs.
set.seed(123)

# Step 5: Perform K-Means clustering
# kmeans(): The function for K-Means clustering.
# iris_features: The data to be clustered.
# centers = 3: We specify 3 clusters, as we know there are 3 species in the iris
dataset.
# nstart = 20: Runs the algorithm 20 times with different random starting
centroids
#               and chooses the best result (minimizing within-cluster sum of
squares).
kmeans_result <- kmeans(iris_features, centers = 3, nstart = 20)

print("--- K-Means Clustering Result ---")
print(kmeans_result) # Prints the details of the clustering result
```

```
# Step 6: Add cluster assignments to the original data
# This adds a new column to the iris data frame indicating which cluster each
observation belongs to.
iris$cluster <- as.factor(kmeans_result$cluster) # Convert cluster numbers to
factors for plotting

# Step 7: Visualize the clusters
# Install and load 'ggplot2' if not already installed
if (!requireNamespace("ggplot2", quietly = TRUE)) {
  install.packages("ggplot2")
}
library(ggplot2)

# Create a scatter plot using Sepal.Length and Sepal.Width, colored by the
assigned cluster.
print("--- K-Means Cluster Plot (will appear in RStudio Plots pane) ---")
ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width, color = cluster)) +
  geom_point(size = 3) + # Plot points with a size of 3
  labs(title = "K-Means Clusters of Iris Data", # Title of the plot
       x = "Sepal Length (cm)", y = "Sepal Width (cm)") + # Axis labels
  theme_minimal() # Minimal theme for a clean look
```

Input

N/A (The script uses the built-in `iris` dataset.)

Expected Output

1. The R console will display a summary of the K-Means clustering result, including the cluster centers, the number of points in each cluster, and the within-cluster sum of squares.
2. A scatter plot showing the iris data points, with each point colored according to its assigned cluster, will appear in the RStudio Plots pane.

Example Console Output:

```
[1] "--- K-Means Clustering Result ---"
K-means clustering with 3 clusters of sizes 50, 48, 52

Cluster means:
  Sepal.Length Sepal.Width Petal.Length Petal.Width
1     5.006000    3.428000     1.462000    0.246000
2     5.901633    2.748980     4.393878    1.432653
3     6.850000    3.073077     5.748077    2.071154

Clustering vector:
  [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1
 [51] 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2
[101] 3 2 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3 3 3 3

Within cluster sum of squares by cluster:
[1] 15.15100 39.82097 23.89966
 (between_SS / total_SS =  88.5 %)

Available components:

[1] "cluster"      "centers"      "totss"        "withinss"     "tot.withinss"
"betweenss"
[7] "size"         "iter"         "ifault"
```

```
[1] "--- K-Means Cluster Plot (will appear in RStudio Plots pane) ---"
```

Lab 11: Implementation of medoids

Title

Implementation of K-Medoids (PAM) Clustering in R

Aim

To perform K-Medoids clustering using the Partitioning Around Medoids (PAM) algorithm with the `cluster` package in R.

Procedure

1. Open RStudio.
2. **Install and Load Package:** Ensure the `cluster` package (which provides the `pam()` function) is installed and loaded.
3. **Load Dataset:** Load a suitable dataset for clustering (e.g., the built-in `iris` dataset). As with K-Means, exclude the known class labels.
4. **Select Features:** Choose the numeric features from the dataset that will be used for clustering.
5. **Set Seed:** Set a random seed for reproducibility, as PAM also involves random sampling in its initial steps.
6. **Perform K-Medoids (PAM):** Use the `pam()` function, specifying the data to cluster and the desired number of clusters (`k`). PAM identifies actual data points as cluster centers (medoids), making it more robust to outliers than K-Means.
7. **Add Cluster Assignments:** Add the cluster assignments generated by PAM back to the original dataset.
8. **Visualize Clusters:** Use `ggplot2` to create a scatter plot of two selected features, coloring the points by their assigned cluster to visually inspect the clustering result.

Source Code

```
# Step 2: Install and load the 'cluster' package
# This package provides functions for various clustering algorithms, including
PAM.
if (!requireNamespace("cluster", quietly = TRUE)) {
  install.packages("cluster")
}
library(cluster)

# Step 3: Load the iris dataset
data(iris)

# Step 4: Select features for clustering
# Exclude the 'Species' column as it's the target variable in classification,
# and clustering is an unsupervised task.
iris_features <- iris[, -5]

# Step 5: Set a seed for reproducibility
# Setting a seed ensures that the results of PAM are consistent across multiple
runs.
set.seed(123)

# Step 6: Perform PAM (K-Medoids) clustering
# pam(): The function for Partitioning Around Medoids.
# iris_features: The data to be clustered.
# k = 3: We specify 3 clusters, aligning with the known species in the iris
dataset.
pam_result <- pam(iris_features, k = 3)

print("--- PAM (K-Medoids) Clustering Result ---")
print(pam_result) # Prints the details of the clustering result, including
medoids.

# Step 7: Add cluster assignments to the original data
# 'pam_result$clustering' contains the cluster assignment for each observation.
iris$cluster_pam <- as.factor(pam_result$clustering) # Convert to factor for
plotting

# Step 8: Visualize the clusters
# Install and load 'ggplot2' if not already installed
if (!requireNamespace("ggplot2", quietly = TRUE)) {
  install.packages("ggplot2")
}
library(ggplot2)

# Create a scatter plot using Sepal.Length and Sepal.Width, colored by the
assigned PAM cluster.
print("--- PAM Cluster Plot (will appear in RStudio Plots pane) ---")
ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width, color = cluster_pam)) +
  geom_point(size = 3) + # Plot points with a size of 3
  labs(title = "K-Medoids (PAM) Clusters of Iris Data", # Title of the plot
       x = "Sepal Length (cm)", y = "Sepal Width (cm)") + # Axis labels
  theme_minimal() # Minimal theme for a clean look
```

Input

N/A (The script uses the built-in `iris` dataset.)

Expected Output

1. The R console will display a summary of the PAM clustering result, including the medoids (actual data points that are cluster centers), the clustering vector, and various statistics like silhouette width.
2. A scatter plot showing the iris data points, with each point colored according to its assigned PAM cluster, will appear in the RStudio Plots pane.

Example Console Output:

```
[1] "--- PAM (K-Medoids) Clustering Result ---"
Medoids:
        ID Sepal.Length Sepal.Width Petal.Length Petal.Width
setosa  42          4.5         2.3          1.3         0.3
... (truncated for brevity, full output shows medoids and other details)

Clustering vector:
  [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1
 [51] 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2
[101] 3 2 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3 3 3 3

... (further details like silhouette information)

[1] "--- PAM Cluster Plot (will appear in RStudio Plots pane) ---"
```

Lab 12: Implementation of Hierarchical with R

Title

Implementation of Hierarchical Clustering in R

Aim

To perform hierarchical clustering on a dataset and visualize the resulting dendrogram.

Procedure

1. Open RStudio.
2. **Load Dataset:** Load a suitable dataset for clustering (e.g., the built-in `iris` dataset), excluding the class labels.
3. **Calculate Distance Matrix:** Compute the pairwise distances between observations in the dataset using a distance metric (e.g., Euclidean distance). This is a prerequisite for hierarchical clustering.
4. **Perform Hierarchical Clustering:** Apply the `hclust()` function to the distance matrix. Specify an agglomeration method (e.g., "ward.D2" for Ward's method, which minimizes the total within-cluster variance).
5. **Plot Dendrogram:** Visualize the hierarchical clustering result as a dendrogram using the `plot()` function. A dendrogram graphically represents the merges or divisions at each step of the clustering process.
6. **Cut Dendrogram:** Use the `cutree()` function to cut the dendrogram at a specific height or to obtain a desired number of clusters.
7. **Add Cluster Assignments (Optional):** Add the hierarchical cluster assignments to the original dataset for further analysis or visualization.
8. **Visualize Clusters (Optional):** Use `ggplot2` to create a scatter plot of two selected features, coloring the points by their assigned hierarchical cluster.

Source Code

```
# Step 2: Load the iris dataset
data(iris)

# Step 3: Select features for clustering
# Exclude the 'Species' column as it's the target variable in classification.
iris_features <- iris[, -5]

# Step 4: Calculate the distance matrix
# dist(): Computes a distance matrix between rows of a data matrix.
# method = "euclidean": Specifies the Euclidean distance as the metric.
distance_matrix <- dist(iris_features, method = "euclidean")

# Step 5: Perform hierarchical clustering
# hclust(): Performs hierarchical cluster analysis.
# method = "ward.D2": Specifies Ward's method for agglomeration.
#                     It minimizes the total within-cluster variance.
h_cluster_result <- hclust(distance_matrix, method = "ward.D2")

# Step 6: Plot the dendrogram
# The dendrogram visually represents the hierarchical structure of the clusters.
print("--- Hierarchical Clustering Dendrogram (will appear in RStudio Plots
pane) ---")
```

```
plot(h_cluster_result,
     main = "Dendrogram of Iris Data (Hierarchical Clustering)", # Main title
     xlab = "Data Points", ylab = "Distance") # Axis labels

# Step 7: Cut the dendrogram to form a specified number of clusters
# cutree(): Cuts a dendrogram into several groups.
# k = 3: Specifies that we want to obtain 3 clusters.
num_clusters <- 3
cluster_assignments <- cutree(h_cluster_result, k = num_clusters)

print(paste("--- Cluster assignments for", num_clusters, "clusters ---"))
print(table(cluster_assignments)) # Displays the count of observations in each
cluster

# Step 8 (Optional): Add cluster assignments to the original data for
visualization
iris$h_cluster <- as.factor(cluster_assignments)

# Step 9 (Optional): Visualize clusters on a scatter plot
# Install and load 'ggplot2' if not already installed
if (!requireNamespace("ggplot2", quietly = TRUE)) {
  install.packages("ggplot2")
}
library(ggplot2)

print("--- Hierarchical Cluster Plot (will appear in RStudio Plots pane) ---")
ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width, color = h_cluster)) +
  geom_point(size = 3) +
  labs(title = "Hierarchical Clusters of Iris Data",
       x = "Sepal Length (cm)", y = "Sepal Width (cm)") +
  theme_minimal()
```

Input

N/A (The script uses the built-in `iris` dataset.)

Expected Output

1. A dendrogram plot showing the hierarchical clustering structure will appear in the RStudio Plots pane.
2. The R console will display a table showing the number of observations assigned to each of the 3 clusters.
3. An optional scatter plot showing data points colored by their hierarchical cluster will appear in the RStudio Plots pane.

Example Console Output:

```
[1] "--- Hierarchical Clustering Dendrogram (will appear in RStudio Plots pane)
---"
[1] "--- Cluster assignments for 3 clusters ---"
cluster_assignments
 1  2  3
50 52 48
[1] "--- Hierarchical Cluster Plot (will appear in RStudio Plots pane) ---"
```

Lab 13: Implementation of data visualization in R

Title

Basic Data Visualization in R

Aim

To create fundamental data visualizations like histograms, box plots, and scatter plots using both base R graphics and the `ggplot2` package.

Procedure

1. Open RStudio.
2. **Load Dataset:** Load a built-in dataset (e.g., `mtcars`, which contains data on various car models).
3. **Base R Graphics:**
   - **Histogram:** Use `hist()` to visualize the distribution of a single numeric variable.
   - **Box Plot:** Use `boxplot()` to compare the distribution of a numeric variable across different categories.
   - **Scatter Plot:** Use `plot()` to examine the relationship between two numeric variables.
4. **ggplot2 Graphics:**
   - Install and load the `ggplot2` package.
   - Recreate a scatter plot using `ggplot2`, mapping variables to aesthetics (x, y, color) and adding geometric objects (`geom_point`). `ggplot2` offers more flexibility and aesthetic control.

Source Code

```
# Step 2: Load a built-in dataset
# mtcars: Contains data about various car models, including miles per gallon
(mpg),
# horsepower (hp), weight (wt), and number of cylinders (cyl).
data(mtcars)

# --- Part 1: Base R Graphics ---

# Step 3.1: Histogram of 'mpg' (miles per gallon)
# Shows the distribution of a single continuous variable.
print("--- Histogram of MPG (Base R) (will appear in RStudio Plots pane) ---")
hist(mtcars$mpg,
     main = "Histogram of Miles Per Gallon", # Main title of the plot
     xlab = "Miles Per Gallon (mpg)",        # X-axis label
     col = "skyblue",                        # Bar color
     border = "black")                       # Border color of bars

# Step 3.2: Box plot of 'mpg' by 'cyl' (number of cylinders)
# Compares the distribution (median, quartiles, outliers) of a numeric variable
# across different categories.
print("--- Box Plot of MPG by Cylinders (Base R) (will appear in RStudio Plots
pane) ---")
boxplot(mpg ~ cyl, data = mtcars, # Formula: response ~ predictor (categorical)
        main = "MPG by Number of Cylinders",
        xlab = "Number of Cylinders",
        ylab = "Miles Per Gallon (mpg)",
        col = c("lightgreen", "lightblue", "lightcoral")) # Colors for each box

# Step 3.3: Scatter plot of 'hp' (horsepower) vs 'wt' (weight)
# Shows the relationship between two continuous variables.
print("--- Scatter Plot of HP vs WT (Base R) (will appear in RStudio Plots pane)
---")
plot(mtcars$hp, mtcars$wt,
     main = "Horsepower vs Weight",
     xlab = "Horsepower (hp)",
     ylab = "Weight (wt)",
     pch = 16, # Plotting character (solid circle)
     col = "darkblue") # Color of points

# --- Part 2: ggplot2 Graphics ---

# Step 4: Install and load 'ggplot2' if not already installed
# ggplot2 is a powerful and flexible package for creating elegant data
visualizations.
if (!requireNamespace("ggplot2", quietly = TRUE)) {
  install.packages("ggplot2")
}
library(ggplot2)

# ggplot2 Scatter plot of 'hp' vs 'wt' with 'cyl' mapped to color
# This demonstrates how to map a third variable (cyl) to an aesthetic (color).
print("--- Scatter Plot of HP vs WT (ggplot2) (will appear in RStudio Plots
pane) ---")
ggplot(mtcars, aes(x = hp, y = wt, color = factor(cyl))) + # Define aesthetics
(x, y, color)
  geom_point(size = 3) + # Add points layer
  labs(title = "Horsepower vs Weight by Cylinders", # Plot title
       x = "Horsepower (hp)", y = "Weight (wt)",    # Axis labels
       color = "Cylinders") +                       # Legend title for color
  theme_minimal() # Use a minimal theme for a clean look
```

Input

N/A (The script uses the built-in `mtcars` dataset.)

Expected Output

Multiple plots will be generated and displayed sequentially in the RStudio Plots pane:

1. A histogram showing the distribution of `mpg`.
2. A box plot comparing `mpg` for different numbers of cylinders.
3. A scatter plot showing the relationship between `hp` and `wt` (Base R).
4. A scatter plot showing the relationship between `hp` and `wt`, with points colored by `cyl` (ggplot2).

The R console will indicate when each plot is being generated.

Lab 14: Implementation of various charts

Title

Implementation of Various Charts in R

Aim

To create different types of charts, including bar charts, pie charts, and line charts, using `ggplot2` (and base R for pie chart simplicity).

Procedure

1. Open RStudio.
2. **Install and Load Package:** Ensure the `ggplot2` package is installed and loaded for creating modern and customizable charts.
3. **Bar Chart:**
   - Create sample data representing categorical values and their corresponding frequencies or aggregates.
   - Use `ggplot()` with `geom_bar(stat = "identity")` to create a bar chart, mapping the categorical variable to the x-axis and the aggregate value to the y-axis.
4. **Pie Chart:**
   - While `ggplot2` can create pie charts, it often involves more complex transformations. For simplicity, demonstrate a basic pie chart using base R's `pie()` function.
   - Prepare data as a numeric vector for values and a character vector for labels.
5. **Line Chart:**
   - Create sample data representing a trend over time or an ordered category.
   - Use `ggplot()` with `geom_line()` and `geom_point()` to create a line chart, mapping the ordered variable to the x-axis and the quantitative variable to the y-axis.

Source Code

```
# Step 2: Install and load 'ggplot2' if not already installed
if (!requireNamespace("ggplot2", quietly = TRUE)) {
  install.packages("ggplot2")
}
library(ggplot2)

# --- Part 1: Bar Chart ---
# Step 3: Sample data for bar chart (e.g., sales data by region)
sales_data <- data.frame(
  Region = c("North", "South", "East", "West"),
  Sales = c(150, 200, 120, 180)
)

print("--- Bar Chart of Sales by Region (will appear in RStudio Plots pane) ---
")
ggplot(sales_data, aes(x = Region, y = Sales, fill = Region)) + # Map Region to
x, Sales to y, and Region to fill color
  geom_bar(stat = "identity") + # Use 'identity' stat as we are providing the y-
values directly
  labs(title = "Total Sales by Region",
       x = "Region", y = "Sales Amount") +
  theme_minimal() + # Use a minimal theme
  theme(legend.position = "none") # Hide the legend as regions are already on
the x-axis

# --- Part 2: Pie Chart (using Base R for simplicity) ---
# Step 4: Prepare data for pie chart
# Base R's pie() function is straightforward for simple pie charts.
print("--- Pie Chart of Sales by Region (Base R) (will appear in RStudio Plots
pane) ---")
pie(sales_data$Sales,
    labels = paste0(sales_data$Region, " (", sales_data$Sales, ")"), # Labels
with region and sales value
    main = "Sales Distribution by Region", # Main title
    col = rainbow(length(sales_data$Region))) # Colors for slices

# --- Part 3: Line Chart ---
# Step 5: Sample data for line chart (e.g., a simple time series or trend)
time_series_data <- data.frame(
  Month = 1:6,
  Value = c(100, 110, 105, 120, 115, 130)
)

print("--- Line Chart of Value Over Time (will appear in RStudio Plots pane) ---
")
ggplot(time_series_data, aes(x = Month, y = Value)) + # Map Month to x, Value to
y
  geom_line(color = "darkgreen", size = 1.2) + # Add line layer, set color and
thickness
  geom_point(color = "darkgreen", size = 3) + # Add points at data observations
  labs(title = "Value Trend Over Months",
       x = "Month", y = "Value") +
  theme_minimal() +
  scale_x_continuous(breaks = 1:6) # Ensure all month numbers are shown on the
x-axis
```

Input

N/A (The script uses sample data defined within the code.)

Expected Output

Multiple plots will be generated and displayed sequentially in the RStudio Plots pane:

1. A bar chart showing sales by region.
2. A pie chart showing the distribution of sales by region.
3. A line chart illustrating a value trend over months.

The R console will indicate when each plot is being generated.

Lab 15: Implementation of predictive model in R

Title

Implementation of a Predictive Model (Linear Regression) in R

Aim

To build and evaluate a simple linear regression model to predict a continuous variable based on another continuous variable, and to visualize the regression line.

Procedure

1. Open RStudio.
2. **Load Dataset:** Load a suitable dataset containing at least two numeric variables (e.g., the built-in `mtcars` dataset).
3. **Define Variables:** Identify the independent (predictor) and dependent (response) variables for your regression model.
4. **Build Linear Regression Model:** Use the `lm()` function to build the linear regression model. Specify the model formula (`response_variable ~ predictor_variable`).
5. **Summarize Model:** Use `summary()` on the linear model object to view detailed statistics, including coefficients, R-squared, and p-values, which indicate the model's fit and the significance of predictors.
6. **Make Predictions:** Use the `predict()` function to make predictions on new data points using the trained model.
7. **Visualize Regression Line:** Use `ggplot2` to create a scatter plot of the actual data points and overlay the fitted regression line, providing a visual representation of the model.

Source Code

```
# Step 2: Load a built-in dataset
# mtcars: Contains data on various car models, including miles per gallon (mpg)
# and horsepower (hp). We'll use these for linear regression.
data(mtcars)

# Set a seed for reproducibility (optional for simple lm, but good practice)
set.seed(123)

# Step 3: Define independent (predictor) and dependent (response) variables
# We aim to predict 'mpg' (Miles Per Gallon) based on 'hp' (Horsepower).
# Model formula: response_variable ~ predictor_variable
model_formula <- mpg ~ hp

# Step 4: Build the linear regression model
# lm(): The function for fitting linear models.
# The formula specifies the relationship, and 'data' specifies the data frame.
linear_model <- lm(model_formula, data = mtcars)

# Step 5: Summarize the model
# summary(): Provides a detailed summary of the linear model, including:
# - Coefficients (intercept and slope for hp)
# - Standard errors, t-values, and p-values for coefficients
# - R-squared and Adjusted R-squared (measures of how well the model fits the
data)
# - F-statistic and its p-value (overall significance of the model)
print("--- Linear Regression Model Summary ---")
print(summary(linear_model))

# Step 6: Make predictions using the model
# Create new data points for which we want to predict 'mpg'.
# The column name in 'newdata' must match the predictor variable name in the
model formula ('hp').
new_data_for_prediction <- data.frame(hp = c(100, 150, 200))
predictions <- predict(linear_model, newdata = new_data_for_prediction)

print("--- Predictions for new data points ---")
print(predictions)

# Step 7: Visualize the regression line
# Install and load 'ggplot2' if not already installed
if (!requireNamespace("ggplot2", quietly = TRUE)) {
  install.packages("ggplot2")
}
library(ggplot2)

print("--- Scatter Plot with Regression Line (will appear in RStudio Plots pane)
---")
ggplot(mtcars, aes(x = hp, y = mpg)) + # Map hp to x-axis, mpg to y-axis
  geom_point(size = 3, color = "darkblue") + # Add scatter plot points
  geom_smooth(method = "lm", col = "red", se = FALSE) + # Add the linear
regression line
                                                         # 'se=FALSE' removes the
confidence interval shading
  labs(title = "MPG vs Horsepower with Regression Line", # Plot title
       x = "Horsepower (hp)", y = "Miles Per Gallon (mpg)") + # Axis labels
  theme_minimal() # Use a minimal theme for a clean look
```

Input

N/A (The script uses the built-in `mtcars` dataset and hardcoded new data for prediction.)

Expected Output

1. The R console will display a comprehensive summary of the linear regression model, including statistical details about the coefficients, R-squared, and model significance.
2. The R console will then show the predicted `mpg` values for the new horsepower inputs.
3. A scatter plot showing the `mtcars` data points with the fitted red regression line overlaid will appear in the RStudio Plots pane.

Example Console Output:

```
[1] "--- Linear Regression Model Summary ---"

Call:
lm(formula = mpg ~ hp, data = mtcars)

Residuals:
    Min      1Q  Median      3Q     Max
-5.7121 -2.1122 -0.8854  1.5819  8.2392

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 30.09886    1.63392  18.421  < 2e-16 ***
hp          -0.06823    0.01012  -6.742 1.79e-07 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 3.863 on 30 degrees of freedom
```

```
Multiple R-squared:  0.6024,	Adjusted R-squared:  0.5892 
F-statistic: 45.46 on 1 and 30 DF,  p-value: 1.788e-07

[1] "--- Predictions for new data points ---"
       1        2        3 
23.27559 20.06371 16.85183 
[1] "--- Scatter Plot with Regression Line (will appear in RStudio Plots pane) -
--"
```