PERIYAR UNIVERSITY

(NAAC 'A++' Grade with CGPA 3.61 (Cycle - 3)
State University - NIRF Rank 56 – State Public University Rank 25
SALEM - 636 011

CENTRE FOR DISTANCE AND ONLINE EDUCATION

(CDOE)

MASTER OF BUSINESS ADMINSTRATION

SEMESTER - III

SPECIALIZATION COURSES : DATA ANALYTICS WITH R PROGRAMMING

(Candidates admitted from 2025 onwards)

PERIYAR UNIVERSITY

CENTRE FOR DISTANCE AND ONLINE EDUCATION (CDOE)

M.B.A 2025 admission onwards

SPECIALIZATION COURSESS
Data Analytics with R Programming

Prepared by:

Dr. H. Hannah Inbarani
Professor
Department of Computer Science
Periyar University
Salem - 636011

SYLLABUS

DATA ANALYTICS WITH R PROGRAMMING

Overview of R programming - Environment setup with R Studio - SAS versus R - R, S, and S-plus - Obtaining and managing R - Objects - types of objects, classes, creating and accessing objects - Arithmetic and matrix operations - Introduction to functions.
Working with R - Reading and writing data - R libraries - Functions and R programming – the If statement - looping: for, repeat, while - writing functions - function arguments and options – Basic R commands
Reading and getting data into R (External Data): Using CSV files, XML files, Web Data, JSON files, Databases, Excel files. Working with R Charts and Graphs: Histograms, Boxplots, Bar Charts, Line Graphs, Scatterplots, Pie Charts
Random Forest, Decision Tree, Normal and Binomial distributions, Time Series Analysis, Linear and Multiple Regression, Logistic Regression, Survival Analysis.
Creating data for analytics through designed experiments, Creating data for analytics through active learning, Creating data for analytics through reinforcement learning.

TABLE OF CONTENTS

DATA ANALYTICS WITH R PROGRAMMING

UNIT 1 – INTRODUCTION

Overview of R programming - Environment setup with R Studio - SAS versus R - R, S, and S-plus - Obtaining and managing R - Objects - types of objects, classes, creating and accessing objects - Arithmetic and matrix operations - Introduction to functions.

OVERVIEW OF R PROGRAMMING

UNIT OBJECTIVES

This unit aims to provide a comprehensive introduction to R programming, focusing on the foundational elements required for effective data analysis and statistical computing. It begins by familiarizing students with the basics of R, including its purpose, advantages, and typical applications in data science. Learners will be guided through the environment setup, involving the installation of R and RStudio, and will become comfortable navigating and using RStudio as the primary development interface. The unit also offers a comparative understanding of SAS and R, enabling students to appreciate the open-source flexibility of R in contrast to the commercial nature of SAS. Additionally, the historical evolution of R from the S and S-Plus languages is discussed to provide context on its development and capabilities.

Further, students will learn how to obtain, install, and manage the R software and its packages from CRAN repositories. The unit introduces the concept of objects in R, covering various types such as vectors, matrices, lists, and data frames. It explains the notion of object classes and demonstrates how to create, access, and manipulate these objects effectively. Essential arithmetic operations and matrix-based computations are covered to lay a strong mathematical foundation. Finally, the unit introduces functions in R, including how to use built-in functions and create custom functions to perform repetitive or complex tasks, thus promoting modular and efficient coding practices.

SECTION 1.1: OVERVIEW OF R PROGRAMMING

R is a powerful, open-source programming language and software environment primarily used for statistical computing, data analysis, and graphical representation. It was developed in the early 1990s by Ross Ihaka and Robert Gentleman at the University of Auckland, New Zealand, and is considered a free and open-source implementation of the S programming language developed at Bell Laboratories. Unlike many general-purpose languages, R was specifically built with data analysis and statistics in mind, making it highly suitable for tasks such as data cleaning, manipulation, modeling, and visualization. Over time, R has evolved into a vital tool in fields like bioinformatics,

economics, machine learning, social sciences, and business analytics due to its flexibility and strong statistical capabilities.

One of the most significant strengths of R lies in its extensibility through packages. Thousands of packages are available via CRAN (Comprehensive R Archive Network), covering a wide range of functionalities, from linear modeling and time series analysis to advanced machine learning algorithms and deep learning frameworks. Popular packages like ggplot2 for visualization, dplyr for data manipulation, caret for machine learning, and shiny for building interactive web apps have further boosted R's reputation in the data science community. R is also known for its ability to generate high-quality graphical plots and reports, which are essential in academic research and industry presentations.

Another key advantage of R is its active and vibrant community. Being open-source, R is constantly updated and supported by developers and researchers across the globe. This community-driven approach ensures that R stays up to date with the latest statistical methods and computational tools. R supports integration with other languages like C, C++, Python, and Java, enabling users to leverage the strengths of multiple platforms within a single analysis pipeline. Despite its many advantages, R does have some limitations—it can be memory-intensive, and for very large datasets, it may perform slower compared to languages like Python or Java. Additionally, R's syntax and steep learning curve may initially challenge new users.

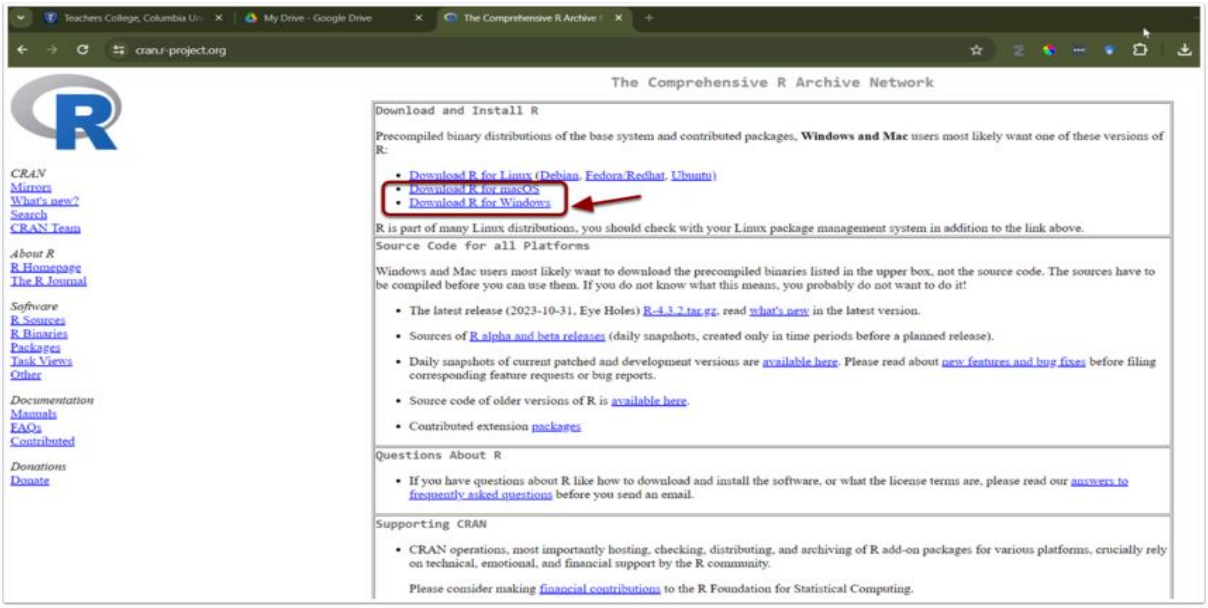
Feature	R	Python	SAS
Type	Statistical language	General-purpose	Proprietary software
Cost	Free & open-source	Free & open-source	Commercial (paid)
Data handling	Strong	Strong	Strong
Graphics	Excellent	Good (via libraries)	Moderate
Community	Strong and active	Very strong	Limited

1.1.2 – ENVIRONMENT SETUP WIITH R STUDIO

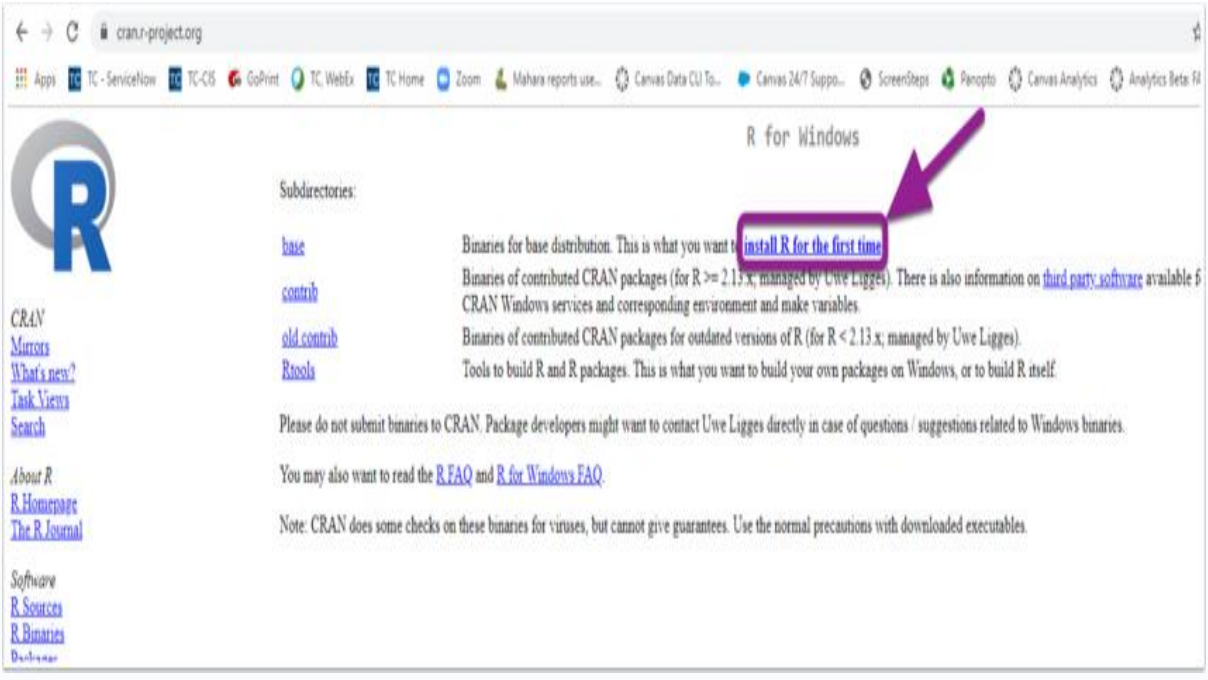
1. To install R, go to cran.r-project.org

cran.r-project.org

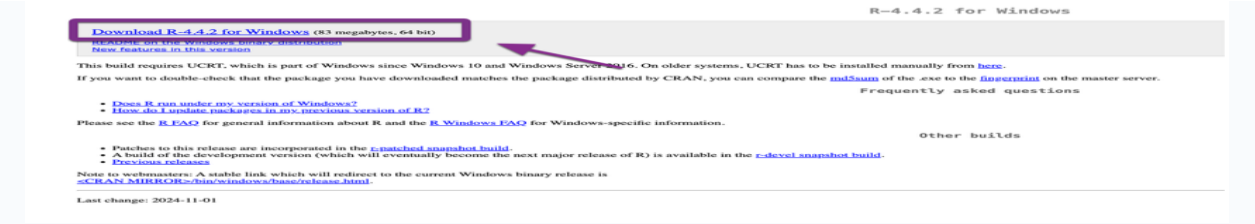
2. Choose for Download R for Windows



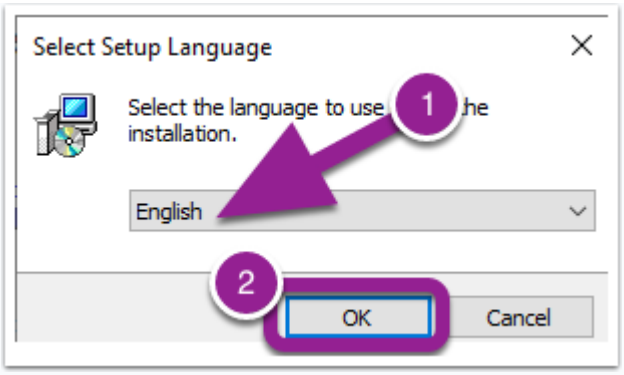
3. Install R Click on install R for the first time.



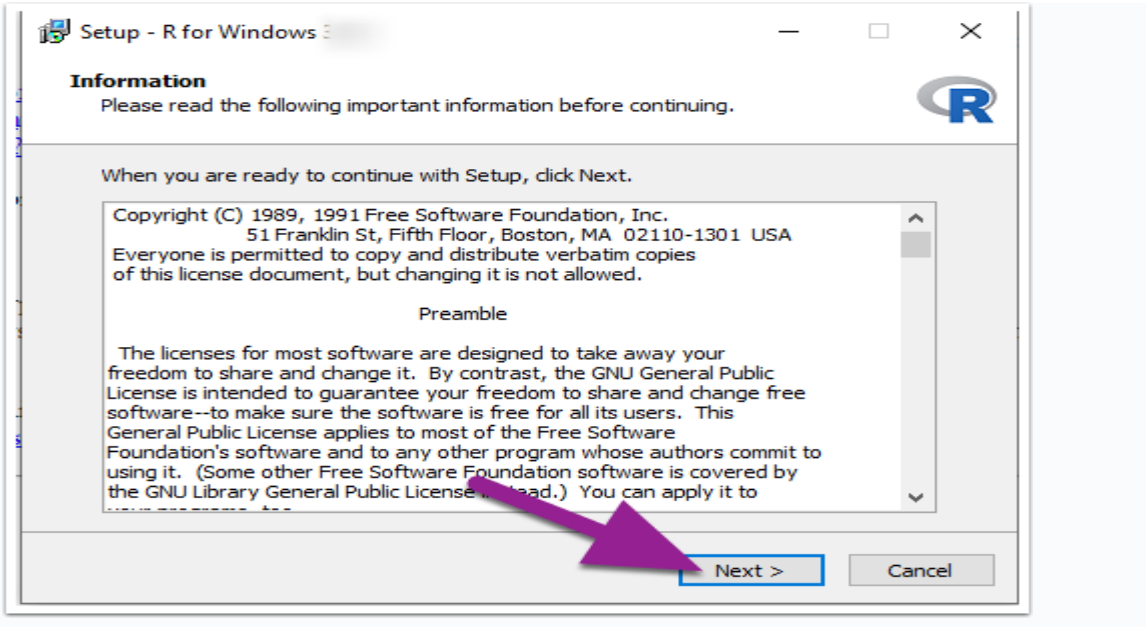
4. Click Download R for Windows. Open the downloaded file.



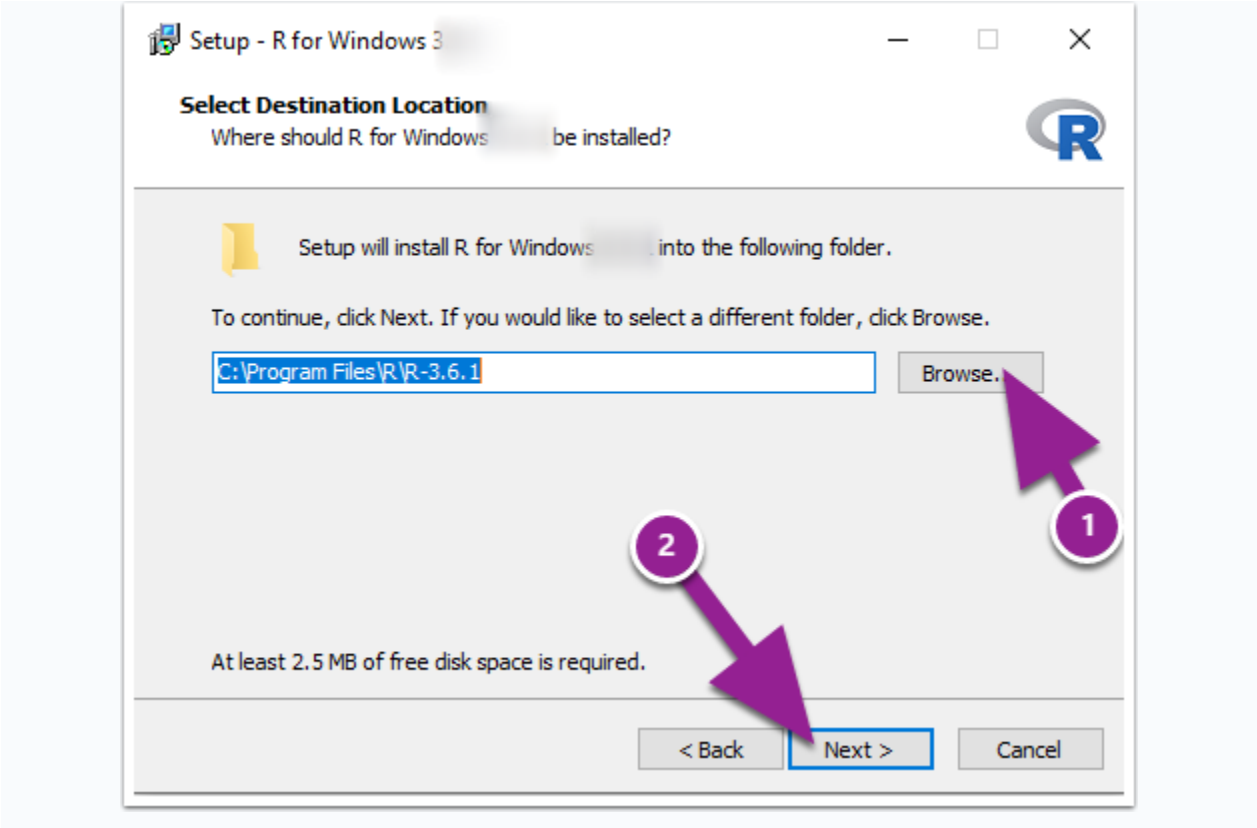
5. Select the language you would like to use during the installation. Then click OK.



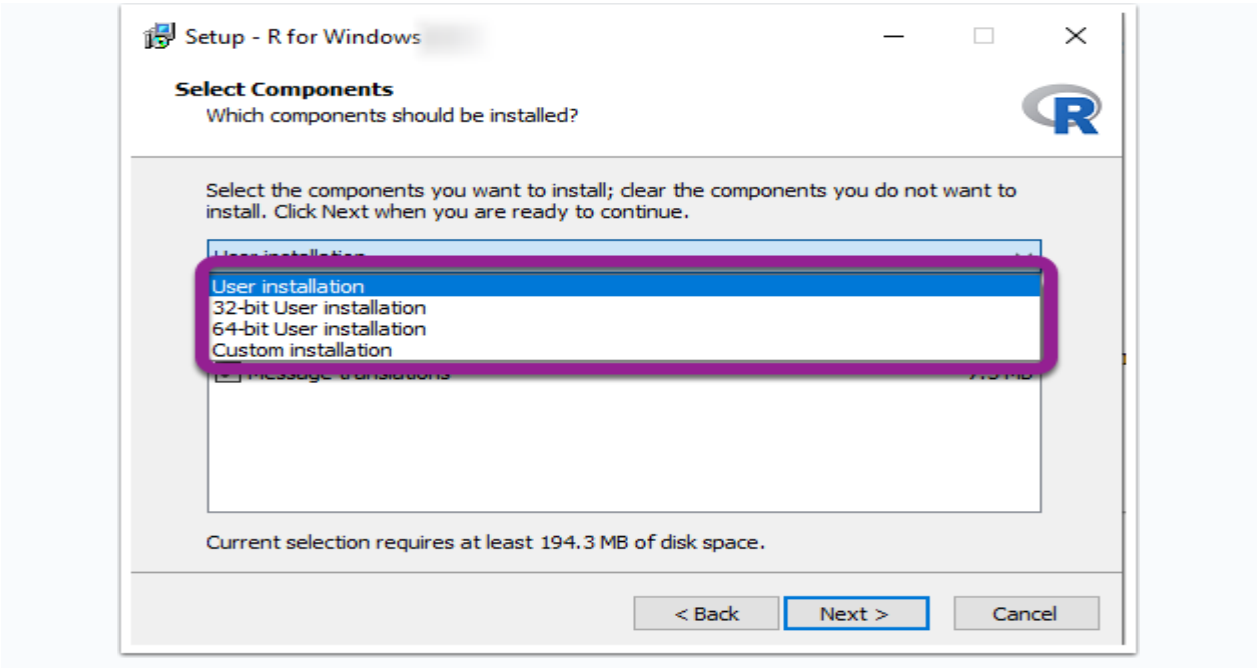
6. Click Next.



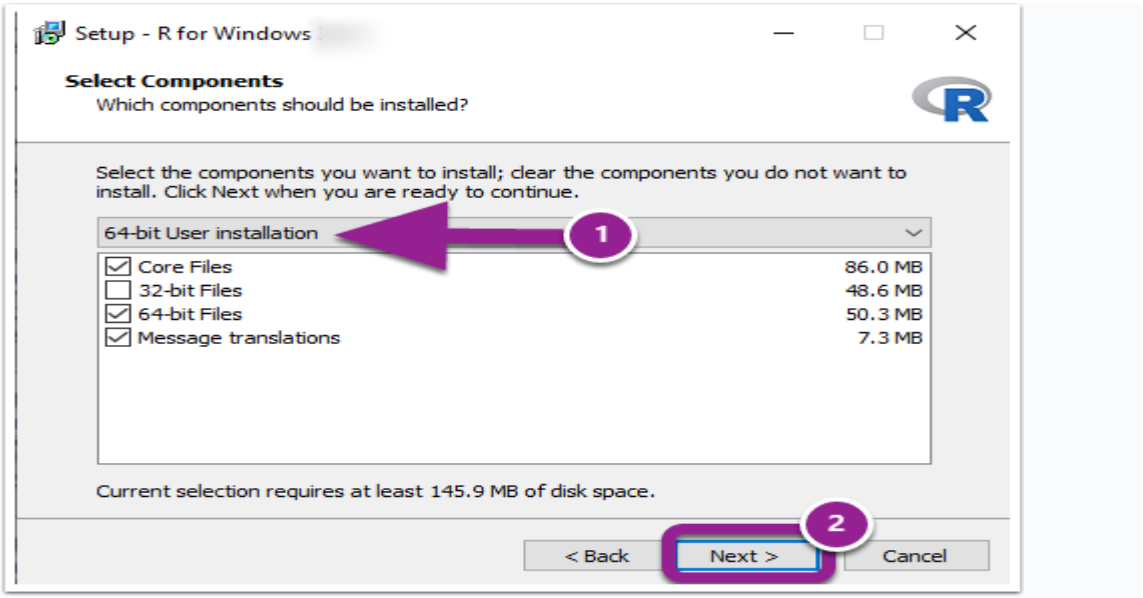
7. Select where you would like R to be installed. It will default to your Program Files on your C Drive. Click Next.



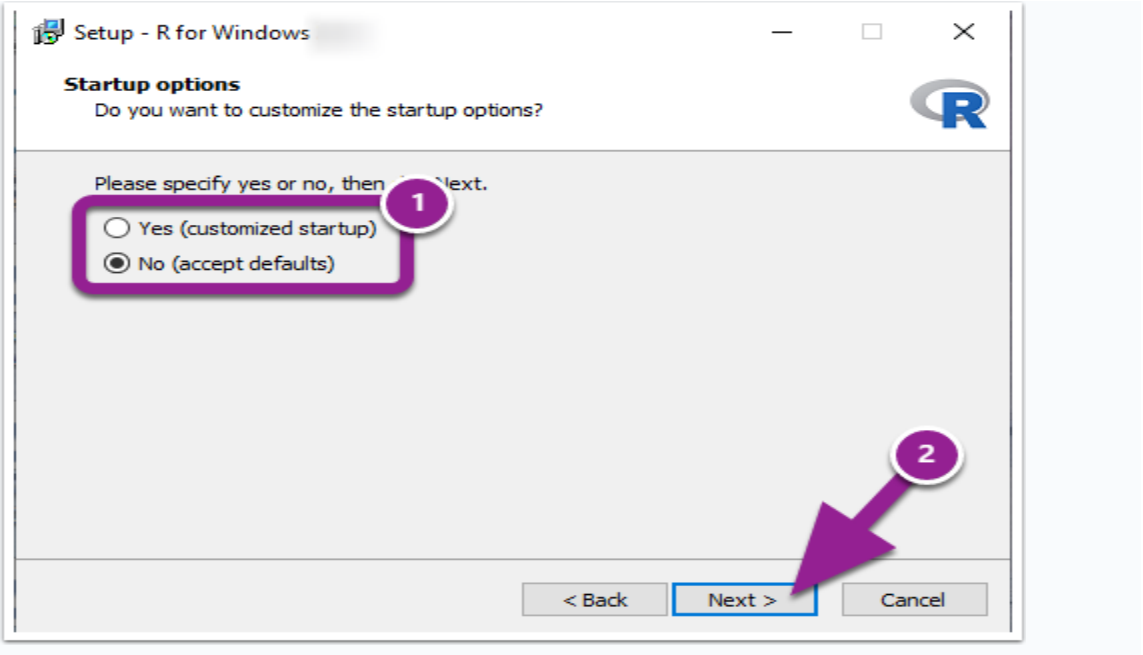
8. You can then choose which installation you would like.



9. (Optional) If your computer is a 64-bit, you can choose the 64-bit User Installation. Then click Next.

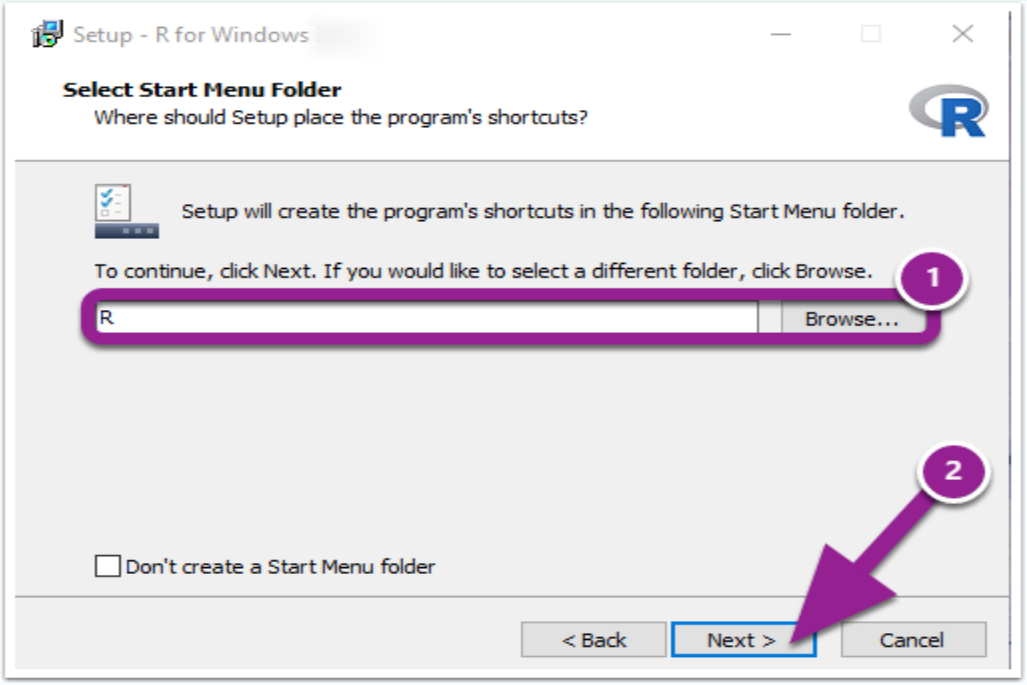


10. Then specify if you want to customized your startup or just use the defaults. Then click Next.

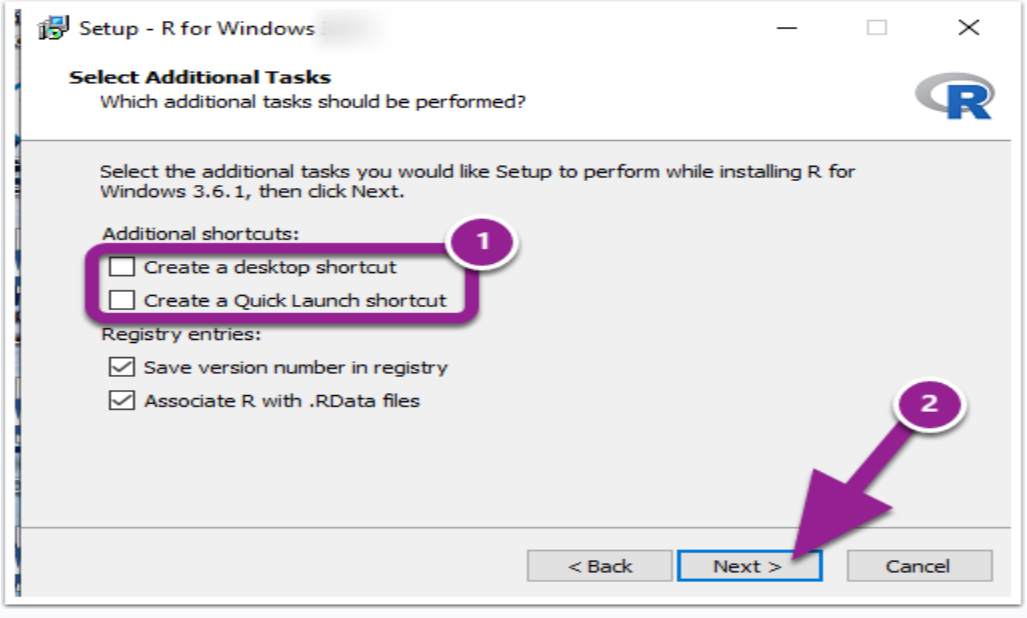


11. Then you can choose the folder that you want R to be saved within or the default if the R folder that was created. Once you have finished, click Next.

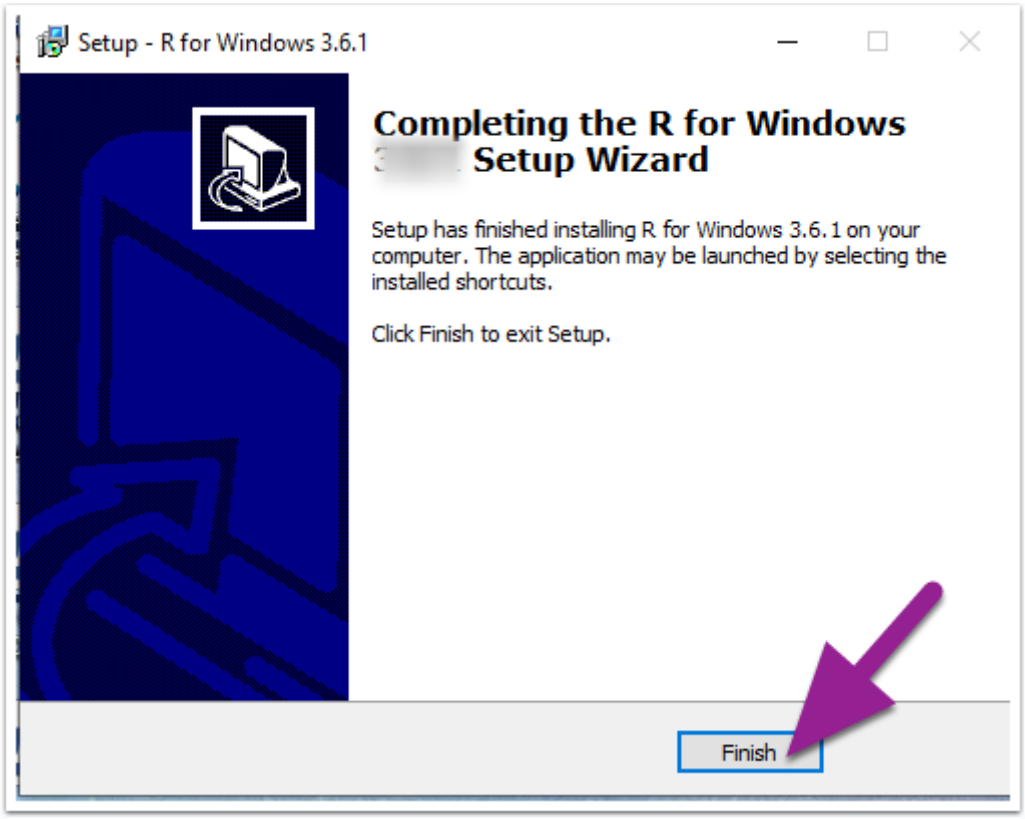
You can also choose if you do not want a Start Menu folder at the bottom.



12. You can then select additional shortcuts if you would like. Click Next.

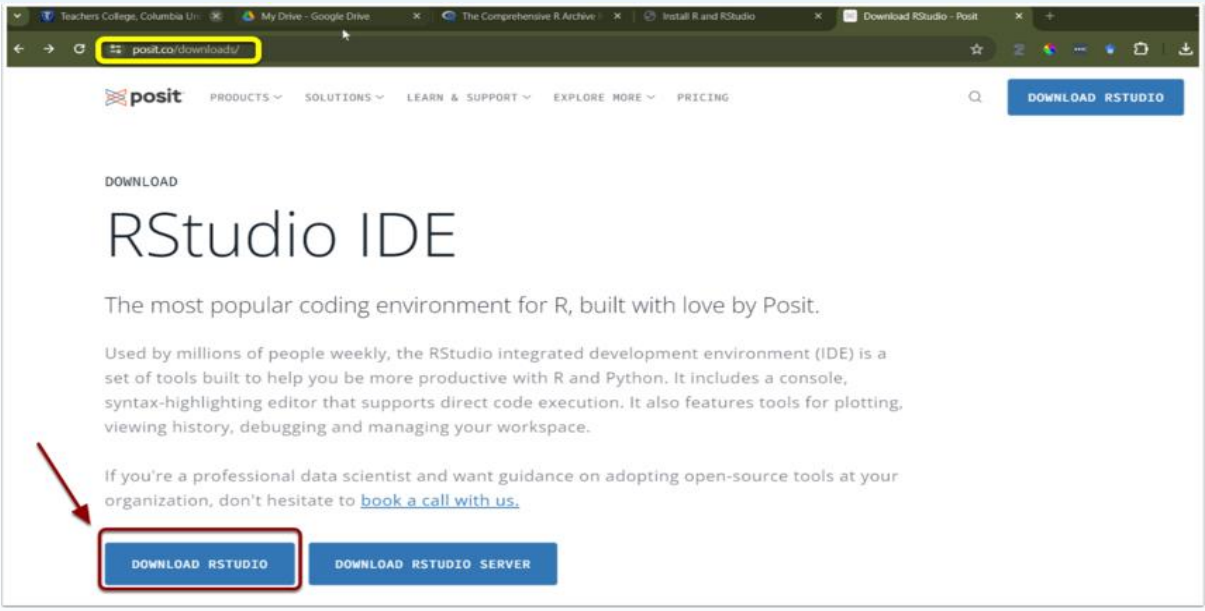


13. Click Finish.

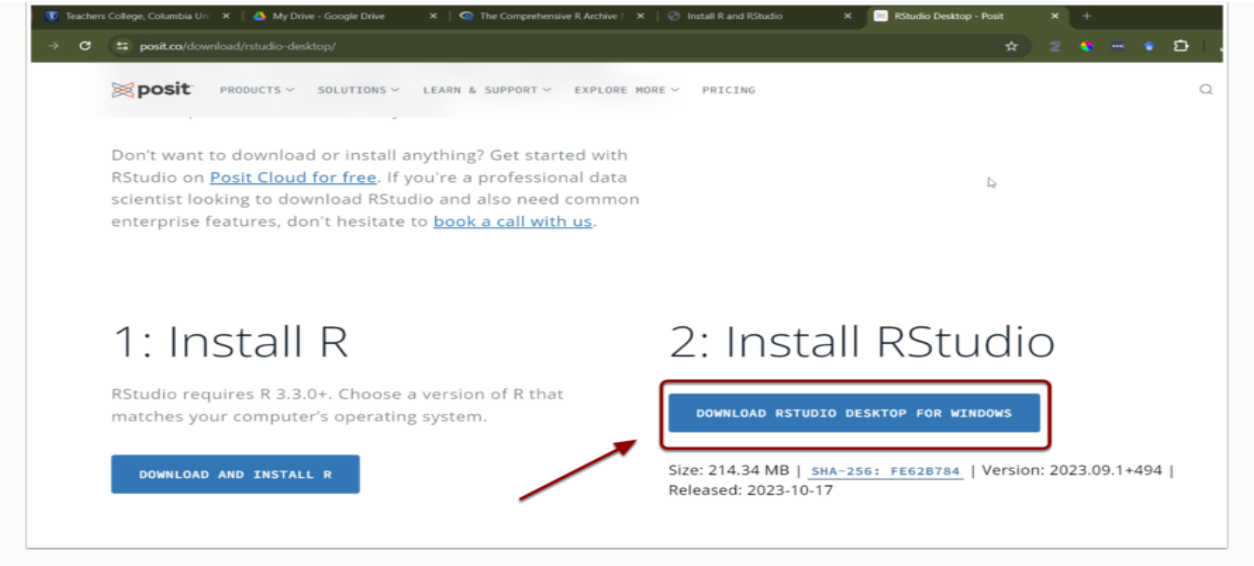


14. Next, download RStudio. Go to https://posit.co/downloads/

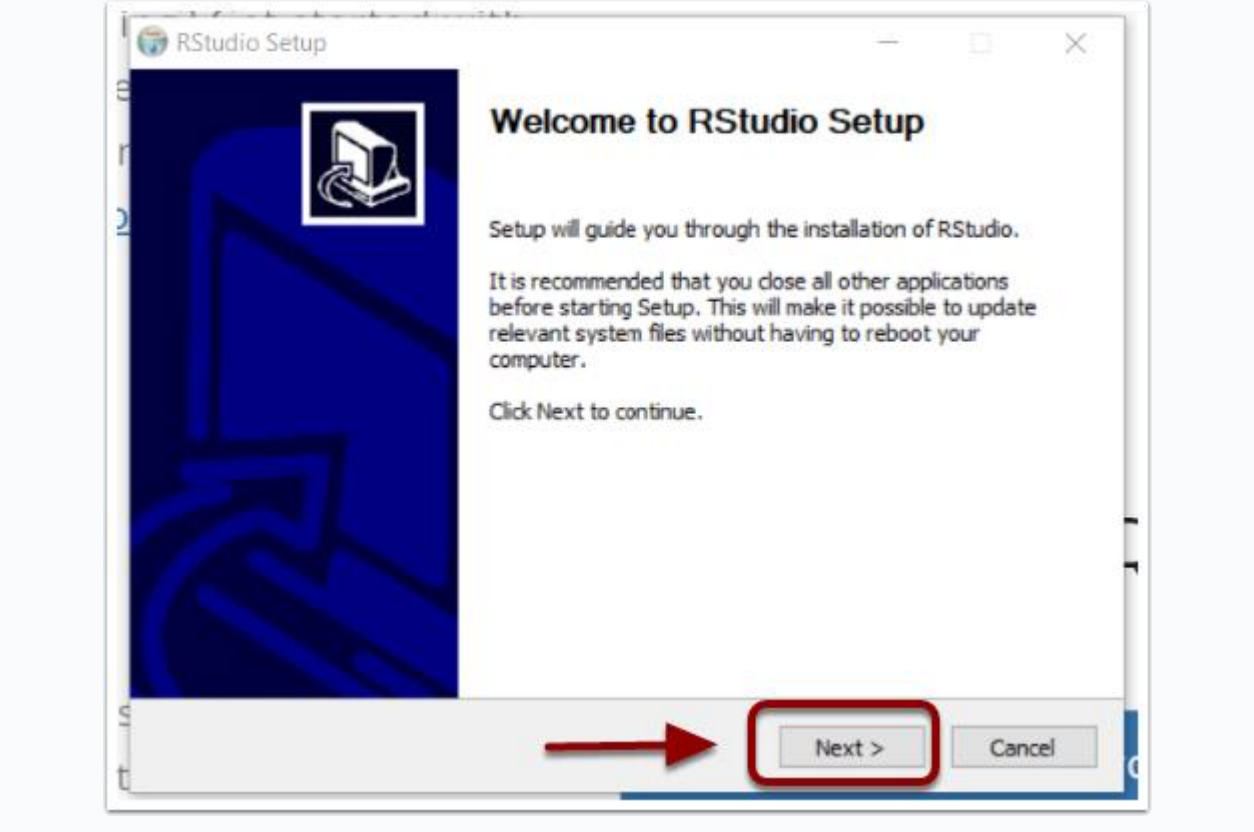
https://posit.co/downloads/



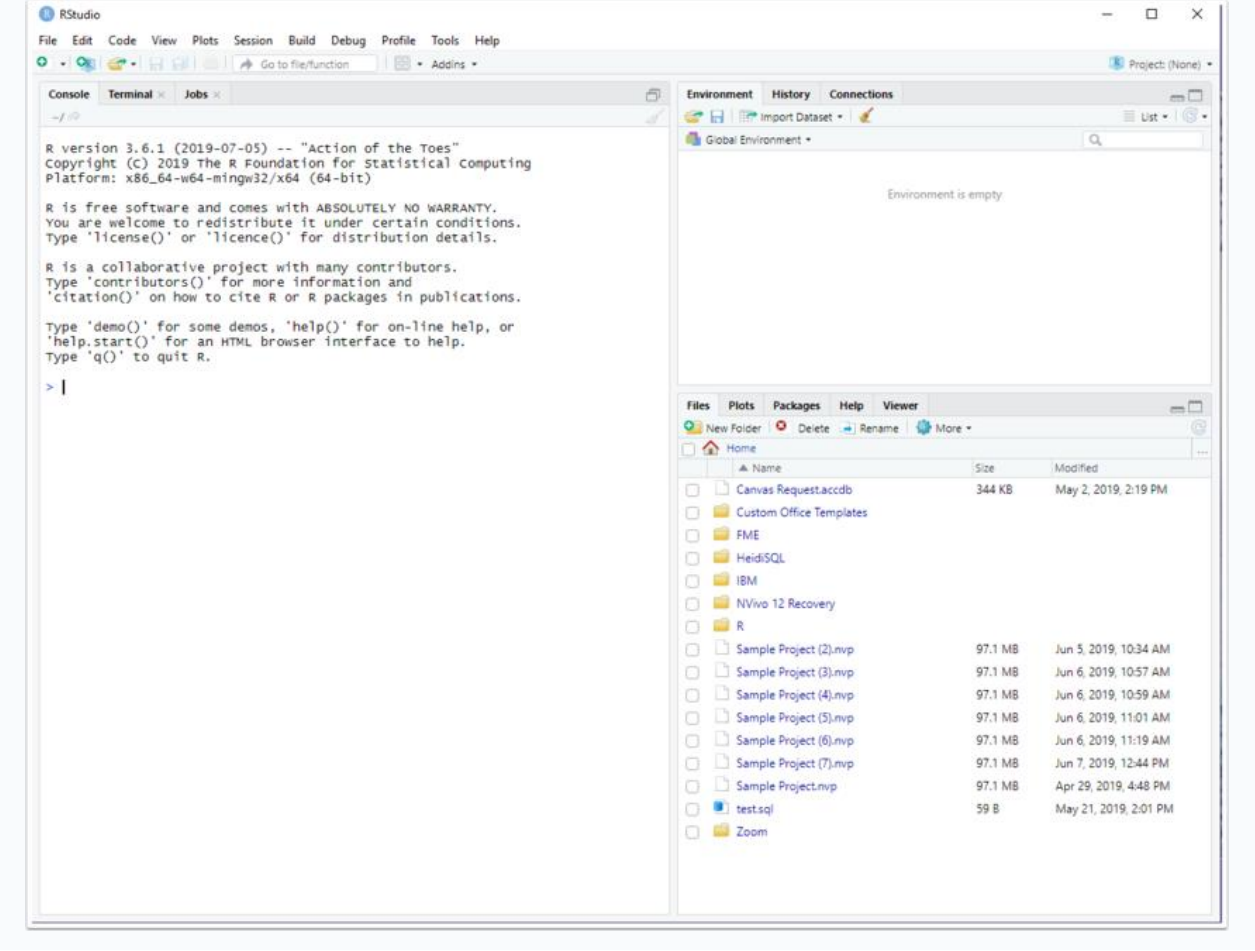
15. Click Download RStudio.



16. Once the packet has downloaded, the Welcome to RStudio Setup Wizard will open. Click Next and go through the installation steps.



17. After the Setup Wizard finishing the installation, RStudio will open.



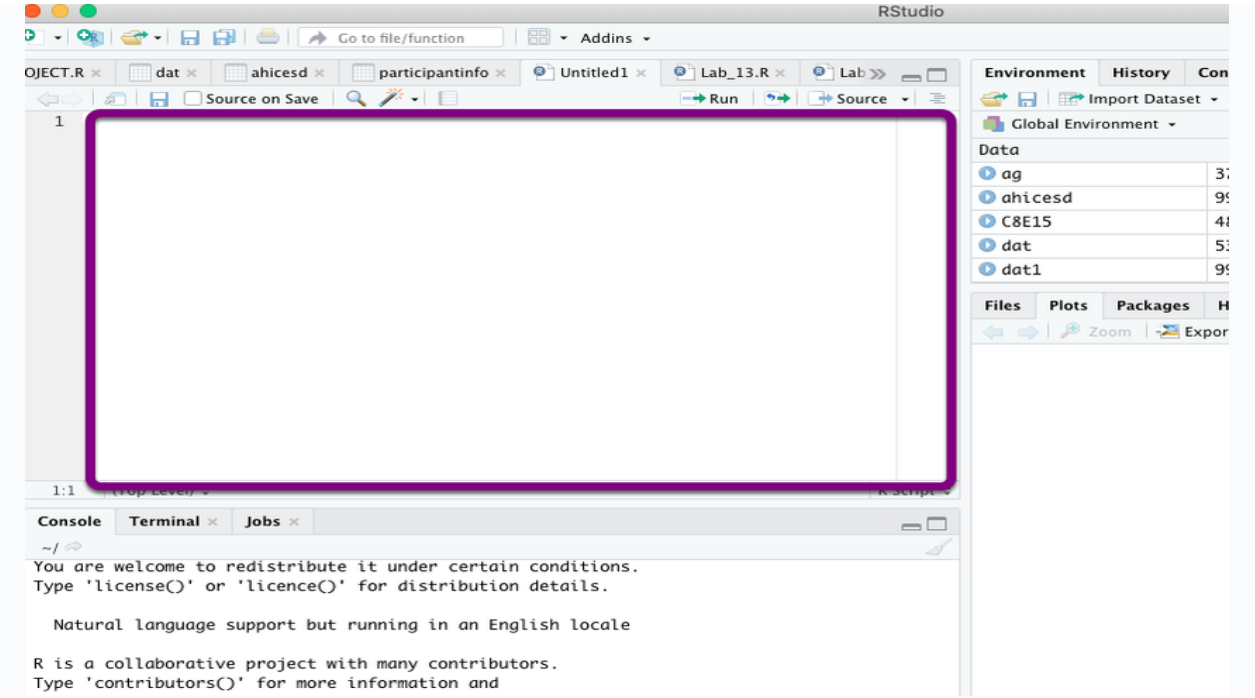
Identify the Panes in RStudio

The Four Main Panes in RStudio Interface

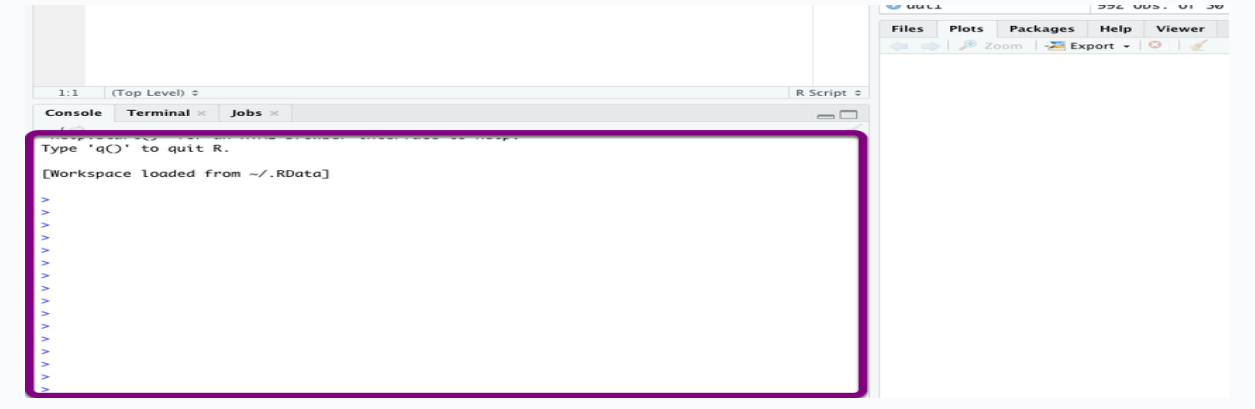
- **Source Editor (Top-Left):**
  Used to write, edit, and save R scripts and functions. Allows execution of selected lines or the full script.
- **Console (Bottom-Left):**
  Used to execute R commands interactively. Immediate output is shown here.
- **Environment / History (Top-Right):**
  Shows active objects (variables, datasets) and command history.
- **Files / Plots / Packages / Help / Viewer (Bottom-Right):**

- **Files:** Navigate files in the working directory.
- **Plots:** View generated graphs and plots.
- **Packages:** Install, load, or update R packages.
- **Help:** Access documentation for functions and packages.
- **Viewer:** Displays web content and Shiny apps.

1. The source pane allows users to view and edit various code-related files, such as .R, .rmd, .qmd, .py, .css, or general text files such as .txt or .md. By default it is the top-left panel and can be launched by opening any editable file in RStudio.

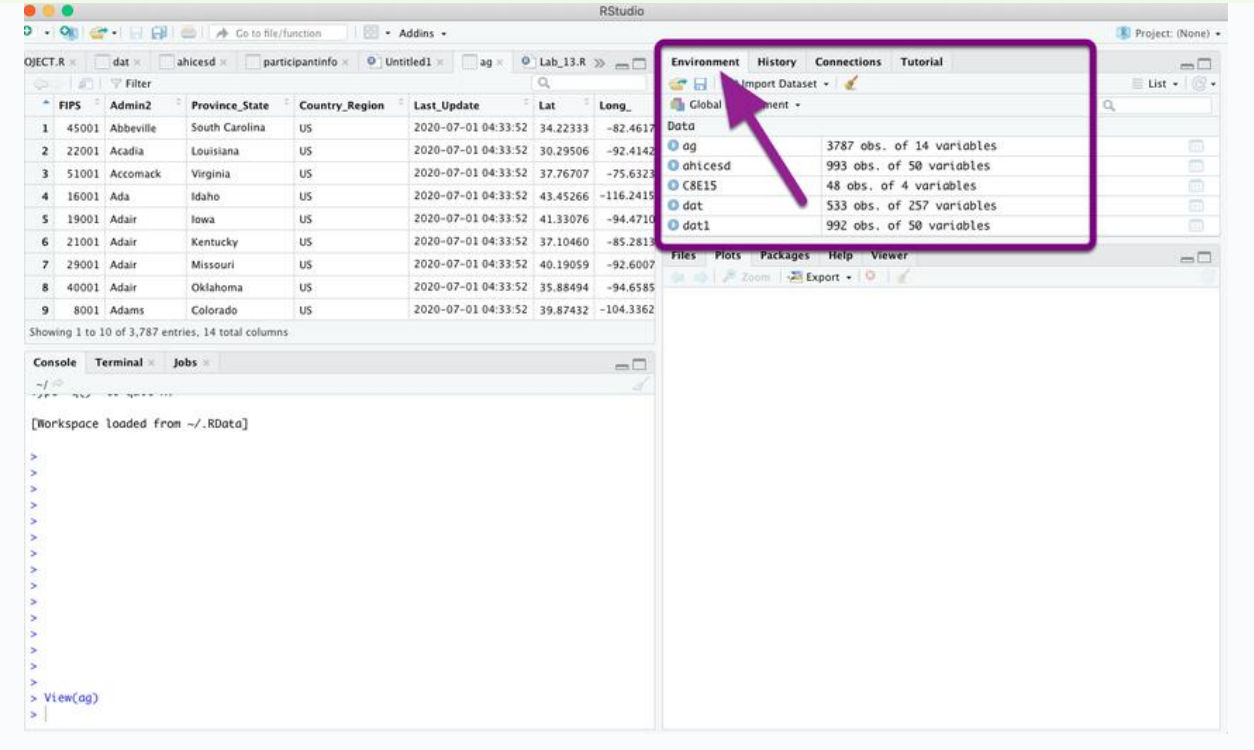


2. By default the console pane is the bottom left pane. The console pane provides an area to interactively execute code.



3. By default, the Environment pane is located in the top-right and includes the Environment, History, Connections, Build, and Version Control System (VCS) tabs. The environment tab displays currently saved R and Python objects.

When you have data in your environment that have two dimensions (rows and columns) you may click on them and they will appear in the Source Editor pane like a spreadsheet. It is at the top right of your screen.



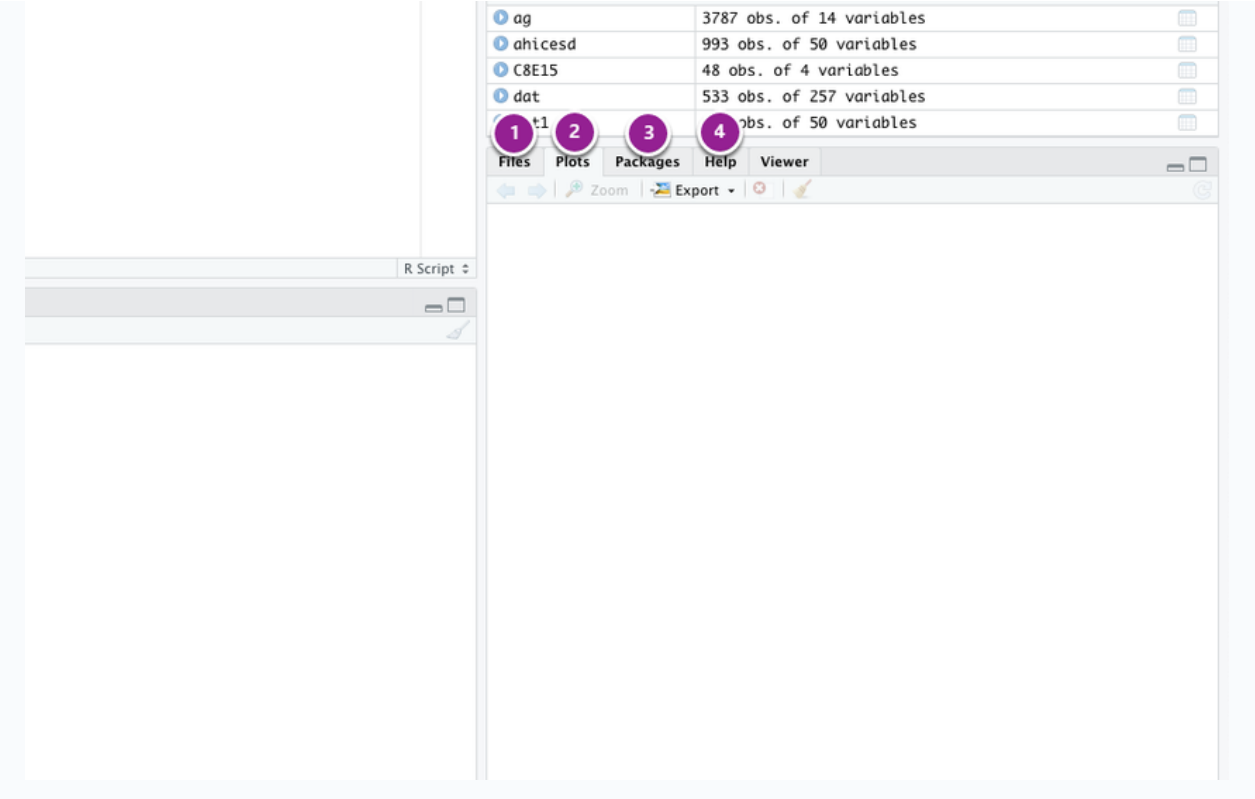
4. The last pane has a number of different tabs.

4.1. The Files tab has a navigable file manager, just like the file system on your operating system.

4.2. The Packages tab shows you the packages that are installed and those that can be installed.

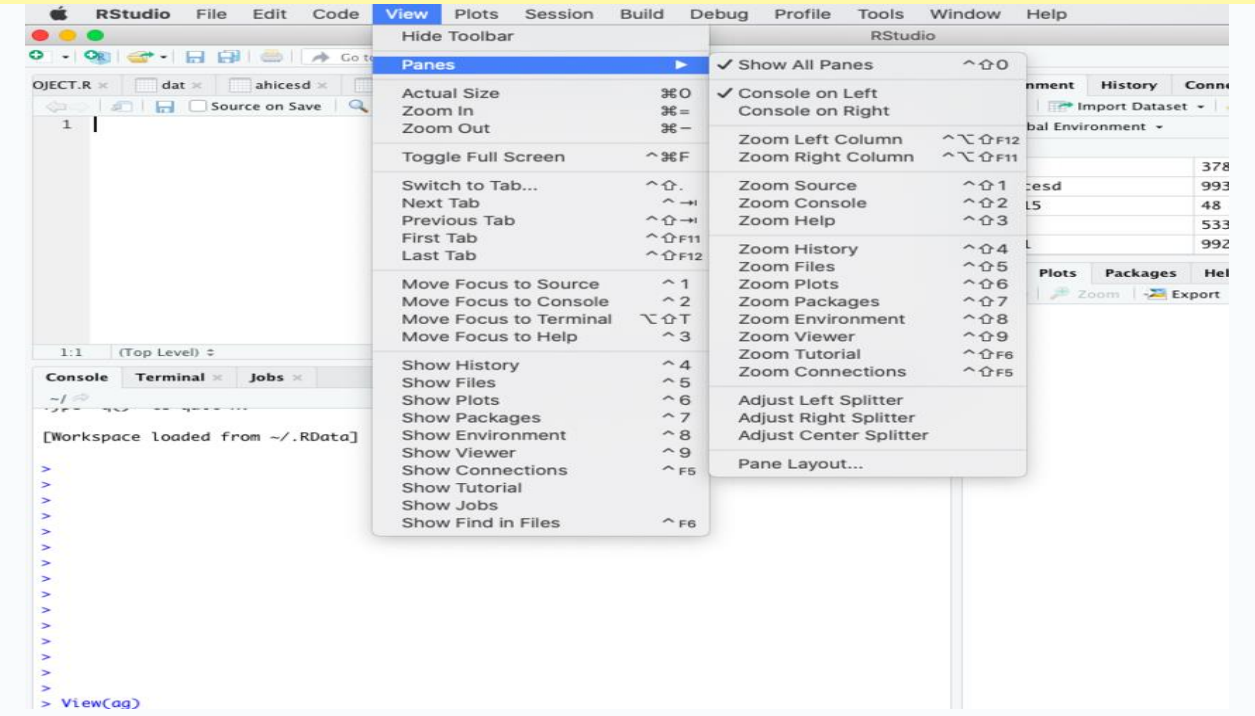
4.3. The Plot tab is where graphics you create will appear.

4.4. The Help tab allows you to search the R documentation for help and is where the help appears when you ask for it from the Console. It is at the bottom right of your screen.



5. Click View and select Panes to adjust the layout of these planes.

There might be subtle differences between RStudio installations on different operating systems.



1.1.3 – SAS VERSUS R

Aspect	SAS	R
Cost	Commercial software, requires expensive licenses	Open-source, free to use and distribute
Developer	SAS Institute Inc.	Developed by the R Core Team, based on the S language
User Interface	Menu-driven interface; GUI and syntax-based programming	Primarily command-line with RStudio as a user-friendly IDE
Learning Curve	Easier for beginners due to step-wise procedures	Steeper learning curve but more flexible and customizable
Statistical Capabilities	Strong in traditional statistics, clinical trials, and reporting	Extensive statistical libraries, supports modern and complex statistical techniques
Data Handling	Handles very large datasets efficiently in enterprise environments	Memory-based; can be optimized using packages like data.table, dplyr, etc.
Graphics and Visualization	Basic built-in graphics, suitable for reports	Advanced and flexible visualization (e.g., ggplot2, plotly, lattice)
Community Support	Limited open community; relies on official tech support	Vast global community, CRAN, Stack Overflow, GitHub for support and resources
Packages and Extensions	Fewer third-party extensions; mostly inbuilt procedures	Over 18,000 packages on CRAN, constantly growing and community-driven
Speed and Performance	Optimized for performance in commercial use cases	Slower with large data unless optimized; depends on coding and memory handling

Updates and Innovation	Slow due to formal testing; commercial release cycle	Frequently updated with new statistical methods and research tools
Integration	Limited integration with open-source tools	Integrates well with Python, Java, C++, SQL, Hadoop, Spark, and cloud platforms
Reproducibility	Strong in reporting but limited in version control	Excellent reproducibility using scripts, R Markdown, and version control tools
Industry Usage	Dominant in pharma, banking, government analytics	Popular in academia, research, data science, and growing in industry
Documentation	Extensive official documentation, regulated content	Rich community-driven documentation, examples, vignettes, and online tutorials
Machine Learning	Limited machine learning libraries; new features added via SAS Viya	Extensive ML support through packages like caret, mlr3, randomForest, xgboost
Platform Availability	Primarily on Windows and Linux with enterprise setups	Cross-platform (Windows, Linux, macOS), also available on cloud and mobile via RStudio

1.1.4 – R ,S AND S-PLUS

- **R** is the most modern and popular of the three, being open-source and flexible with vast support for statistical analysis, machine learning, and data visualization. Its growing community and active development make it an excellent choice for research and data science applications.
- **S** was the foundational language for R, primarily academic, and not widely used outside of research settings.

- **S-plus**, based on S, was developed as a commercial version with a GUI and additional features. However, it is now largely obsolete, with R dominating the field due to its open-source nature and active community support.

Aspect	R	S	S-plus
Origin	Developed by Ross Ihaka and Robert Gentleman in the 1990s at the University of Auckland.	Developed at AT&T Bell Laboratories in the mid-1970s, the precursor to R.	Commercial version of the S language, developed by Statistical Sciences, later acquired by TIBCO.
License	Open-source and free to use.	S is a proprietary language and was not widely available for free use.	Commercial software; requires a paid license for use.
Cost	Free and open-source.	Not available for free use.	Expensive licensing cost for the commercial version.
Programming Interface	Primarily a command-line interface, with support for RStudio as an IDE.	Command-line interface.	GUI-based interface with added functionality for graphical user interface.
Platform Availability	Cross-platform (Windows, macOS, Linux, cloud platforms).	Limited platform support, typically Unix-based systems.	Windows, Unix, and Linux platforms supported.
Syntax	Syntactically similar to S but with significant	Original syntax was used for data	Extended the S language with

	enhancements and differences.	analysis and visualization.	enhanced features and GUI.
Graphical Capabilities	Advanced and highly flexible graphical tools (e.g., ggplot2, plotly).	Basic graphical capabilities, not as advanced as R.	Enhanced graphics with more control over presentation compared to S.
Statistical Capabilities	Rich set of statistical functions, machine learning, and advanced techniques.	Basic statistical analysis and basic linear models.	Strong statistical support, similar to R, but limited flexibility.
Extensibility	Highly extensible with over 18,000 packages available on CRAN.	Limited extensions and libraries.	Limited extensions, but commercial solutions provide additional functionality.
Community Support	Large and active community worldwide, extensive open-source contributions.	Small community, mostly academic and research-focused.	Limited community, primarily professional users.
Development and Updates	Continuously updated with new features, packages, and tools.	S was maintained mainly in the academic and research domain.	S-plus received updates periodically, but less frequently compared to R.
Reproducibility	Excellent reproducibility using scripts, R Markdown, and version control tools.	Limited reproducibility features.	Moderate reproducibility features.

Machine Learning Support	Strong support for machine learning via packages like caret, randomForest, xgboost.	Minimal machine learning support.	Basic machine learning functions with less flexibility than R.
Integration with Other Languages	Excellent integration with Python, C++, Java, and databases.	Limited integration with other languages.	Limited integration capabilities.
Industry Use	Predominantly used in academia, research, and data science.	Used primarily in academic and research domains.	Used in industries such as finance, insurance, and healthcare.

1.1.5 – OBTAINING AND MANAGING R

1. **Managing R Versions**
   - Sometimes, you may need to work with different versions of R. You can manage R versions using version management tools like **installr** (Windows) or **brew** (macOS) for switching versions.
   - In Windows, the **installr** package allows you to install or update R versions from within R:
   - install.packages("installr")
   - library(installr)
   - updateR()
2. **Updating R**
   - It's essential to keep R up to date to ensure compatibility with new packages and improvements.
   - **On Windows**, you can use the **installr** package to update R.
   - **On macOS**, R updates can be managed through the **brew** command or by downloading the latest version from CRAN.
   - **On Linux**, you can update R using your package manager:

- sudo apt update
- sudo apt upgrade r-base

3. **Package Management in R**
   - R's package manager is integral to extending its functionality. To install packages, use the install.packages() function.
   - Example:
   - install.packages("ggplot2")
   - To update all installed packages:
   - update.packages()
   - You can also install packages from GitHub using **devtools**:
   - install.packages("devtools")
   - devtools::install_github("hadley/ggplot2")
4. **Using R from the Command Line**
   - R can be run directly from the command line by typing R. This will open an R session where you can input commands interactively.
   - Scripts can be executed in batch mode using:
   - Rscript my_script.R
5. **Package Repositories**
   - R's primary package repository is CRAN, where most packages are stored.
   - To search for a package, visit CRAN or use RStudio's built-in tool.
   - You can also use **Bioconductor** for bioinformatics-related packages:
   - install.packages("BiocManager")
   - BiocManager::install("DESeq2")
6. **Setting Up R Libraries**
   - Packages are stored in **libraries** (directories). To check which libraries are available:
   - .libPaths()
   - The default library location depends on your operating system. You can specify a custom library path using the .libPaths() function.
7. **R Documentation**

- R has built-in documentation accessible via the help() function or ? operator. For example:
- ?lm # To get help on the 'lm' function
- help("lm")

8. **Managing R Projects in RStudio**
   - RStudio allows you to create and manage projects, which help organize files and scripts.
   - You can create a new project from the **File > New Project** menu in RStudio, which also allows version control integration (e.g., with Git).

1.1.6 – OBJECTS, TYPES OF OBJECTS, CLASSES, CREATING AND ACCESSING OBJECTS

R is a **high-level programming language** and is object-oriented, where data is represented as **objects**. Here are some of the most common types of objects in R:

1. **Vectors**
   - **Definition**: The most basic data structure in R, a vector is an ordered collection of elements that must all be of the same type (numeric, character, logical, etc.).
   - **Example**:
   - x <- c(1, 2, 3, 4) # Numeric vector
   - y <- c("apple", "banana", "cherry") # Character vector
2. **Matrices**
   - **Definition**: A matrix is a two-dimensional data structure where all elements must be of the same type. It is similar to a vector but arranged in rows and columns.
   - **Example**:
   - m <- matrix(1:6, nrow = 2, ncol = 3) # 2x3 matrix
3. **Arrays**
   - **Definition**: Similar to matrices but with more than two dimensions. An array can have one or more dimensions.

- **Example**:
- a <- array(1:12, dim = c(2, 3, 2)) # 3-dimensional array

4. **Data Frames**
   - **Definition**: A data frame is a two-dimensional table-like structure in which each column can hold elements of different types (numeric, character, logical). It is widely used in R for statistical data manipulation.
   - **Example**:
   - df <- data.frame(name = c("John", "Jane"), age = c(23, 25))
5. **Lists**
   - **Definition**: A list can hold elements of different types, including other lists, making it a flexible data structure.
   - **Example**:
   - lst <- list(name = "John", age = 23, marks = c(90, 85, 88))
6. **Factors**
   - **Definition**: Factors are R's data structure for categorical data. They store both the values of a categorical variable and its levels (unique values).
   - **Example**:
   - factor_var <- factor(c("low", "high", "medium", "high"))
7. **Functions**
   - **Definition**: Functions are objects in R that can be created using the function keyword and can accept arguments, perform computations, and return results.
   - **Example**:

```
add <- function(x, y) {
  return(x + y)
}
```

Classes in R

R is an **object-oriented language**, but its object-oriented system is relatively flexible. R supports multiple object-oriented programming (OOP) systems:

1. **S3 Classes**
   - **Definition**: The most common and informal object-oriented system in R. Objects are simply lists, but they are **tagged** with class names. Methods for S3 classes are based on **generic functions** and are defined by class name matching.
   - **Example**:

```
# Define an S3 class
person <- list(name = "John", age = 30)
class(person) <- "Person"
```

2. **S4 Classes**
   - **Definition**: A more formal and rigorous object-oriented system compared to S3. It supports **method dispatching** based on both the class of the object and its arguments. S4 also allows for strict class definitions.
   - **Example**:

```
setClass("Person",
         slots = c(name = "character", age = "numeric"))
john <- new("Person", name = "John", age = 30)
```

3. **R6 Classes**
   - **Definition**: The R6 class system is based on reference classes and is a more modern and efficient OOP system. It provides full encapsulation and inheritance and is useful for creating objects that are **mutable** (can change over time).
   - **Example**:

```
library(R6)
Person <- R6Class("Person",
                  public = list(
                    name = NULL,
                    age = NULL,
                    initialize = function(name, age) {
                      self$name <- name
                      self$age <- age
                    },
```

```
                    greet = function() {
                      cat("Hello, my name is", self$name, "and I am", self$age,
"years old.\n")
                    }
                  ))
john <- Person$new(name = "John", age = 30)
john$greet()
```

4. **Reference Classes (RC)**
   - **Definition**: Reference classes provide mutable objects in R and are part of the S4 system. Reference classes are suitable for situations where objects need to have state that changes over time.
   - **Example**:

```
setRefClass("Person",
            fields = list(name = "character", age = "numeric"))
john <- Person$new(name = "John", age = 30)
```

Creating and Accessing Objects

1. **Creating Objects**
   - Objects can be created using the **assignment operator** (<-), and the structure of the object depends on the data type (vector, matrix, list, etc.).
     - **Example**:

```
x <- 5  # Creating a numeric object
y <- c(1, 2, 3, 4)  # Creating a vector
df <- data.frame(a = 1:5, b = letters[1:5])  # Creating a data frame
```

2. **Accessing Elements of Objects**
   - Accessing elements in objects depends on the object type. Common methods are:
     - **Vectors**: Use indices to access elements.
     - y[2] # Access the second element of the vector
     - **Data Frames**: Access by column name or index.

```
df$a  # Access column 'a' in a data frame
df[2, 1]  # Access element at second row and first column
```

- **Lists**: Use the $ operator or double square brackets [[ ]] to access elements.

  ```
  lst$name  # Access the 'name' element in a list
  lst[[1]]  # Access the first element in the list
  ```

3. **Modifying Objects**
   - You can modify existing objects by **assigning new values** to specific parts of the object.
     - **Example**:

       ```
       y[2] <- 10  # Modify the second element of vector y
       df$a <- df$a + 1  # Modify values in column 'a' of data frame
       ```

4. **Classes and Methods**
   - For **S3** classes, you can define methods using the UseMethod function, and for **S4** classes, methods are defined based on the class of the object.
     - **S3 Example**:

       ```
       print.Person <- function(x) {
         cat("This is a person named", x$name, "aged", x$age, "\n")
       }
       print(john)  # This will use the print.Person method
       ```

Category	Description	Examples
Types of Objects		
Vectors	Ordered collection of elements of the same type (numeric, character, logical, etc.).	x <- c(1, 2, 3, 4) (Numeric vector), y <- c("apple", "banana") (Character vector)
Matrices	Two-dimensional collection of elements of the same type, arranged in rows and columns.	m <- matrix(1:6, nrow = 2, ncol = 3) (2x3 matrix)
Arrays	Similar to matrices but can have more than two dimensions.	a <- array(1:12, dim = c(2, 3, 2)) (3D array)
Data Frames	Two-dimensional table-like structure where each column can	df <- data.frame(name = c("John", "Jane"), age = c(23,

	hold different types of elements (numeric, character, etc.).	25)) (Data frame with mixed data types)
Lists	Collection that can hold elements of different types, including other lists.	lst <- list(name = "John", age = 23, marks = c(90, 85, 88)) (List with mixed data types)
Factors	Used for categorical data, storing both the values and their possible levels.	factor_var <- factor(c("low", "high", "medium", "high")) (Factor with categories)
Functions	Blocks of reusable code that accept inputs, perform tasks, and return results.	add <- function(x, y) { return(x + y) } (Function to add two numbers)
Classes in R		
S3 Classes	Informal object-oriented system in R. Objects are usually lists tagged with a class name. Methods are based on generic functions.	class(person) <- "Person" (S3 class for person object)
S4 Classes	More formal object-oriented system, supports method dispatch based on object class and its arguments.	setClass("Person", slots = c(name = "character", age = "numeric")) (S4 class definition)
R6 Classes	Modern, reference-based object system supporting encapsulation and inheritance.	Person <- R6Class("Person", public = list(name = NULL, age = NULL)) (R6 class)
Reference Classes	Provides mutable objects, similar to S4, but with reference semantics (can be modified in place).	setRefClass("Person", fields = list(name = "character", age = "numeric")) (Reference class for mutable data)
Creating Objects		
Vectors	Use c() function to create a vector.	v <- c(1, 2, 3)

Matrices	Use matrix() function to create a matrix with specified rows and columns.	m <- matrix(1:6, nrow = 2, ncol = 3)
Data Frames	Use data.frame() function to create data frames with mixed column types.	df <- data.frame(a = 1:5, b = letters[1:5])
Lists	Use list() function to create a list.	lst <- list(name = "John", age = 23)
Accessing Objects		
Vectors	Access elements using index [].	v[1] (Access first element)
Matrices	Access elements by row and column indices [row, column].	m[1,2] (Access element in first row, second column)
Data Frames	Access columns by name or index, rows by position or conditions.	df$a (Access column 'a'), df[2,1] (Access element in second row, first column)
Lists	Use $ operator or [[ ]] to access elements in a list.	lst$name (Access element 'name'), lst[[1]] (Access first element)
Modifying Objects		
Vectors	Modify elements by assigning a new value using [].	v[2] <- 10 (Change the second element of vector v)
Matrices	Modify specific cells using row and column indices.	m[1,2] <- 10 (Change the element in first row, second column of matrix m)
Data Frames	Modify values in specific columns or rows.	df$a <- df$a + 1 (Add 1 to all values in column 'a' of data frame df)

EXAMPLE PROGRAM

```
# -------------------------
# Types of Objects in R
```

```
# -------------------------
# 1. Vectors
vec <- c(10, 20, 30)
cat("Vector:\n"); print(vec)
# 2. Matrices
mat <- matrix(1:6, nrow = 2)
cat("\nMatrix:\n"); print(mat)
# 3. Arrays
arr <- array(1:8, dim = c(2, 2, 2))
cat("\nArray:\n"); print(arr)
# 4. Data Frames
df <- data.frame(ID = 1:3, Name = c("Tom", "Jerry", "Spike"), Marks = c(90, 85, 88))
cat("\nData Frame:\n"); print(df)
# 5. Lists
my_list <- list(Name = "Alice", Age = 25, Scores = c(85, 90, 95))
cat("\nList:\n"); print(my_list)
# 6. Factors
fact <- factor(c("High", "Medium", "Low", "Medium", "High"))
cat("\nFactor:\n"); print(fact)
# 7. Functions
sum_func <- function(a, b) {
  return(a + b)
}
cat("\nFunction Output (10 + 5): ", sum_func(10, 5), "\n")
# -------------------------
# Classes in R
# -------------------------
# 1. S3 Class
s3_obj <- list(name = "John", age = 30)
class(s3_obj) <- "person"
print(s3_obj)
# 2. S4 Class
setClass("Employee", slots = list(name = "character", salary = "numeric"))
emp <- new("Employee", name = "Sara", salary = 50000)
cat("\nS4 Class:\n"); print(emp)
# 3. R6 Class
if (!require(R6)) install.packages("R6")
library(R6)
Person <- R6Class("Person",
```

```
  public = list(
    name = NULL,
    initialize = function(name) {
      self$name <- name
    },
    greet = function() {
      cat("Hello, my name is", self$name, "\n")
    }
  )
)
p <- Person$new("Charlie")
cat("\nR6 Class:\n")
p$greet()
# 4. Reference Class
RefPerson <- setRefClass("RefPerson", fields = list(name = "character"))
ref_obj <- RefPerson$new(name = "David")
cat("\nReference Class:\n"); print(ref_obj)
# -------------------------
# Creating Objects
# -------------------------
# Already shown: vec, mat, df, my_list
# -------------------------
# Accessing Objects
# -------------------------
cat("\nAccess Elements:\n")
cat("2nd element of vector:", vec[2], "\n")
cat("Element [1,2] in matrix:", mat[1,2], "\n")
cat("2nd row Name from df:", df$Name[2], "\n")
cat("Age from list:", my_list$Age, "\n")
# -------------------------
# Modifying Objects
# -------------------------
# Vectors
vec[3] <- 35
cat("\nModified Vector:\n"); print(vec)
# Matrices
mat[2,1] <- 100
cat("\nModified Matrix:\n"); print(mat)
# Data Frames
df$Marks[1] <- 95
```

```
cat("\nModified Data Frame:\n"); print(df)
```

OUTPUT

```
Vector:
[1] 10 20 30
Matrix:
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
Array:
, , 1
     [,1] [,2]
[1,]    1    3
[2,]    2    4
, , 2
     [,1] [,2]
[1,]    5    7
[2,]    6    8
Data Frame:
  ID  Name Marks
1  1   Tom    90
2  2 Jerry    85
3  3 Spike    88
List:
$Name
[1] "Alice"
$Age
[1] 25
$Scores
[1] 85 90 95
Factor:
[1] High   Medium Low    Medium High
Levels: High Low Medium
Function Output (10 + 5):  15
$name
[1] "John"
$age
[1] 30
attr(,"class")
[1] "person"
S4 Class:
```

```
An object of class "Employee"
Slot "name":
[1] "Sara"
Slot "salary":
[1] 50000
R6 Class:
Hello, my name is Charlie
Reference Class:
Reference class object of class "RefPerson"
Field "name":
[1] "David"
Access Elements:
2nd element of vector: 20
Element [1,2] in matrix: 3
2nd row Name from df: Jerry
Age from list: 25
Modified Vector:
[1] 10 20 35
Modified Matrix:
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]  100    4    6
Modified Data Frame:
  ID  Name Marks
1  1   Tom    95
2  2 Jerry    85
3  3 Spike    88
```

R provides flexible and powerful methods to create and manage objects. The core types of objects include **vectors**, **matrices**, **data frames**, **lists**, and **functions**, while its object-oriented systems (S3, S4, R6) allow for a range of capabilities from informal class definitions to fully structured and immutable classes. R's object manipulation is easy to use with simple commands for creating, accessing, and modifying objects, which makes it an essential tool for data manipulation, statistical analysis, and advanced object-oriented programming.

1.1.7 – AIRTHMETIC AND MATRIX OPERATIONS IN R

R is a powerful programming language that allows users to perform various arithmetic and matrix operations easily. Below is an overview of common operations:

1. Arithmetic Operations

Arithmetic operations in R are similar to those in most programming languages and can be performed on vectors, matrices, and scalar values.

Operation	Symbol	Description	Example
Addition	+	Adds two values or elements	3 + 5 -> 8
Subtraction	-	Subtracts one value from another	10 - 4 -> 6
Multiplication	*	Multiplies two values or elements	2 * 4 -> 8
Division	/	Divides one value by another	8 / 2 -> 4
Exponentiation	^	Raises a value to a power	2^3 -> 8
Modulo	%%	Returns the remainder after division	10 %% 3 -> 1
Integer Division	%/%	Returns the quotient after division	10 %/% 3 -> 3

2. Matrix Operations

Matrices are two-dimensional arrays in R, and matrix operations are often crucial in data analysis, particularly in linear algebra and machine learning. Common matrix operations include addition, multiplication, inversion, and transposition.

Operation	Symbol	Description	Example
Matrix Addition	+	Adds two matrices element-wise. Both matrices must have the same dimensions.	A + B where A and B are matrices of the same dimensions.
Matrix Subtraction	-	Subtracts one matrix from another element-wise. Must have the same dimensions.	A - B where A and B are matrices of the same dimensions.

Matrix Multiplication	%*%	Multiplies two matrices using matrix multiplication rules (dot product).	A %*% B where A and B are matrices with compatible dimensions.
Element-wise Multiplication	*	Multiplies two matrices element by element. Both matrices must have the same size.	A * B (Note: Not true matrix multiplication)
Matrix Transposition	t()	Flips the rows and columns of a matrix (row to column and vice versa).	t(A) where A is a matrix.
Matrix Inversion	solve()	Calculates the inverse of a matrix. Only square, non-singular matrices can be inverted.	solve(A) where A is a square matrix.
Determinant	det()	Calculates the determinant of a matrix. Only square matrices can have a determinant.	det(A) where A is a square matrix.
Eigenvalues and Eigenvectors	eigen()	Computes the eigenvalues and eigenvectors of a matrix.	eigen(A) where A is a square matrix.

Examples:

Example 1: Arithmetic Operations

```
# Simple arithmetic operations
addition <- 5 + 3      # 8
subtraction <- 10 - 4  # 6
multiplication <- 2 * 3 # 6
division <- 9 / 3      # 3
exponentiation <- 2^3  # 8
```

Example 2: Matrix Operations

```
# Create two matrices A and B
A <- matrix(1:9, nrow = 3, ncol = 3)
B <- matrix(9:1, nrow = 3, ncol = 3)
# Matrix addition
matrix_addition <- A + B
# Matrix multiplication (dot product)
matrix_multiplication <- A %*% B
# Element-wise multiplication
elementwise_multiplication <- A * B
# Matrix transposition
matrix_transpose <- t(A)
# Matrix inversion (if A is invertible)
matrix_inverse <- solve(A)
```

Example 3: Eigenvalues and Eigenvectors

```
# Eigenvalues and eigenvectors of matrix A
eigen_result <- eigen(A)
eigenvalues <- eigen_result$values
eigenvectors <- eigen_result$vectors
```

4. Matrix Operations Notes:

- **Matrix Multiplication (%*%)** follows the standard rules of linear algebra, where the number of columns in the first matrix must equal the number of rows in the second matrix.
- **Element-wise multiplication (*)** performs a simple multiplication for corresponding elements, without the usual linear algebraic rules.
- **Matrix Inversion (solve())** can only be done if the matrix is square and non-singular (i.e., the determinant is not zero).
- **Transpose (t())** simply swaps rows and columns.
- **Eigenvalues and Eigenvectors (eigen())** provide important insights for linear algebraic transformations, and are used in various applications like principal component analysis (PCA).

5. Additional Functions for Matrix Operations:

- crossprod(x, y) - Computes the cross product of two vectors or matrices.
- tcrossprod(x, y) - Computes the transpose of the cross product.
- qr() - Computes the QR decomposition of a matrix.
- svd() - Performs Singular Value Decomposition (SVD) of a matrix.

PROGRAM

```
# Arithmetic Operations
a <- 10
b <- 5
addition <- a + b
subtraction <- a - b
multiplication <- a * b
division <- a / b
exponentiation <- a^b
modulo <- a %% b
integer_division <- a %/% b
cat("Arithmetic Operations:\n")
cat("Addition: ", addition, "\n")
cat("Subtraction: ", subtraction, "\n")
cat("Multiplication: ", multiplication, "\n")
cat("Division: ", division, "\n")
cat("Exponentiation: ", exponentiation, "\n")
cat("Modulo: ", modulo, "\n")
cat("Integer Division: ", integer_division, "\n\n")
# Matrix Operations
# Create two matrices A and B
A <- matrix(1:9, nrow = 3, ncol = 3)
B <- matrix(9:1, nrow = 3, ncol = 3)
cat("Matrix A:\n")
print(A)
cat("\nMatrix B:\n")
print(B)
# Matrix Addition
matrix_addition <- A + B
# Matrix Multiplication
matrix_multiplication <- A %*% B
```

```
# Element-wise Multiplication
elementwise_multiplication <- A * B
# Matrix Transposition
matrix_transpose <- t(A)
# Matrix Inversion
matrix_inverse <- tryCatch(solve(A), error = function(e) "Matrix is singular and cannot be inverted")
cat("\nMatrix Operations:\n")
cat("Matrix Addition:\n")
print(matrix_addition)
cat("Matrix Multiplication (A %*% B):\n")
print(matrix_multiplication)
cat("Element-wise Multiplication (A * B):\n")
print(elementwise_multiplication)
cat("Matrix Transpose (t(A)):\n")
print(matrix_transpose)
cat("Matrix Inversion (solve(A)):\n")
print(matrix_inverse)
```

OUTPUT

```
Arithmetic Operations:
Addition:  15
Subtraction:  5
Multiplication:  50
Division:  2
Exponentiation:  100000
Modulo:  0
Integer Division:  2
Matrix A:
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
Matrix B:
     [,1] [,2] [,3]
[1,]    9    6    3
[2,]    8    5    2
[3,]    7    4    1
Matrix Operations:
```

```
Matrix Addition:
     [,1] [,2] [,3]
[1,]   10   10   10
[2,]   10   10   10
[3,]   10   10   10
Matrix Multiplication (A %*% B):
     [,1] [,2] [,3]
[1,]   50   32   14
[2,]   56   38   20
[3,]   62   44   26
Element-wise Multiplication (A * B):
     [,1] [,2] [,3]
[1,]    9   24   21
[2,]   16   25   16
[3,]   21   24    9
Matrix Transpose (t(A)):
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
Matrix Inversion (solve(A)):
Matrix is singular and cannot be inverted
```

R provides extensive capabilities for performing arithmetic and matrix operations, making it a versatile tool for data analysis, linear algebra, and scientific computing. With its matrix-oriented functions, R simplifies the implementation of mathematical and statistical algorithms, especially in fields like machine learning, data science, and engineering.

1.1.8 INTRODUCTION TO FUNCTIONS IN R

Functions in R are one of the most powerful aspects of the language, allowing users to group reusable code and perform repetitive tasks more efficiently. Functions can take inputs (called *arguments*), perform some operations on them, and return a result. R has a large set of built-in functions, but you can also define your own custom functions.

1. Syntax of Functions in R

To define a function in R, you use the function() keyword. The basic structure is:

```
function_name <- function(arg1, arg2, ...) {
  # Function body
  # Operations on arguments
  return(result)  # Optional
}
```

2. Example: Built-in Functions in R

R has numerous built-in functions, for example:

- sum(): Adds all the numbers in a vector.
- mean(): Calculates the mean of a numeric vector.
- sd(): Computes the standard deviation of a numeric vector.

Example

```
# Using built-in functions
numbers <- c(1, 2, 3, 4, 5)
total <- sum(numbers)  # Adds all numbers in the vector
average <- mean(numbers)  # Calculates the mean
deviation <- sd(numbers)  # Calculates the standard deviation
cat("Sum: ", total, "\n")
cat("Mean: ", average, "\n")
cat("Standard Deviation: ", deviation, "\n")
```

3. Creating Your Own Function in R

Functions allow you to encapsulate code into a block that can be reused. Here's how you can create a simple function that adds two numbers:

```
# Defining a function to add two numbers
add_numbers <- function(x, y) {
  result <- x + y
  return(result)
}
# Calling the function
```

```
sum_result <- add_numbers(10, 5)
cat("The sum is: ", sum_result, "\n")
```

4. Example Program: Custom Function in R

Let's write a program with multiple custom functions to demonstrate how functions can interact with each other.

```
# Function to add two numbers
add <- function(x, y) {
  return(x + y)
}
# Function to subtract two numbers
subtract <- function(x, y) {
  return(x - y)
}
# Function to multiply two numbers
multiply <- function(x, y) {
  return(x * y)
}
# Function to divide two numbers
divide <- function(x, y) {
  if (y == 0) {
    return("Error: Division by zero")
  }
  return(x / y)
}
# Calling the functions and printing the results
num1 <- 10
num2 <- 5
cat("Addition: ", add(num1, num2), "\n")
cat("Subtraction: ", subtract(num1, num2), "\n")
cat("Multiplication: ", multiply(num1, num2), "\n")
cat("Division: ", divide(num1, num2), "\n")
```

Output of the Program

```
Addition:  15
Subtraction:  5
Multiplication:  50
Division:  2
```

Explanation of the Program:

1. **Custom Functions**: The program defines four functions: add, subtract, multiply, and divide. Each function performs a specific arithmetic operation.
2. **Function Call**: The functions are called with the arguments num1 = 10 and num2 = 5.
3. **Return Value**: Each function returns a result, which is then printed using the cat() function.

6. Arguments in Functions

- **Default Arguments**: You can provide default values for function arguments.

  Example:

```
multiply <- function(x = 2, y = 3) {
  return(x * y)
}
cat("Default multiplication: ", multiply(), "\n")
```

  Output:

  Default multiplication: 6

- **Passing Arguments by Name**: You can also call a function by explicitly naming the arguments.

Example:

```
multiply(x = 5, y = 10)
```

- **Variable Number of Arguments**: Functions in R can accept a variable number of arguments using the ... parameter. This allows you to pass any number of arguments to a function.

Example:

```
sum_all <- function(...) {
  total <- sum(...)
  return(total)
}
cat("Sum of numbers: ", sum_all(1, 2, 3, 4, 5), "\n")
```

Output:

Sum of numbers: 15

7. Scope of Variables in Functions

- **Local Scope**: Variables defined within a function are local to that function.
- **Global Scope**: Variables defined outside the function are accessible globally, but must be explicitly referred to inside the function if used.

Example:

```
x <- 10  # Global variable
my_function <- function() {
  x <- 5  # Local variable
  cat("Inside function, x =", x, "\n")
}
my_function()  # Local variable x is used inside the function
cat("Outside function, x =", x, "\n")  # Global variable x is used outside
```

EXAMPLE

```
# 1. Global variable
global_var <- 100
# 2. Basic function with return value
add <- function(a, b) {
  return(a + b)
}
# 3. Function with default arguments
multiply <- function(a = 2, b = 3) {
  return(a * b)
}
# 4. Function using named arguments
power <- function(base, exponent) {
```

```
  return(base ^ exponent)
}
# 5. Function with variable number of arguments using ...
sum_all <- function(...) {
  total <- sum(...)
  return(total)
}
# 6. Function demonstrating local vs global scope
scope_demo <- function() {
  local_var <- 50
  cat("Inside function - Local variable:", local_var, "\n")
  cat("Inside function - Global variable accessed:", global_var, "\n")
}
# Calling all functions
a <- 10
b <- 5
cat("Addition of", a, "and", b, ":", add(a, b), "\n")
cat("Multiplication (default args):", multiply(), "\n")
cat("Multiplication (custom args):", multiply(4, 5), "\n")
cat("Power (named args):", power(exponent = 3, base = 2), "\n")
cat("Sum of many numbers:", sum_all(1, 2, 3, 4, 5), "\n")
scope_demo()
# Accessing global variable outside
cat("Outside function - Global variable:", global_var, "\n")
```

OUTPUT

```
Addition of 10 and 5 : 15
Multiplication (default args): 6
Multiplication (custom args): 20
Power (named args): 8
Sum of many numbers: 15
Inside function - Local variable: 50
Inside function - Global variable accessed: 100
Outside function - Global variable: 100
```

Functions in R are essential for organizing code into reusable components. By using functions, you can write cleaner, more modular code, reducing redundancy and increasing efficiency. Functions are widely used in statistical analysis, data processing, and machine learning pipelines in R.

1.2 Let Us Sum Up

This unit introduced the fundamentals of R programming, emphasizing its development, installation, and application in data analytics. We discussed the setup of R and RStudio, distinctions between SAS, R, and S/S-plus, and how to install and manage R packages. We covered R's object-oriented structure, including types, classes, and object manipulation. Fundamental operations like arithmetic and matrix handling were explored, culminating in an introduction to user-defined and built-in functions in R.

1.3 Check Your Progress

1. Which of the following languages influenced the development of R?
   A) Python
   B) S
   C) Java
   D) Ruby
2. What is the main purpose of R programming language?
   A) Web Development
   B) Mobile App Development
   C) Statistical Computing and Graphics
   D) Operating System Design
3. **R was primarily developed by:**
   A) Dennis Ritchie
   B) Guido van Rossum
   C) Ross Ihaka and Robert Gentleman
   D) John McCarthy
4. **Which IDE is most commonly used for R programming?**
   A) Spyder
   B) PyCharm
   C) RStudio
   D) Eclipse

5. **What is the correct way to assign a value to a variable in R?**

   A) x := 10

   B) x = 10

   C) x <- 10

   D) 10 -> x

6. **Which function is used to install packages in R?**

   A) install.package()

   B) install()

   C) install.packages()

   D) load.package()

7. **What is the file extension of an R script?**

   A) .txt

   B) .r

   C) .py

   D) .rs

8. **Which function is used to create a vector in R?**

   A) make.vector()

   B) create.vector()

   C) c()

   D) vector()

9. **In R, which operator is used for assignment?**

   A) :=

   B) ->

   C) <-

   D) =

10. **Which of the following is NOT a data structure in R?**

    A) Data frame

    B) Matrix

    C) Tuple

    D) List

11. **How do you access the elements of a vector in R?**
    A) vector[]
    B) vector()
    C) vector{}
    D) vector{}[]
12. **What is the default method for generating random numbers in R?**
    A) runif()
    B) rnorm()
    C) rpois()
    D) sample()
13. **Which of the following is the correct way to create a matrix in R?**
    A) matrix()
    B) create_matrix()
    C) new_matrix()
    D) mat()
14. **Which command is used to check the structure of an object in R?**
    A) view()
    B) structure()
    C) str()
    D) summary()
15. **Which of the following types of objects are used to store tabular data in R?**
    A) List
    B) Data frame
    C) Matrix
    D) Array
16. **What type of object is returned by the function lm() in R?**
    A) List
    B) Data frame
    C) Linear model object
    D) Matrix

17. **How do you concatenate two vectors in R?**
    A) concatenate()
    B) join()
    C) merge()
    D) c()
18. **In R, which operator is used for element-wise multiplication of two vectors?**
    A) *
    B) %*%
    C) .
    D) &
19. **Which of the following is NOT a valid data type in R?**
    A) Character
    B) Integer
    C) Boolean
    D) Object
20. **Which function is used to check the data type of an object in R?**
    A) typeof()
    B) is()
    C) class()
    D) check()
21. **In R, which function is used to create a new factor?**
    A) create_factor()
    B) factor()
    C) levels()
    D) labels()
22. **Which data structure in R is used to store elements of different types?**
    A) Matrix
    B) Data frame
    C) Vector
    D) Array

23. **Which of the following is the correct way to access the first element of a data frame in R?**
    A) df[1]
    B) df(1)
    C) df[1,]
    D) df(1,)
24. **Which of the following functions is used to plot a histogram in R?**
    A) hist()
    B) barplot()
    C) plot()
    D) scatter()
25. **In R, what is the purpose of the function apply()?**
    A) To apply a function over a matrix or array
    B) To apply a function over a list
    C) To apply a function over a vector
    D) To apply a function over a data frame

1.4 Unit Summary

This unit introduces R programming, focusing on setting up the R environment with RStudio and understanding the basics of R's functionalities. It covers the installation of R and essential packages, along with the core data structures like vectors, matrices, lists, and data frames, which are vital for data manipulation. The unit also explains object types and classes in R, basic arithmetic and matrix operations, and the use of functions for modular coding.

1.5 Glossary

1. **R Programming**: A programming language and environment used for statistical computing, data analysis, and graphical visualization.
2. **RStudio**: An integrated development environment (IDE) for R, providing a user-friendly interface to write and execute R code.

3. **Vector**: A basic data structure in R used to store a sequence of elements of the same type (numeric, character, etc.).
4. **Matrix**: A two-dimensional array in R, which is used to store elements in rows and columns.
5. **List**: A flexible data structure in R that can hold elements of different types, such as vectors, matrices, or even other lists.
6. **Data Frame**: A two-dimensional data structure in R used to store data in a table format, where each column can be of a different data type.
7. **Object**: A variable or data structure that holds data and can be manipulated in R.
8. **Function**: A reusable block of code that performs a specific task. Functions in R can take arguments and return values.
9. **Arithmetic Operations**: Mathematical operations like addition, subtraction, multiplication, and division performed on R objects.
10. **Matrix Operations**: Operations such as addition, multiplication, and transposition performed on matrices in R.
11. **Package**: A collection of functions, data, and documentation in R, which extends its functionality for specific tasks like data visualization, statistical analysis, etc.
12. **Scripting**: Writing a series of commands in a script file to automate tasks in R.
13. **S-Plus**: A commercial software package that is based on the S programming language, from which R is derived.
14. **Packages**: Pre-built collections of functions and datasets in R that extend its base functionality. Examples include ggplot2 for plotting or dplyr for data manipulation.
15. **R Console**: An interactive interface where you can run R commands and view results in real-time.

1.6 Self-Assessment Questions

1. What are the differences between R and S-Plus?
2. Explain the process of setting up RStudio for the first time.
3. What are the key features of the R environment?
4. Define the term "object" in R and provide an example.

5. What is the difference between a matrix and a data frame in R?
6. Describe the process of creating and accessing objects in R.
7. How would you perform matrix multiplication in R?
8. What are the primary arithmetic operations available in R? Give an example of each.
9. What are the benefits of using R functions?
10. What is a list in R, and how is it different from a vector?
11. How do you install and use a package in R?
12. Explain the concept of "data frames" in R and how they are useful for data analysis.
13. What are the different types of objects in R?
14. How do you create a matrix in R? Provide an example.
15. Describe the types of operations you can perform on vectors in R.
16. What is the significance of the R console, and how is it used during coding?
17. How would you convert a vector into a matrix in R?

1.7 Activities / Exercises / Case Studies:

1. **Activity 1: Setting Up RStudio**
    - Download and install RStudio and R. Set up a basic project in RStudio and explore the interface. Write a simple script to calculate the sum of numbers from 1 to 100.
2. **Activity 2: Basic Arithmetic Operations in R**
    - Create a vector of numbers and perform basic arithmetic operations (addition, subtraction, multiplication, and division) on the vector. Write a function to find the square and cube of each number in the vector.
3. **Activity 3: Matrix Operations**
    - Create two matrices in R and perform matrix addition, multiplication, and transposition on them. Write a script to check the properties of a matrix, such as its determinant and rank.
4. **Activity 4: Data Frames and Data Analysis**

- Import a CSV file into R as a data frame. Clean the data by handling missing values and performing basic summary statistics like mean, median, and standard deviation on numerical columns.

5. **Case Study 1: R vs SAS in Data Analysis**
   - Research and compare the use of R and SAS for data analysis. Prepare a report that includes the advantages and disadvantages of each software. Discuss scenarios where one might be preferred over the other.
6. **Activity 5: Custom Function Creation**
   - Write a custom function in R that takes a numeric vector as input and returns the mean and standard deviation of the vector. Then, apply this function to different datasets.
7. **Exercise 1: Matrix Operations and Functions**
   - Create a matrix in R and apply different matrix operations. Write a function that accepts two matrices and returns the product of the two matrices.
8. **Case Study 3: Debugging in R**
   - Write a script with intentional errors (e.g., syntax errors or runtime errors) and practice debugging using RStudio's debugging tools like breakpoints and traceback(). Resolve the errors and ensure the script runs as intended.

1.8 ANSWERS FOR CHECK YOUR PROGRESS

1. **B)** S
2. **C)** Statistical Computing and Graphics
3. **C)** Ross Ihaka and Robert Gentleman
4. **C)** RStudio
5. **C)** x <- 10
6. **C)** install.packages()
7. **B)** .r
8. **C)** c()
9. **C)** <-
10. **C)** Tuple

11. **A)** vector[]
12. **B)** rnorm()
13. **A)** matrix()
14. **C)** str()
15. **B)** Data frame
16. **C)** Linear model object
17. **D)** c()
18. **A)** *
19. **C)** Boolean
20. **A)** typeof()
21. **B)** factor()
22. **B)** Data frame
23. **C)** df[1,]
24. **A)** hist()
25. **A)** To apply a function over a matrix or array

1.9 REFERENCES

1. **R Documentation (Official)**
    - https://www.rdocumentation.org/
    - Provides comprehensive and detailed documentation on all functions, packages, and libraries in R.
2. **RStudio Website**
    - https://posit.co/products/rstudio/
    - Official page for RStudio IDE, including downloads, installation guides, and user resources.
3. **CRAN - Comprehensive R Archive Network**
    - https://cran.r-project.org/
    - CRAN is the official repository for R, containing all available packages and documentation for R.

4. **R for Data Science (Book and Online Resource) by Hadley Wickham and Garrett Grolemund**
    - https://r4ds.had.co.nz/
    - A popular book to learn R programming with a focus on data science. It's free to read online and covers topics from basics to advanced.
5. **The R Journal**
    - https://journal.r-project.org/
    - The official publication of the R community, which includes articles on new features, packages, and methodologies in R.
6. **Stack Overflow - R Programming Questions**
    - https://stackoverflow.com/questions/tagged/r
    - A great place to find solutions to R-related problems and programming challenges posted by the community.
7. **RStudio Community**
    - https://community.rstudio.com/
    - A platform for asking and answering RStudio-related questions, sharing resources, and engaging with the R community.
8. **R-Bloggers (Blogs and Tutorials)**
    - https://www.r-bloggers.com/
    - A blog aggregation website that brings together articles and tutorials on R programming from various contributors.
9. **Kaggle - R Programming Tutorials and Datasets**
    - https://www.kaggle.com/learn/r
    - Offers courses, tutorials, and practical exercises for mastering R in the context of machine learning and data analysis.
10. **DataCamp - R Programming Courses**
    - https://www.datacamp.com/courses/tech:r
    - A platform offering a variety of R programming courses, including introductory, intermediate, and advanced levels.
11. **GeeksforGeeks - R Programming Tutorials**
    - https://www.geeksforgeeks.org/r-programming-language/

UNIT II – WORKING WITH R
Working with R - Reading and writing data - R libraries - Functions and R programming – the If statement - looping: for, repeat, while - writing functions - function arguments and options – Basic R commands

Introduction : Working with R

UNIT OBJECTIVES

This unit introduces the fundamental concepts of R programming with a focus on data handling and automation. Students will learn how to read and write data in various formats such as CSV, TXT, and Excel files, and will become familiar with essential R libraries that support data manipulation and analysis. The unit covers the structure and syntax of R, including basic commands for working with vectors, matrices, and data frames. Students will explore how to write custom functions, manage function arguments and default options, and apply conditional logic using the if statement. Looping structures such as for, repeat, and while will be practiced to automate repetitive tasks. Through hands-on exercises, students will develop the skills to write efficient, modular, and readable R scripts, enabling them to handle a wide range of data analysis tasks confidently.

SECTION 2.1: WORKING WITH R

2.1.1 – READING AND WRITING DATA

In data analysis, the ability to import (read) and export (write) data is crucial because data typically comes from external sources such as CSV files, text files, Excel sheets, databases, or web APIs. R, as a statistical programming language, provides a variety of built-in functions and additional packages to handle different types of data formats.

Reading Data in R

Reading data refers to loading external data into R for analysis. The most common formats include:

a) Reading CSV Files

CSV (Comma-Separated Values) files are one of the most popular formats for storing data tables.

Syntax:

data <- read.csv(file = "filename.csv", header = TRUE, sep = ",")

- file: The name or path of the file.
- header: If TRUE, the first row contains column names.
- sep: The delimiter (default is a comma).

Example:

```
data <- read.csv("student_data.csv")
```

b) **Reading Text Files**

For text files with tabular data:

Syntax:

```
data <- read.table(file = "filename.txt", header = TRUE, sep = "\t")
```

c) Reading Excel Files

Excel files can be read using the readxl package:

Install and load the package:

```
install.packages("readxl")
library(readxl)
data <- read_excel("student_data.xlsx")
```

d) Other File Types

- read.delim() for tab-delimited files.
- readLines() for reading a file line by line (useful for unstructured data).

Writing Data in R

Writing data refers to saving R data objects (like data frames) into external files for later use, sharing, or reporting.

a) Writing CSV Files

Syntax:

```
write.csv(data, file = "output.csv", row.names = FALSE)
```

- row.names = FALSE: Prevents row numbers from being written as a separate column.

b) Writing Text Files

Syntax:

```
write.table(data, file = "output.txt", sep = "\t", row.names = FALSE)
```

c) Writing Excel Files

Using the openxlsx package:

Install and load the package:

```
install.packages("openxlsx")
library(openxlsx)
write.xlsx(data, file = "output.xlsx")
```

- Always **check the working directory** using getwd() and set it with setwd() to ensure R can find your files.
- Use **absolute paths** if files are located outside the working directory.
- Check **file encoding** if you are dealing with non-English characters (e.g., fileEncoding = "UTF-8").

EXAMPLE PROGRAM

```
# Step 1: Create a sample data frame
```

```
student_data <- data.frame(
  Roll_No = c(101, 102, 103, 104),
  Name = c("Alice", "Bob", "Charlie", "David"),
  Marks = c(88, 92, 79, 85),
  Grade = c("A", "A+", "B", "A")
)

# Step 2: Write the data frame to a CSV file
write.csv(student_data, file = "student_data.csv", row.names = FALSE)
cat("Data has been written to 'student_data.csv'\n")
# Step 3: Read the data back from the CSV file
read_student_data <- read.csv(file = "student_data.csv")
# Step 4: Display the data
print("Data read from 'student_data.csv':")
print(read_student_data)
```

Output:

```
Data has been written to 'student_data.csv'
[1] "Data read from 'student_data.csv':"
  Roll_No    Name Marks Grade
1     101   Alice    88     A
2     102     Bob    92    A+
3     103 Charlie    79     B
4     104   David    85     A
```

Functions for Reading Data into R

R provides a rich set of functions to read data from various formats and sources. These functions make it easy to import data for analysis, ranging from simple text files to R-specific binary objects.

Key Functions:

1. **read.table() and read.csv()**

   These are **fundamental functions** for reading tabular data (e.g., CSV or text files).

   - read.table(): For general tabular data.

- read.csv(): A variant with defaults suitable for comma-separated files (CSV).

2. **readLines()**
   Reads data **line by line** from a text file and stores each line as a character vector. Useful for **text analysis** or when reading non-tabular files.
3. **source()**
   Reads and executes **R code files**. This is very handy when you want to modularize your code into scripts.
4. **dget()**
   Reads **R objects** saved using dput(). It allows reading a single R object and restoring its exact structure.
5. **load()**
   Loads one or more R objects from an **R data file** (binary format). Typically used with .RData or .rda files.
6. **unserialize()**
   Reads a **binary R object** that has been serialized. This is a **low-level tool** useful for transmitting R objects or saving data in binary format.

Functions for Writing Data to Files

R also offers a variety of functions to **save data and objects** into files for later use or sharing.

Key Functions:

1. **write.table()**
   Writes tabular data (e.g., data frames) into text files, commonly in CSV format. You can control separators, row names, and more.
2. **writeLines()**
   Writes **character data line by line** to a file or connection. Best for writing plain text files.

3. **dump()**

   Outputs a **textual representation of multiple R objects**, allowing full metadata preservation.

4. **dput()**

   Outputs a **textual representation of a single R object** (ideal for sharing reproducible examples).

5. **save()**

   Saves one or more R objects into a **binary format file** for efficient storage and fast retrieval.

6. **serialize()**

   Converts an R object into a **binary stream** to be saved or transmitted. The binary format preserves complete precision and metadata.

EXAMPLE PROGRAM

```
# ==============================
# 1□. Create objects
# ==============================
# Data frame
df <- data.frame(ID = 1:3, Name = c("Alice", "Bob", "Charlie"), Score = c(90, 85, 88))
# Numeric vector
vec <- seq(10, 100, by = 10)
# List
lst <- list(Number = 42, Status = "Passed", Valid = TRUE)
# Matrix
mat <- matrix(1:9, nrow = 3, byrow = TRUE)
# Function
my_func <- function(x) {
  return(x * x)
}
# ==============================
# 2□ Write table, CSV, and lines to files
# ==============================
# Write table
write.table(df, file = "E:/full_demo_table.txt", sep = "\t", row.names = FALSE)
```

```
# Write CSV
write.csv(df, file = "E:/full_demo_table.csv", row.names = FALSE)
# Write Lines (from a character vector)
lines_content <- c("R is great!", "Learning file operations.", "Good luck!")
writeLines(lines_content, con = "E:/full_demo_lines.txt")
# ==============================
# 3 Read table, CSV, and lines from files
# ==============================
# Read table
read_df_table <- read.table("E:/full_demo_table.txt", header = TRUE, sep = "\t")
print(read_df_table)
# Read CSV
read_df_csv <- read.csv("E:/full_demo_table.csv")
print(read_df_csv)

# Read Lines
read_lines <- readLines("E:/full_demo_lines.txt")
print(read_lines)
# ==============================
# 4 Dump and source objects
# ==============================
# Dump multiple objects and a function
dump(c("df", "vec", "lst", "mat", "my_func"), file = "E:/full_dump_objects.R")
# Remove from environment
rm(df, vec, lst, mat, my_func)
# Source back
source("E:/full_dump_objects.R")
# Check all objects are restored
print(df)
print(vec)
print(lst)
print(mat)
print(my_func(5))  # Test the function
# ==============================
# 5 Save and load objects
# ==============================
# Save multiple objects in a binary .RData file
save(df, vec, lst, mat, my_func, file = "E:/full_saved_objects.RData")
# Remove them again to test load
rm(df, vec, lst, mat, my_func)
```

```
# Load back
load("E:/full_saved_objects.RData")
print(df)
print(vec)
print(lst)
print(mat)
print(my_func(6))
# ==============================
# 6 SaveRDS and readRDS (single object)
# ==============================
saveRDS(mat, file = "E:/full_saved_mat.rds")
mat_new <- readRDS("E:/full_saved_mat.rds")
print(mat_new)
# ==============================
# 7 Serialize and unserialize (list)
# ==============================
# Serialize list to raw vector and save it
serialized_lst <- serialize(lst, NULL)
save(serialized_lst, file = "E:/full_serialized_list.RData")
# Remove & reload
rm(lst)
load("E:/full_serialized_list.RData")
unserialized_lst <- unserialize(serialized_lst)
print(unserialized_lst)
# ==============================
# 8 BONUS: Save entire workspace
# ==============================
# Save full workspace image (all current objects)
save.image(file = "E:/full_workspace_image.RData")
# To test later: rm(list = ls()) and load("E:/full_workspace_image.RData")
# ==============================
# END of Program
# ==============================
```

OUTPUT

```
#-------------------------
# Section 3: Reading files
#-------------------------
  ID   Name Score
```

```
1  1  Alice    90
2  2    Bob    85
3  3 Charlie    88
  ID    Name Score
1  1  Alice    90
2  2    Bob    85
3  3 Charlie    88
[1] "R is great!"
[2] "Learning file operations."
[3] "Good luck!"
#--------------------------
# Section 4: Sourced objects
#--------------------------
  ID    Name Score
1  1  Alice    90
2  2    Bob    85
3  3 Charlie    88
[1]  10  20  30  40  50  60  70  80  90 100
$Number
[1] 42

$Status
[1] "Passed"
$Valid
[1] TRUE
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
[1] 25
#--------------------------
# Section 5: Loaded objects
#--------------------------
  ID    Name Score
1  1  Alice    90
2  2    Bob    85
3  3 Charlie    88
[1]  10  20  30  40  50  60  70  80  90 100
$Number
[1] 42
```

```
$Status
[1] "Passed"
$Valid
[1] TRUE
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
[1] 36
#-------------------------
# Section 6: Read RDS file
#-------------------------
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
#-------------------------
# Section 7: unserialize
#-------------------------
$Number
[1] 42
$Status
[1] "Passed"

$Valid
[1] TRUE
#-------------------------
# Bonus: save.image()
#-------------------------
Workspace image saved at:
E:/full_workspace_image.RData
```

2.1.2 – R LIBRARY

In R programming, **libraries** (also called **packages**) are collections of **functions**, **datasets**, and **documentation** that extend the basic functionality of R. While R's base installation provides many powerful tools for statistical computing and graphics, **libraries** make it possible to:

- Perform **specialized tasks** (e.g., machine learning, data wrangling, visualization).
- Access **pre-built algorithms** and **utilities** without writing code from scratch.
- Share and reuse code across projects or among researchers.

Library Name	Description	Example
base (save/load)	Save/load multiple R objects in .RData format (binary).	save(df, file = "data.RData")load("data.RData")
base (saveRDS/readRDS)	Save/load a single R object in .RDS format.	saveRDS(df, "data.rds")df2 <- readRDS("data.rds")
base (serialize/unserialize)	Serialize R objects to binary format for advanced storage or transfer.	serialize(df, NULL)unserialize(raw_data)
Readr	Tidyverse library for fast reading/writing of CSV, TSV files.	df <- read_csv("data.csv")write_csv(df, "out.csv")
data.table	Fastest reading/writing of large datasets using fread() and fwrite().	df <- fread("data.csv")fwrite(df, "out.csv")
Readxl	Read Excel files (.xls, .xlsx) without Java.	df <- read_excel("data.xlsx")
writexl	Write Excel files easily.	write_xlsx(df, "out.xlsx")
jsonlite	Work with JSON data (import/export).	df <- fromJSON("data.json")toJSON(df, pretty=TRUE)
rio	Simplify file import/export across	df <- import("data.csv")export(df, "out.xlsx")

	many formats (auto-detects file type).	
Qs	High-speed binary serialization of R objects (much faster than saveRDS()).	qsave(df, "data.qs")df2 <- qread("data.qs")
openxlsx	Read/write Excel files with support for formatting/styling.	write.xlsx(df, "styled.xlsx")
foreign	Read data from SPSS, Stata, SAS formats (older package).	df <- read.spss("data.sav", to.data.frame=TRUE)
haven	Tidyverse-friendly library for SPSS, Stata, SAS files.	df <- read_spss("data.sav")
feather	Fast cross-language data sharing format (Python-R).	write_feather(df, "data.feather")df2 <- read_feather("data.feather")
Arrow	Handle Apache Arrow / Parquet files (big data cross-language format).	write_parquet(df, "data.parquet")df2 <- read_parquet("data.parquet")
R.utils	Advanced tools: saving/loading R objects, compression, and utilities.	saveObject(df, "obj_file")df2 <- loadObject("obj_file")
filehash	Persistent key-value database for R objects (simple	db <- dbInit("mydb", type="DB1")dbInsert(db, "mydata", df)

	database-style storage).	
xml2	Read and write XML files.	doc <- read_xml("data.xml")write_xml(doc, "out.xml")
DBI + RSQLite	Work with SQLite databases (save/load data into SQL format).	con <- dbConnect(RSQLite::SQLite(), "db.sqlite")dbWriteTable(con, "mytable", df)
Library Name	Description	Example
Dplyr	Grammar of data manipulation: fast and readable data wrangling (select, filter, mutate, summarize).	df %>% filter(x > 5) %>% summarize(mean_x = mean(x))
ggplot2	Most popular library for creating high-quality visualizations based on the Grammar of Graphics.	ggplot(df, aes(x, y)) + geom_point()
esquisse	Drag-and-drop interface to build ggplot2 plots interactively.	esquisse::esquisser()
Shiny	Build interactive web applications directly from R.	shinyApp(ui, server)

mlr3	Modern machine learning framework with easy model training, tuning, and evaluation.	tsk = tsk("iris"); lrn = lrn("classif.rpart"); lrn$train(tsk)
lubridate	Simplifies date-time manipulation (parsing, extracting, updating).	ymd("2025-05-01") + days(5)
Rcrawler	Web crawling and scraping directly from R, automated website data extraction.	Rcrawler(Website = "https://example.com")
Knitr	Converts R scripts into dynamic reports (HTML, PDF, Word), supports RMarkdown.	{r} summary(df)
DT	Interactive data tables (filtering, sorting) for web applications and notebooks.	datatable(df)
Plotly	Create interactive and animated plots with hover info (extends ggplot2 or stand-alone).	plot_ly(df, x=~x, y=~y, type='scatter', mode='markers')
caret	Streamlines machine learning: training,	train(y ~ ., data=df, method="rf")

	tuning, resampling, and preprocessing.	
ROCR	Visualize classifier performance metrics like ROC curves, precision-recall.	pred <- prediction(prob, label); perf <- performance(pred, "tpr", "fpr"); plot(perf)
glmnet	Fit regularized regression models (LASSO, Ridge) efficiently for large datasets.	cv.glmnet(as.matrix(x), y, alpha=1)
markdown	Convert markdown text into HTML, integrates with RMarkdown.	markdownToHTML("input.md", "output.html")
RSQLite	Interface for working with SQLite databases (store/query data using SQL).	db <- dbConnect(SQLite(), "db.sqlite"); dbWriteTable(db, "iris", iris)

EXAMPLE PROGRAM

```
# Install required packages (only first time)
# install.packages(c("readr", "data.table", "readxl", "writexl", "jsonlite", "rio", "qs", "openxlsx",
#                    "foreign", "haven", "arrow", "feather", "xml2", "R.utils", "filehash", "DBI", "RSQLite",
#                    "dplyr", "ggplot2", "plotly", "DT", "mlr3", "caret", "ROCR", "glmnet",
#                    "lubridate", "Rcrawler", "knitr", "markdown", "esquisse"))
# Load libraries
library(readr); library(data.table); library(readxl); library(writexl); library(jsonlite)
library(rio); library(qs); library(openxlsx); library(foreign); library(haven)
library(arrow); library(feather); library(xml2); library(R.utils); library(filehash)
```

```
library(DBI); library(RSQLite)
library(dplyr); library(ggplot2); library(plotly); library(DT)
library(mlr3); library(caret); library(ROCR); library(glmnet)
library(lubridate); library(knitr); library(markdown)
# 1 Create a simple DataFrame
df <- data.frame(
  ID = 1:5,
  Name = c("Alice", "Bob", "Charlie", "David", "Eve"),
  Score = c(85, 92, 78, 90, 88),
  Date = Sys.Date() - 1:5
)
print("DataFrame created:")
print(df)
# 2 Save/load examples
# (base) save/load
save(df, file = "data.RData"); load("data.RData")
# (base) saveRDS/readRDS
saveRDS(df, "data.rds"); df_rds <- readRDS("data.rds")
# (base) serialize/unserialize
ser <- serialize(df, NULL); df_ser <- unserialize(ser)
# readr
write_csv(df, "data.csv"); df_csv <- read_csv("data.csv")
# data.table
fwrite(df, "data_dt.csv"); df_dt <- fread("data_dt.csv")
# writexl / readxl
write_xlsx(df, "data.xlsx"); df_xl <- read_excel("data.xlsx")
# jsonlite
json <- toJSON(df, pretty=TRUE); write(json, "data.json"); df_json <-
fromJSON("data.json")
# rio
export(df, "data_rio.xlsx"); df_rio <- import("data_rio.xlsx")
# qs
qsave(df, "data.qs"); df_qs <- qread("data.qs")
# openxlsx
write.xlsx(df, "styled.xlsx")
# foreign
write.foreign(df, datafile = "data.txt", codefile = "data.sps")
# haven
# Need a SPSS file to read; we skip writing for brevity
# df_spss <- read_spss("data.sav")
```

```
# feather
write_feather(df, "data.feather"); df_feather <- read_feather("data.feather")
# arrow (parquet)
write_parquet(df, "data.parquet"); df_parquet <- read_parquet("data.parquet")
# xml2
xml_doc <- as_xml_document(list(root=df)); write_xml(xml_doc, "data.xml");
xml_read <- read_xml("data.xml")
# R.utils
saveObject(df, "obj_file"); df_obj <- loadObject("obj_file")
# filehash
db <- dbInit("mydb", type="DB1"); dbInsert(db, "mydata", df); df_db <- dbFetch(db,
"mydata")
# DBI + RSQLite
con <- dbConnect(SQLite(), "db.sqlite"); dbWriteTable(con, "mytable", df); df_sql
<- dbReadTable(con, "mytable")
dbDisconnect(con)
# 3 Data manipulation + visualization
 dplyr
df_summary <- df %>% summarize(Avg_Score = mean(Score))
print("Average Score (dplyr):")
print(df_summary)
# ggplot2
p <- ggplot(df, aes(x = Name, y = Score)) + geom_bar(stat = "identity") +
ggtitle("Scores by Name")
print(p)
# plotly (interactive)
p_plotly <- ggplotly(p)
print("Plotly interactive plot (in RStudio Viewer)")
# DT
datatable(df)
# 4 Machine learning
# caret (train a simple model)
set.seed(123)
fit <- train(Score ~ ID, data = df, method = "lm")
print("caret model summary:")
print(fit)
# mlr3
tsk <- tsk("iris")
lrn <- lrn("classif.rpart")
```

```
lrn$train(tsk)
# ROCR (simple ROC example)
pred <- prediction(c(0.1, 0.4, 0.35, 0.8), c(0, 0, 1, 1))
perf <- performance(pred, "tpr", "fpr")
plot(perf, main="ROCR: ROC Curve")
# glmnet (regularized regression)
x <- as.matrix(df$ID)
y <- df$Score
cvfit <- cv.glmnet(x, y)
print("glmnet coefficients:")
print(coef(cvfit, s = "lambda.min"))
# 5 Miscellaneous
# lubridate
date_new <- ymd("2025-05-01") + days(5)
print(paste(" Date after 5 days:", date_new))
# Rcrawler (only structure here; won't run without internet)
# Rcrawler(Website = "https://httpbin.org")
# knitr / markdown (renders report)
write("## Report\n\nThis is a test report.", "report.md")
markdownToHTML("report.md", "report.html")
# esquisse (interactive plot builder - launches GUI)
# esquisse::esquisser()
# shiny (simple app structure)
# shinyApp(
#   ui = fluidPage(
#     titlePanel("Simple Shiny App"),
#     sidebarLayout(
#       sidebarPanel(sliderInput("obs", "Number of observations:", min = 1, max =
1000, value = 500)),
#       mainPanel(plotOutput("distPlot"))
#     )
#   ),
#   server = function(input, output) {
#     output$distPlot <- renderPlot({
#       hist(rnorm(input$obs))
#     })
#   }
# )
print(" All steps completed.")
```

2.1.3 – FUNCTIONS AND R PROGRAMMING

A **function** accepts **input arguments** and produces an **output** by executing **valid R commands** that are inside the function.

Why are functions useful?

- Functions are **useful when you want to perform a task multiple times** without rewriting the code.
- They help in **organizing code** and make it **reusable** and **modular.**

Creating a Function in R

Functions are created in R using the **function()** keyword.

General Syntax:

```
function_name <- function(arguments) {
  statements
  return(value)
}
```

Example:

```
add_numbers <- function(a, b) {
  result <- a + b
  return(result)
}
```

Call the function:

```
add_numbers(5, 7)  # Output: 12
```

Built-in Functions Example

```
# Create a sequence of numbers from 32 to 44.
print(seq(32, 44))
# Find mean of numbers from 25 to 82.
print(mean(25:82))
# Find sum of numbers from 41 to 68.
print(sum(41:68))
```

Explanation:

- seq(32, 44) generates a sequence of numbers from 32 to 44.
- mean(25:82) calculates the average of numbers from 25 to 82.
- sum(41:68) adds all numbers from 41 to 68.

2 User-defined Function

Example :

```
# Create a function to print squares of numbers in sequence.
new.function <- function(a) {
  for(i in 1:a) {
    b <- i^2
    print(b)
  }
}
# Call the function with argument 6.
new.function(6)
```

Explanation:

- new.function is a **user-defined function** that takes **one argument a**.
- The **loop (1:a)** runs from 1 to a, calculating the **square of each number** using i^2.
- Calling new.function(6) prints squares of numbers 1 to 6.

3 Calling Function Without Argument

```
# Create a function without an argument.
new.function <- function() {
  for(i in 1:5) {
    print(i^2)
  }
}
# Call the function.
new.function()
```

Explanation:

- This function **does not take any arguments**.
- It loops from 1 to 5 and prints squares of each number.
- new.function() calls the function.

4 Function with Arguments (by Position & by Name)

```
# Create a function with three arguments.
new.function <- function(a, b, c) {
  result <- a * b + c
  print(result)
}
# Call by position.
new.function(5, 3, 11)

# Call by name.
new.function(a = 11, b = 5, c = 3)
```

Explanation:

- The function takes **three arguments a, b, c**.
- It calculates a * b + c and prints the result.
- Call:
  - new.function(5, 3, 11) uses **positional arguments**.
  - new.function(a = 11, b = 5, c = 3) uses **named arguments**.

5 Function with Default Argument

```
# Create a function with default values.
new.function <- function(a = 3, b = 6) {
  result <- a * b
  print(result)
}
# Call without arguments (uses defaults).
new.function()
# Call with new values.
new.function(9, 5)
```

Explanation:

- a = 3 and b = 6 are **default values**.
- When you **don't supply arguments**, defaults are used.
- When you supply values, those **override defaults**.

6 Lazy Evaluation of Function

```
# Create a function with two arguments.
new.function <- function(a, b) {
  print(a^2)
  print(a)
  print(b)
}
# Call the function with only one argument.
new.function(6)
```

Explanation:

- a and b are arguments.
- **Lazy evaluation**: R **only evaluates b when it is needed**.
- Here:
  - It prints a^2 and a successfully.
  - But when it tries to print b, **error occurs because b is missing.**

7 Create & Call a Simple Function

```
# Create a simple function.
my_function <- function() {
  print("Hello World!")
}
# Call the function.
my_function()
```

Explanation:

- my_function **prints "Hello World!"** when called.
- You call it using my_function().

8 Function with One Argument

```
# Function with one argument.
my_function <- function(fname) {
  print(paste(fname, "Griffin"))
}
# Call with different arguments.
my_function("Peter")
my_function("Lois")
my_function("Stewie")
```

Explanation:

- The function takes fname as input.
- It **concatenates fname + "Griffin"**.
- Example:
  - my_function("Peter") → "Peter Griffin"

9 Function with Two Arguments

```
# Function with two arguments.
my_function <- function(fname, lname) {
  print(paste(fname, lname))
}
# Call with two arguments.
my_function("Peter", "Griffin")
```

Explanation:

- Function takes fname and lname.
- Combines them using paste().
- Example: my_function("Peter", "Griffin") → "Peter Griffin"

10. Default Parameter Value

```
# Function with default parameter.
my_function <- function(country = "Norway") {
  print(paste("I am from", country))
}
# Call with different arguments.
```

```
my_function("Sweden")
my_function("India")
my_function()  # uses default
my_function("USA")
```

Explanation:

- If you **don't pass an argument**, it uses "Norway" as default.
- You can **override the default** by passing another country name.

EXAMPLE

```
# ==========================================
# R PROGRAM: ALL FUNCTION TYPES EXAMPLES
# ==========================================
# 1 Built-in Functions ------------------------------------
# Create a sequence of numbers from 32 to 44
print("Built-in Function: seq(32, 44)")
print(seq(32, 44))
# Find mean of numbers from 25 to 82
print("Built-in Function: mean(25:82)")
print(mean(25:82))

# Find sum of numbers from 41 to 68
print("Built-in Function: sum(41:68)")
print(sum(41:68))
# 2 User-defined Function (with argument) ------------------
# Function to print squares of numbers up to 'a'
print("User-defined Function: Squares up to a number")
new.function1 <- function(a) {
  for(i in 1:a) {
    b <- i^2
    print(b)
  }
}
new.function1(6)
# 3 Function without Argument ------------------------------
# Function that prints squares of numbers from 1 to 5
print("User-defined Function (No Arguments): Squares of 1 to 5")
```

```
new.function2 <- function() {
  for(i in 1:5) {
    print(i^2)
  }
}
new.function2()
# 4 Function with Multiple Arguments ----------------------
# Function with 3 arguments
print("User-defined Function: Multiply and Add (3 args)")
new.function3 <- function(a, b, c) {
  result <- a * b + c
  print(result)
}
# Call by position
new.function3(5, 3, 11)
# Call by name
new.function3(a = 11, b = 5, c = 3)
# 5 Function with Default Arguments ------------------------
# Function with default values
print("User-defined Function: Default Argument Example")
new.function4 <- function(a = 3, b = 6) {
  result <- a * b
  print(result)
}
# Call without arguments (uses defaults)
new.function4()
# Call with values
new.function4(9, 5)
# 6 Lazy Evaluation ----------------------------------------
# Function to show lazy evaluation
print("User-defined Function: Lazy Evaluation Example")
new.function5 <- function(a, b) {
  print(a^2)
  print(a)
  if (!missing(b)) {
     print(b)
  } else {
     print("b is missing")
  }
}
```

```
# Call with only one argument (b is missing)
new.function5(6)
# 7 Simple Function (No Arguments) -------------------------
# Simple function that prints Hello World
print("Simple Function: Hello World")
my_function1 <- function() {
  print("Hello World!")
}
my_function1()
# 8 Function with One Argument -----------------------------
# Function with one argument
print("Function with One Argument: fname + Griffin")
my_function2 <- function(fname) {
  print(paste(fname, "Griffin"))
}
my_function2("Peter")
my_function2("Lois")
my_function2("Stewie")
# 9 Function with Two Arguments ----------------------------
# Function with two arguments
print("Function with Two Arguments: Full Name")
my_function3 <- function(fname, lname) {
  print(paste(fname, lname))
}
my_function3("Peter", "Griffin")
# 10 Function with Default Parameter -------------------------
# Function with default country parameter
print("Function with Default Parameter: Country Example")
my_function4 <- function(country = "Norway") {
  print(paste("I am from", country))
}
my_function4("Sweden")
my_function4("India")
my_function4()  # uses default
my_function4("USA")
# Function with Return Value -------------------------------
# Function that returns a value
print("Function with Return Value: 5 * x")
my_function5 <- function(x) {
  return(5 * x)
```

```
}
print(my_function5(3))
print(my_function5(5))
print(my_function5(9))
# ========================================
# END OF PROGRAM
```

2.1.4 IF STATEMENT – CONDITIONAL STATEMENTS

In R, **conditional statements** allow you to control the flow of execution based on specific conditions. The main types of conditional statements in R are if, if-else, if-else if-else, and the switch statement.

1. if Statement:

The if statement is used to execute a block of code if a specific condition is TRUE.

Syntax:

```
if (condition) {
  # Code to execute if condition is TRUE
}
```

Example:

```
x <- 10
if (x > 0) {
  print("x is positive")
}
```

2. if-else Statement:

The if-else statement allows you to execute one block of code if the condition is TRUE and another block if the condition is FALSE.

Syntax:

```
if (condition) {
  # Code to execute if condition is TRUE
} else {
  # Code to execute if condition is FALSE
}
```

Example:

```
x <- -5
if (x > 0) {
```

```
  print("x is positive")
} else {
  print("x is negative or zero")
}
```

3. if-else if-else Statement:

The if-else if-else statement allows you to check multiple conditions. If the first condition is FALSE, it will check the next else if condition, and so on. If none of the conditions are true, the else block will execute.

Syntax:

```
if (condition1) {
  # Code to execute if condition1 is TRUE
} else if (condition2) {
  # Code to execute if condition2 is TRUE
} else {
  # Code to execute if none of the conditions are TRUE
}
```

Example:

```
x <- 7
if (x < 0) {
  print("x is negative")
} else if (x == 0) {
  print("x is zero")
} else {
  print("x is positive")
}
```

4. switch Statement:

The switch statement evaluates an expression and matches the result to a list of options.

Syntax:

```
switch(expression,
  case1 = result1,
  case2 = result2,
  ...
  default = result)
```

Example:

```
month <- 3
season <- switch(month,
  "1" = "Winter",
  "2" = "Winter",
  "3" = "Spring",
  "4" = "Spring",
  "5" = "Spring",
  "6" = "Summer",
  "7" = "Summer",
  "8" = "Summer",
  "9" = "Fall",
  "10" = "Fall",
  "11" = "Fall",
  "12" = "Winter",
  "Unknown month"
)
print(season)
```

Complete Program for All Conditional Statements:

```
# 1. If Statement Example
x <- 10
if (x > 0) {
  print("x is positive")
}
# 2. If-Else Statement Example
x <- -5
if (x > 0) {
  print("x is positive")
} else {
  print("x is negative or zero")
}
# 3. If-Else if-Else Statement Example
x <- 7
if (x < 0) {
  print("x is negative")
} else if (x == 0) {
  print("x is zero")
} else {
```

```
  print("x is positive")
}
# 4. Switch Statement Example
month <- 3
season <- switch(month,
  "1" = "Winter",
  "2" = "Winter",
  "3" = "Spring",
  "4" = "Spring",
  "5" = "Spring",
  "6" = "Summer",
  "7" = "Summer",
  "8" = "Summer",
  "9" = "Fall",
  "10" = "Fall",
  "11" = "Fall",
  "12" = "Winter",
  "Unknown month"
)
print(season)
```

Output:

```
[1] "x is positive"
[1] "x is negative or zero"
[1] "x is positive"
[1] "Spring"
```

Explanation of the Code:

1. **If Statement**: Checks if x is greater than 0 and prints "x is positive" if true.
2. **If-Else Statement**: Checks if x is greater than 0, and if not, prints "x is negative or zero".
3. **If-Else If-Else Statement**: Checks multiple conditions for x, printing appropriate messages depending on whether x is negative, zero, or positive.
4. **Switch Statement**: Evaluates the value of month and prints the corresponding season.

- The **if** statement is used to check a condition and execute code based on it.
- The **if-else** allows two possible branches of execution: one for the condition being TRUE and another for FALSE.
- The **if-else if-else** structure checks multiple conditions sequentially.
- The **switch** statement provides an alternative way of selecting from multiple cases based on a single expression.

2.1.5 – LOOPING – FOR, WHILE , REPEAT

Looping in R:

In R, loops are used to execute a block of code repeatedly based on a condition. There are three main types of loops: **for**, **while**, and **repeat**.

1. for Loop:

The **for** loop is used to iterate over a sequence (like a vector, list, or a range of numbers).

Syntax:

```
for (variable in sequence) {
  # Code to execute in each iteration
}
```

- variable: the loop variable that takes the value of each element in the sequence.
- sequence: a vector, list, or range of values.

Example:

```
# Example: Print numbers from 1 to 5
for (i in 1:5) {
  print(i)
}
```

Output:

```
[1] 1
[1] 2
[1] 3
```

```
[1] 4
[1] 5
```

2. while Loop:

The **while** loop repeats a block of code as long as a specified condition is TRUE.

Syntax:

```
while (condition) {
  # Code to execute as long as condition is TRUE
}
```

- condition: a logical expression that is evaluated before each iteration. If the condition is TRUE, the code block is executed.

Example:

```
# Example: Print numbers from 1 to 5 using while loop
i <- 1
while (i <= 5) {
  print(i)
  i <- i + 1  # Increment i to avoid infinite loop
}
```

Output:

```
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

3. repeat Loop:

The **repeat** loop is used to repeat a block of code indefinitely until a break condition is met. It does not check a condition before each iteration.

Syntax:

```
repeat {
  # Code to execute repeatedly
```

```
  if (condition) {
    break  # Exit the loop if condition is met
  }
}
```

- The loop runs indefinitely until a break statement is executed.

Example:

```
# Example: Print numbers from 1 to 5 using repeat loop
i <- 1
repeat {
  print(i)
  i <- i + 1
  if (i > 5) {
    break  # Exit the loop when i is greater than 5
  }
}
```

Output:

```
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

Summary of Syntax and Examples:

```
# 1. FOR Loop Example
print("FOR Loop:")
for (i in 1:5) {
  print(i)
}
# 2. WHILE Loop Example
print("WHILE Loop:")
i <- 1
while (i <= 5) {
  print(i)
  i <- i + 1
}
# 3. REPEAT Loop Example
print("REPEAT Loop:")
```

```
i <- 1
repeat {
  print(i)
  i <- i + 1
  if (i > 5) {
    break
  }
}
```

Output for All Three Loops:

```
FOR Loop:
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
WHILE Loop:
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
REPEAT Loop:
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

Explanation:

- **for loop**: Iterates over a sequence of values (in this case, 1 to 5) and prints each number.
- **while loop**: Checks the condition i <= 5 before each iteration and increments i until the condition is no longer true.

- **repeat loop**: Repeats the block of code indefinitely until the condition i > 5 is met, at which point it exits the loop using the break statement.

2.1.6 – WRITING FUNCTIONS

A **function** in R is a block of reusable code designed to perform a specific task. Functions help to **avoid code repetition** and make the code **modular and organized**.

Basic Structure of a Function:

```
function_name <- function(arguments) {
  # body of the function
  # R statements
  return(value)
}
```

- **function_name**: The name you give to the function.
- **arguments**: (Optional) Input values (parameters) that the function takes.
- **body**: The code block where tasks are performed.
- **return()**: (Optional) The output that the function sends back.

Example 1: Simple Function (No Arguments)

```
# Function that prints a message
my_function <- function() {
  print("Hello, this is my first function!")
}
# Call the function
my_function()
```

Output:

```
[1] "Hello, this is my first function!"
```

Example 2: Function with Arguments

```
# Function that adds two numbers
add_numbers <- function(a, b) {
  result <- a + b
  return(result)
}
# Call the function
add_numbers(5, 3)
```

Output:

[1] 8

Example 3: Function with Default Argument

```
# Function with default argument
greet <- function(name = "Guest") {
  print(paste("Hello,", name))
}

# Call with and without argument
greet("Nivetha")
greet()
```

Output:

```
[1] "Hello, Nivetha"
[1] "Hello, Guest"
```

Example 4: Function with Return Value

```
# Function that multiplies a number by 10
multiply_by_10 <- function(x) {
  return(x * 10)
}
# Call and store the result
result <- multiply_by_10(7)
print(result)
```

Output:

[1] 70

Example 5: Function with Multiple Statements

```
# Function that checks if a number is even or odd
check_number <- function(n) {
  if (n %% 2 == 0) {
    print(paste(n, "is even"))
  } else {
    print(paste(n, "is odd"))
  }
}
# Call the function
check_number(4)
check_number(7)
```

Output:

```
[1] "4 is even"
```

```
[1] "7 is odd"
```

Example 6: Function Using a Loop Inside

```
# Function that prints squares of numbers up to n
print_squares <- function(n) {
  for (i in 1:n) {
    print(i^2)
  }
}
# Call the function
print_squares(5)
```

Output:

```
[1] 1
[1] 4
[1] 9
[1] 16
[1] 25
```

Important Points:

1. **Functions can be saved in files** and sourced into your R environment using source("filename.R").
2. **Arguments can have default values.**
3. **R supports anonymous functions** (functions without a name) often used with apply() family.

Example

```
# ====================================
# R Program: Different Function Types
# ====================================
# 1□ Simple Function
hello_func <- function() {
  print("Hello World!")
}
hello_func()
# 2□ Function with Arguments
```

```
add_func <- function(a, b) {
  result <- a + b
  return(result)
}
print(add_func(10, 20))
# 3 Function with Default Argument
greet_func <- function(name = "Guest") {
  print(paste("Hello,", name))
}
greet_func("Alice")
greet_func()
# 4 Function with Loop (For Loop inside Function)
square_func <- function(n) {
  print(paste("Squares up to", n))
  for (i in 1:n) {
    print(i^2)
  }
}
square_func(4)
# 5 Function with If-Else Logic
even_odd_func <- function(x) {
  if (x %% 2 == 0) {
    print(paste(x, "is Even"))
  } else {
    print(paste(x, "is Odd"))
  }
}
even_odd_func(5)
even_odd_func(8)
# ====================================
# End of Program
# ====================================
```

Output:

```
[1] "Hello World!"
[1] 30
[1] "Hello, Alice"
[1] "Hello, Guest"
[1] "Squares up to 4"
[1] 1
[1] 4
```

```
[1] 9
[1] 16
[1] "5 is Odd"
[1] "8 is Even"
```

2.1.7 – FUNCTION ARGUMENTS AND OPTIONS

What Are Function Arguments?

- **Arguments** are **input values** you pass to a function when you call it.
- They let you control how the function behaves.

Example:

```
my_sum <- function(a, b) {
  return(a + b)
}
my_sum(10, 20)
```

Output:

```
[1] 30
```

Here, 10 and 20 are **arguments** to the function.

How to Define Function Arguments?

The general structure is:

```
function_name <- function(arg1, arg2, ...) {
  # body of the function
}
```

- You can have **zero, one, or multiple arguments.**
- You can also specify **default values**.

Default Arguments (Options)

You can **set default values** for arguments in the function definition.

Example:

```
greet <- function(name = "Guest") {
  print(paste("Hello,", name))
}
# Call with argument
greet("Alice")
# Call without argument (uses default)
greet()
```

Output:

```
[1] "Hello, Alice"
[1] "Hello, Guest"
```

If no argument is passed, the default is used.

Named vs. Positional Arguments

- **Positional Arguments**: Based on the order.
- **Named Arguments**: You specify the name when calling.

Example:

```
my_info <- function(name, age) {
  print(paste(name, "is", age, "years old"))
}

# Call by position
my_info("John", 25)
# Call by name
my_info(age = 30, name = "Mary")
```

Output:

```
[1] "John is 25 years old"
[1] "Mary is 30 years old"
```

Variable Number of Arguments: ... (Ellipsis)

The ... allows passing a **variable number of arguments.**

Example:

```
sum_all <- function(...) {
  numbers <- c(...)
  print(sum(numbers))
}
sum_all(1, 2, 3, 4, 5)
```

Output:

```
[1] 15
```

Missing Arguments

The missing() function can check if an argument was provided.

Example:

```
check_args <- function(a, b) {
  if (missing(b)) {
    print("b is missing")
  } else {
    print(a + b)
  }
}
check_args(5)
check_args(5, 3)
```

Output:

```
[1] "b is missing"
[1] 8
```

Full Example Program:

```
# Function with multiple argument types
my_function <- function(a, b = 10, ...) {
  # Check if b is missing
  if (missing(b)) {
    print("Argument b is missing")
  } else {
    print(paste("a + b =", a + b))
  }
```

```
  # Sum all extra arguments
  extra_args <- c(...)
  if (length(extra_args) > 0) {
    print(paste("Sum of extra args:", sum(extra_args)))
  }
}
# Call examples
my_function(5)
my_function(5, 20)
my_function(5, 20, 1, 2, 3)
```

Output:

```
[1] "a + b = 15"
[1] "a + b = 25"
[1] "a + b = 25"
[1] "Sum of extra args: 6"
```

2.1.8 – BASIC R COMMANDS

Command	Definition	Example	Output
c()	Combine values into a vector.	v <- c(1, 2, 3)	[1] 1 2 3
length()	Find length of vector/list.	length(v)	3
class()	Get class/type of an object.	class(v)	"numeric"
typeof()	Get internal type of an object.	typeof(v)	"double"
sum()	Sum of vector.	sum(v)	6
mean()	Mean of values.	mean(v)	2
median()	Median of values.	median(v)	2
min(), max()	Minimum/maximum value.	min(v)	1
sort()	Sort vector.	sort(c(3, 1, 2))	[1] 1 2 3
unique()	Unique values.	unique(c(1, 1, 2))	[1] 1 2

table()	Count frequencies.	table(c("A", "B", "A"))	A: 2, B: 1
data.frame()	Create data frame.	df <- data.frame(x=1:2, y=c("a", "b"))	x y \n 1 1 a \n 2 2 b
str()	Show structure.	str(df)	'data.frame': 2 obs. of 2 variables
summary()	Summary statistics.	summary(df)	Min, 1st Qu., Median, Mean...
list()	Create list.	lst <- list(a=1, b=2)	List of 2 \n $a: 1 \n $b: 2
matrix()	Create matrix.	matrix(1:4, nrow=2)	1 3 \n 2 4
array()	Create array.	array(1:8, dim=c(2,2,2))	3D array
dim()	Get dimensions.	dim(df)	2 2
colnames()	Get column names.	colnames(df)	"x" "y"
rownames()	Get row names.	rownames(df)	"1" "2"
apply()	Apply function over rows/columns.	apply(matrix(1:4,2), 1, sum)	[1] 4 6
paste()	Combine strings.	paste("A", "B")	"A B"
cat()	Concatenate and print.	cat("Hello", 123)	Hello 123
help()	Get help.	help(mean)	Help page opens
install.packages()	Install package.	install.packages("ggplot2")	Installs ggplot2
library()	Load package.	library(ggplot2)	Loads ggplot2
plot()	Simple plot.	plot(1:5, c(2,4,6,8,10))	Opens plot window
hist()	Histogram.	hist(c(1,1,2,3,3,3,4))	Histogram plot

boxplot()	Boxplot.	boxplot(c(5,6,7,8,9))	Boxplot plot
setwd()	Set working directory.	setwd("C:/myfolder")	No output
getwd()	Get working directory.	getwd()	"C:/myfolder"
read.csv()	Read CSV file.	read.csv("file.csv")	DataFrame loaded
write.csv()	Write CSV file.	write.csv(df, "out.csv")	CSV file created
NA	Missing value.	x <- c(1, NA, 3)	1 NA 3
is.na()	Check for NA.	is.na(x)	FALSE TRUE FALSE
na.omit()	Remove NA rows.	na.omit(x)	1 3
ifelse()	Vectorized if/else.	ifelse(c(1,2,3)>2, "Yes", "No")	"No" "No" "Yes"
factor()	Create categorical variable.	factor(c("Low", "Med", "High"))	Factor w/ levels: High Low Med

Complete Program Using Many Commands:

```
# 1. Create a vector
v <- c(10, 20, 30, 40, 50)
print("Vector v:")
print(v)
# 2. Basic operations
print(paste("Length:", length(v)))
print(paste("Sum:", sum(v)))
print(paste("Mean:", mean(v)))
print(paste("Max:", max(v)))
# 3. Data frame
students <- data.frame(Name = c("Alice", "Bob", "Charlie"),
                       Score = c(85, 92, 78),
                       Passed = c(TRUE, TRUE, FALSE))
print("Data Frame:")
```

```
print(students)
# 4. Add new column using ifelse
students$Grade <- ifelse(students$Score >= 90, "A",
                         ifelse(students$Score >= 80, "B", "C"))
print("Data Frame with Grades:")
print(students)
# 5. Use apply
m <- matrix(1:9, nrow=3)
row_sums <- apply(m, 1, sum)
print("Row sums of matrix m:")
print(row_sums)
# 6. Use factor
levels <- factor(students$Grade)
print("Factor levels:")
print(levels)
# 7. Plot
plot(students$Score, type = "o", main = "Student Scores", xlab = "Student", ylab
= "Score")
# 8. NA handling
scores_with_na <- c(100, 90, NA, 80)
print("Is NA:")
print(is.na(scores_with_na))
print("Mean without NA:")
print(mean(scores_with_na, na.rm = TRUE))
```

Expected Output:

```
[1] "Vector v:"
[1] 10 20 30 40 50
[1] "Length: 5"
[1] "Sum: 150"
[1] "Mean: 30"
[1] "Max: 50"
[1] "Data Frame:"
    Name Score Passed
1   Alice    85   TRUE
2     Bob    92   TRUE
3 Charlie    78  FALSE
[1] "Data Frame with Grades:"
    Name Score Passed Grade
1   Alice    85   TRUE     B
```

```
2     Bob    92   TRUE     A
3 Charlie    78  FALSE     C
[1] "Row sums of matrix m:"
[1] 15 18 21
[1] "Factor levels:"
[1] B A C
Levels: A B C
[1] "Is NA:"
[1] FALSE FALSE  TRUE FALSE
[1] "Mean without NA:"
[1] 90
```

2.2 Let Us Sum Up

In this module, we explored how to work with R, including reading and writing data using functions like read.csv() and write.csv(). We learned about R libraries and how to load them using library(). We practiced writing user-defined functions with arguments and default values. The use of conditional statements such as if, if...else, and nested conditions was explained. We also covered looping structures including for, while, and repeat loops. Additionally, we reviewed basic R commands for sequences, summary statistics, and data manipulation. All these concepts were combined in a complete program for better understanding.

2.3 Check Your Progress

1. Which of the following functions is used to read a CSV file in R?

A) read.txt()

B) read.csv()

C) read.table()

D) read.file()

2. Which R function is used to write data to a CSV file?

A) write.csv()

B) write.file()

C) write.table()

D) save.csv()

3. Which command is used to install a new package in R?

A) library()

B) install.library()

C) install.packages()

D) include.packages()

4. Which function loads an installed R package?

A) install()

B) load()

C) library()

D) attach()

5. In R, which symbol is used to assign a value to a variable?

A) ==

B) <-

C) =

D) =>

6. Which function is used to create a sequence of numbers?

A) rep()

B) seq()

C) sum()

D) mean()

7. What is the output of mean(c(2, 4, 6))?

A) 2

B) 3

C) 4

D) 6

8. Which looping statement runs for a specific number of iterations?

A) while

B) repeat

C) for

D) goto

9. Which loop continues until a condition becomes false?

A) for

B) repeat

C) while

D) loop

10. Which loop in R is guaranteed to execute at least once?

A) for

B) while

C) repeat

D) if

11. What is the correct syntax for an if...else statement in R?

A) if (condition) { } else { }

B) if condition then else

C) if condition { else }

D) if ( ) else ( )

12. What is the output of if (5 > 3) print("Yes")?

A) No output

B) Yes

C) Error

D) 5 > 3

13. Which function returns the number of rows in a data frame?

A) colnames()

B) rownames()

C) nrow()

D) length()

14. What does length(c(1, 2, 3, 4)) return?

A) 3

B) 4

C) 2

D) 5

15. How do you define a function in R?

A) func()

B) new.function()

C) function()

D) create.function()

16. Which of the following is a valid R function definition?

A) myfun <- function(x) { x + 1 }

B) myfun = function(x) x + 1

C) Both A and B

D) None

17. What is the default data type of a vector in R?

A) numeric

B) character

C) logical

D) integer

18. What is the result of sum(1:5)?

A) 10

B) 12

C) 15

D) 20

19. Which command gives the structure of an R object?

A) class()

B) typeof()

C) str()

D) summary()

20. How do you concatenate strings in R?

A) paste()

B) concat()

C) combine()

D) strcat()

21. What is the output of paste("Hello", "World")?

A) HelloWorld

B) Hello World

C) Hello, World

D) Error

22. Which of the following will create a numeric vector?

A) c(1, 2, 3)

B) list(1, 2, 3)

C) matrix(1:3)

D) data.frame(1:3)

23. Which R function repeats a vector multiple times?

A) repeat()

B) rep()

C) seq()

D) loop()

24. Which R function checks the class of an object?

A) typeof()

B) is()

C) class()

D) getclass()

25. Which is a valid way to comment in R?

A) /* comment */

B) // comment

C) # comment

D) -- comment

26. Which function calculates the square root in R?

A) sqrt()

B) power()

C) root()

D) sqroot()

27. What is the result of rep(2, 3)?

A) 2 3

B) 3 3 3

C) 2 2 2

D) 2 3 2

28. Which operator is used for logical AND in R?

A) &&

B) &

C) Both A and B

D) AND()

29. What will while (FALSE) { print("Hello") } do?

A) Print Hello

B) Run forever

C) Do nothing

D) Error

30. What is the purpose of the break statement in loops?

A) Skip to next iteration

B) Stop the loop immediately

C) Restart loop

D) Continue forever

2.4 Unit Summary:

This unit introduces the basics of working with R, covering reading and writing data using functions like read.csv() and write.csv(). It explains how to manage R libraries for extended functionality and how to create and use functions in R programming. The module describes conditional statements (if, if...else) and looping structures such as for, while, and repeat for iterative tasks. It also details how to write custom functions, handle arguments and default options, and emphasizes mastering fundamental R commands for data manipulation and analysis. By the end of the unit, learners gain a foundational understanding of R programming, enabling them to handle data, automate processes, and build reusable code efficiently.

2.5 GLOSSARY

- **R:** A programming language and environment for statistical computing and graphics.
- **Library:** A collection of functions and datasets in R that can be loaded using library().
- **Function:** A reusable block of code that performs a specific task; created using the function() keyword.
- **If Statement:** A conditional statement used to execute code based on whether a condition is true.
- **For Loop:** A control structure that repeats a block of code for a set number of iterations.
- **While Loop:** A loop that continues to execute as long as a specified condition is true.
- **Repeat Loop:** A loop that executes repeatedly until a break condition is met.
- **Arguments:** Inputs provided to a function to customize its behavior.
- **Data Frame:** A table or 2D array structure used in R for storing data.
- **read.csv():** A function used to read data from a CSV file into R.
- **write.csv():** A function used to write data to a CSV file.
- **Vector:** A basic data structure in R that holds elements of the same type.
- **Binary Search:** An efficient algorithm to search sorted data by repeatedly dividing the search interval in half.
- **Recursion:** A programming technique where a function calls itself to solve sub-problems.
- **Lazy Evaluation:** A concept where arguments to functions are evaluated only when needed.

2.6 SELF ASSESSMENT QUESTIONS

1. What is R and what are its key features?
2. How do you read data from a CSV file in R?
3. Explain how to write data to a file in R.

4. What is the purpose of using libraries in R? Name two common libraries.
5. How do you create a user-defined function in R?
6. Write the syntax of an if-else statement in R.
7. What is the difference between for, while, and repeat loops in R?
8. What is lazy evaluation in R functions?
9. How do you pass default arguments in an R function?
10. What is the use of mean() and sum() functions?
11. Explain how to create a vector in R with an example.
12. What is the role of the library() function?
13. Describe the use of the print() function in R.
14. How can you handle missing values in R?
15. Write a program that prints numbers from 1 to 10 using a for loop.
16. Explain the difference between read.table() and read.csv().
17. What is the output of the function seq(1, 10, by = 2)?
18. How can you combine two vectors in R?
19. What is the importance of comments in an R program?
20. Give an example of a function that returns a value.
21. What is recursion? Give an example from R.
22. How do you break out of a repeat loop?
23. What is the use of the paste() function in R?
24. How do you check the structure of a dataset in R?
25. What is the syntax for a while loop in R?
26. How can you install a new package in R?
27. What is a data frame? How is it different from a matrix?
28. Write a simple function that takes two numbers and returns their product.
29. Explain the difference between positional and named arguments.
30. What is binary recursion? Provide an example.

2.7 Activities / Exercises / Case Studies

1 **Activity:**

Objective: Read a dataset and display the first 10 rows.

Task: Use read.csv() to import a dataset and apply head() to view it.

2 **Exercise:**

Objective: Create a function that calculates the area of a rectangle.

Task: Write a function with arguments length and width, returning length * width.

3 **Activity:**

Objective: Practice looping in R.

Task: Use a for loop to print all even numbers between 1 and 20.

4 **Exercise:**

Objective: Use conditional statements.

Task: Write an if-else block that checks if a number is positive, negative, or zero.

5 **Activity:**

Objective: Apply a library function.

Task: Use the ggplot2 library to create a simple bar plot of sample data.

6 **Exercise:**

Objective: Handle missing data.

Task: Create a vector with NA values and use na.omit() to clean it.

7 **Activity:**

Objective: Explore function arguments.

Task: Create a function with default arguments and call it with and without arguments.

Case Studies:

Case Study 1: COVID-19 Data Analysis

Scenario: You are provided with a CSV file containing COVID-19 cases data by country.

Task:

- Read the data into R.
- Use basic R commands to summarize total cases and deaths.
- Write a function that calculates the case fatality rate.
- Visualize the top 5 countries using a bar chart.

Case Study 2: Sales Data Report

Scenario: A retail company wants to analyze their monthly sales data.

Task:

- Import the sales CSV file.
- Use loops to calculate the monthly total sales.
- Use conditional statements to highlight months with sales below target.
- Save the cleaned and analyzed data to a new CSV file.

Case Study 3: Student Marks Analysis

Scenario: Analyze student marks stored in a data frame.

Task:

- Write a function that classifies students as Pass/Fail based on marks.
- Use loops to print all students who scored above 75%.
- Apply basic R commands to find the highest and lowest marks.

2.8 Answers for Check Your Progress

1. b) read.csv()
2. a) write.csv()
3. c) install.packages()
4. c) library()
5. b) <-

6. b) seq()
7. c) 4
8. c) for
9. c) while
10. c) repeat
11. a) if (condition) { } else { }
12. b) Yes
13. c) nrow()
14. b) 4
15. c) function()
16. c) Both A and B
17. a) numeric
18. c) 15
19. c) str()
20. a) paste()
21. b) Hello World
22. a) c(1, 2, 3)
23. b) rep()
24. c) class()
25. c) # comment
26. a) sqrt()
27. c) 2 2 2
28. c) Both A and B
29. c) Do nothing
30. b) Stop the loop immediately

2.9 REFERENCES

1. Books:

1. **"R for Data Science" by Hadley Wickham & Garrett Grolemund**
    - Link: R for Data Science

2. **"The Art of R Programming" by Norman Matloff**
    - Link: The Art of R Programming
3. **"Advanced R" by Hadley Wickham**
    - Link: Advanced R
4. **"Hands-On Programming with R" by Garrett Grolemund**
    - Link: Hands-On Programming with R

2. Online Resources:

1. **R Project Documentation** Link: R Project
2. **CRAN (Comprehensive R Archive Network)**Link: CRAN
3. **DataCamp** Link: DataCamp R Courses
4. **Khan Academy – R Programming** Link: Khan Academy R Programming

3. Articles:

1. **"R for Beginners" by Roger D. Peng** Link: R for Beginners
2. **"A Beginner's Guide to R Programming" by DataCamp** Link: Beginner's Guide to R Programming

4. YouTube Channels:

1. **Data Science Dojo** Link: Data Science Dojo YouTube
2. **RStudio** Link: RStudio YouTube Channel

5. Tutorials and Cheat Sheets:

1. **RStudio Cheat Sheets** Link: RStudio Cheat Sheets
2. **"R Programming Cheat Sheet" by DataCamp** Link: DataCamp R Cheat Sheet

Reading and Getting Data into R (External Data)

UNIT III – INTRODUCTION

UNIT OBJECTIVES

In this unit, students will learn how to effectively read and import various types of external data into R for analysis. They will gain the skills to import data from common formats such as CSV, XML, JSON, Excel, and databases. Students will explore functions like read.csv() for CSV files, xml2 for XML files, and jsonlite for JSON, as well as how to retrieve data from online sources using web APIs. Additionally, the unit covers reading data from databases using the DBI package and importing Excel files with packages like

readxl and openxlsx. In the second part of the unit, students will focus on data visualization, learning how to create various types of charts and graphs. They will learn how to create histograms, boxplots, bar charts, line graphs, scatterplots, and pie charts using base R functions and ggplot2. The unit also covers how to customize visualizations by modifying labels, titles, axes, and colors, allowing students to effectively present and interpret their data. By the end of this unit, students will be proficient in both importing and visualizing data in R, essential skills for any data analysis project.

SECTION 3.1: READING AND GETTING DATA INTO R (EXTERNAL DATA)

3.1.1 – READING AND GETTING DATA INTO R (EXTERNAL DATA) USING CSV FILES

The read.csv() function in R is one of the most commonly used methods to import data stored in CSV (Comma Separated Values) files into R. The function reads the contents of the CSV file and imports it as a data frame, making it easy to analyze and manipulate the data in R. The function also provides a variety of optional parameters that allow users to customize the data import process, such as specifying column headers, delimiters, decimal characters, and more.

Syntax:

```
read.csv(file, header = TRUE, sep = ",", dec = ".")
```

Parameters:

- **file**: Specifies the path or URL of the file to be read.
- **header**: A logical value (TRUE/FALSE) indicating whether the first row of the file should be treated as column names. Default is TRUE.
- **sep**: The field separator character. The default is a comma (,), but this can be changed to accommodate files with different delimiters.
- **dec**: The character used for decimal points in the file (typically a period .).

Examples of Using read.csv()

Example 1: Reading a CSV File from the Same Folder

```
data <- read.csv("CSVFileExample.csv", header = FALSE, sep = "\t")
head(data)
```

In this example, the CSV file is assumed to be in the same directory as the R script. The data is read without column names (header = FALSE) and uses tab as the delimiter (sep = "\t"). The head() function displays the first few rows of the imported data.

Output:

```
  V1  V2  V3
1 10  20  30
2 40  50  60
```

Example 2: Reading a CSV File from a Different Directory

```
x <- read.csv("D://Datas//myfile.csv")
head(x)
```

In this example, the CSV file is read from a directory located at D://Datas//. The file path must be specified correctly to read the data.

Output:

```
  Name Age Salary
1 John  28  50000
2 Anna  35  60000
```

Example 3: Reading a CSV File with a Different Delimiter

```
data <- read.csv("path/to/your/file.csv", sep = ";")
head(data)
```

Here, the sep parameter is set to ";", indicating that the file uses semicolons as the field delimiter instead of commas.

Output:

```
  Name Age Salary
```

```
1 John  28  50000
2 Anna  35  60000
```

Example 4: Treating the First Row as Column Names

By default, header = TRUE, which means R treats the first row of the CSV file as column names.

```
data <- read.csv("path/to/your/file.csv", header = TRUE)
head(data)
```

This reads the file and assigns the values in the first row as column names.

Output:

```
  Name Age Salary
1 John  28  50000
2 Anna  35  60000
```

Example 5: Specifying Column Classes

You can explicitly define the data type for each column using the colClasses parameter. For example, the first column will be a character, the second a numeric value, and the third an integer.

```
data <- read.csv("path/to/your/file.csv", colClasses = c("character", "numeric", "integer"))
head(data)
```

Output:

```
  Name Age Salary
1 John  28  50000
2 Anna  35  60000
```

Example 6: Skipping Rows and Specifying Missing Values

The skip parameter allows you to skip a specific number of rows from the top of the file. Additionally, the na.strings parameter specifies which values should be treated as missing (e.g., "NA" or empty strings).

```
data <- read.csv("path/to/your/file.csv", skip = 3, na.strings = c("", "NA"))
head(data)
```

This example skips the first three rows of the file and treats empty strings and "NA" as missing values.

Output:

```
  Name Age Salary
1 John  28  50000
2 Anna  35  60000
```

The read.csv() function in R is versatile and easy to use for importing CSV files. It provides various options to customize the import process, such as handling column names, delimiters, missing values, and row skipping. Whether you're working with simple CSV files or complex ones with special formatting, understanding how to use read.csv() effectively can help streamline your data analysis in R.

CSV files are text files where each row's values are separated by a delimiter (comma, tab, etc.). R handles these files easily using **data frames**.

Working with Directories

```
# Get current working directory
print(getwd())
# Set new working directory
setwd("/web/com")
# Confirm the working directory
print(getwd())
```

Output Example:

```
[1] "C:/Users/GFG19565/Documents"
```

Reading a CSV File : CSV Sample (sample.csv):

```
id, name, department, salary, projects
1, A, IT, 60754, 4
2, B, Tech, 59640, 2
```

```
csv_data <- read.csv(file = 'C:\\Users\\GFG19565\\Downloads\\sample.csv')
print(csv_data)
# Number of columns and rows
print(ncol(csv_data))
print(nrow(csv_data))
```

Querying CSV Data

Minimum Projects:

```
min_pro <- min(csv_data$projects)
print(min_pro)
# Output: 2
```

Filter Employees (Salary > 60000):

```
result <- csv_data[csv_data$salary > 60000, c("name", "salary")]
print(result)
```

Output:

Name	salary
A	60754
C	69040
D	65043
F	65000
G	69000

Writing to a CSV File

Example: Average Salary by Department:

```
result <- tapply(csv_data$salary, csv_data$department, mean)
print(result)
```

Output:

Department	Avg Salary
HR	69000.0
IT	62877.0
Marketing	67041.5
Tech	59791.5

3.1.2 XML FILES

XML (eXtensible Markup Language) is a **widely used format** for storing and transporting data. R provides powerful tools to **read, parse, and extract data** from XML files using libraries like XML and xml2.

Step 1 Install and Load Required Packages

```
# Install the XML package if not already installed
install.packages("XML")
# Load the XML package
library(XML)
```

Alternatively, you can also use xml2 (more modern and tidy):

```
# Install and load xml2
install.packages("xml2")
library(xml2)
```

Sample XML File

Let's say you have an XML file named employees.xml with the following content:

```
<employees>
  <employee>
    <id>1</id>
    <name>Alice</name>
    <department>IT</department>
    <salary>60754</salary>
  </employee>
  <employee>
    <id>2</id>
    <name>Bob</name>
    <department>Tech</department>
    <salary>59640</salary>
  </employee>
</employees>
```

Reading XML Data in R

Using XML package

```
# Parse the XML file
xml_file <- xmlParse("employees.xml")
# Print the structure
print(xml_file)
# Extract the root node
root_node <- xmlRoot(xml_file)
# View the root tag name
print(xmlName(root_node))
# Get all employee nodes
employees <- xmlSApply(root_node, function(x) xmlSApply(x, xmlValue))
# Convert to a data frame
df <- data.frame(t(employees), row.names = NULL)
print(df)
```

Output:

Id	name	department	salary
1	Alice	IT	60754
2	Bob	Tech	59640

Using xml2 package

```
# Read the XML file
xml_file <- read_xml("employees.xml")
# Extract all employee nodes
employee_nodes <- xml_find_all(xml_file, ".//employee")
# Extract data into a list
employee_list <- lapply(employee_nodes, function(emp) {
  list(
    id = xml_text(xml_find_first(emp, "id")),
    name = xml_text(xml_find_first(emp, "name")),
    department = xml_text(xml_find_first(emp, "department")),
    salary = xml_text(xml_find_first(emp, "salary"))
  )
})

# Convert the list to a data frame
df <- do.call(rbind, lapply(employee_list, as.data.frame))
print(df)
```

Extract Specific Information

Example: Get names of all employees:

```
names <- xpathSApply(xml_file, "//employee/name", xmlValue)
print(names)
```

Or using xml2:

```
names <- xml_text(xml_find_all(xml_file, ".//employee/name"))
print(names)
```

Write Data to XML File

You can also **write data to an XML file** using saveXML():

```
# Example: Write the same XML structure back
saveXML(xml_file, file = "output.xml")
```

Extracting Information About the XML File

Before processing XML data, you may want to **inspect the structure** of the file.

```
library(xml2)
# Read the XML file
doc <- read_xml("path/to/your/file.xml")
# View the root node
xml_root(doc)
# List all nodes in the document
xml_find_all(doc, "//*")
# Extract names of all nodes
node_names <- xml_name(xml_find_all(doc, "//*"))
print(node_names)
# Extract attributes from nodes
attributes_list <- xml_attrs(xml_find_all(doc, "//*"))
print(attributes_list)
```

This allows you to **understand the hierarchy and structure** of the XML document.

Conversion of XML to DataFrame

A common task is to **convert XML data into a structured table (data frame)** for further analysis.

Example: Parsing a books XML:

```
library(xml2)
library(dplyr)
# Read the XML file
doc <- read_xml("books.xml")
# Find all <book> nodes
books <- xml_find_all(doc, "//book")
# Convert to data frame
df <- data.frame(
  id = xml_attr(books, "id"),
  title = xml_text(xml_find_all(books, "title")),
  author = xml_text(xml_find_all(books, "author")),
  price = xml_text(xml_find_all(books, "price")),
  stringsAsFactors = FALSE
)
print(df)
```

This gives a **tidy table** where each row corresponds to a book and each column contains its details.

R XML Basics – How to Read and Parse XML Files

The basic **workflow for reading and parsing XML files** is:

```
library(xml2)
# Read XML file
doc <- read_xml("example.xml")
# Access the root node
root_node <- xml_root(doc)
# Access child nodes
children <- xml_children(root_node)
# Loop through each child node to extract content
for (child in children) {
  cat("Node name:", xml_name(child), "\n")
  cat("Text content:", xml_text(child), "\n")
}
```

You can also use **XPath queries** for precise extraction:

```
items <- xml_find_all(doc, "//item")
for (item in items) {
  print(xml_text(item))
}
```

How to Convert XML Data to tibble and data.frame

A **tibble** is a modern version of the data frame that's part of the tidyverse.

```
library(xml2)
library(tibble)
# Read the XML
doc <- read_xml("books.xml")
# Find <book> nodes
books <- xml_find_all(doc, "//book")
# Create a tibble
book_tbl <- tibble(
  id = xml_attr(books, "id"),
```

```
  title = xml_text(xml_find_all(books, "title")),
  author = xml_text(xml_find_all(books, "author")),
  price = as.numeric(xml_text(xml_find_all(books, "price")))
)
print(book_tbl)
```

Advantages of tibble:

- Better printing
- Never converts strings to factors
- Part of the tidyverse, making it easy to integrate with other tools

31.3 – WEB DATA

Many **websites provide data** for users in various formats such as **CSV, TXT, and XML.** For example, the **World Health Organization (WHO)** shares global health data that can be programmatically accessed. In R, we can **automate the extraction of specific data** from these sites using **web scraping and download tools.**

Popular Packages for Web Scraping

Package	Purpose
RCurl	Connect to web URLs and fetch content.
XML	Parse and extract data from HTML/XML content.
stringr	Handle and manipulate strings (e.g., to filter links).
plyr	Apply functions over lists/vectors (used here to download multiple files).

Installation of Required Packages

If these packages are **not already installed**, use the following commands:

```
install.packages("RCurl")
install.packages("XML")
install.packages("stringr")
install.packages("plyr")
```

Example: Downloading Weather Data (CSV Files)

We will extract links from a weather data webpage and download the CSV files for the year 2015.

URL: http://www.geos.ed.ac.uk/~weather/jcmb_ws/

Step 1: Read the URL and Gather Links

```
library(RCurl)
library(XML)
library(stringr)
library(plyr)

# URL of the website
url <- "http://www.geos.ed.ac.uk/~weather/jcmb_ws/"
# Get all the links from the webpage
links <- getHTMLLinks(url)
```

Step 2: Filter Required Links (2015 CSV Files)

We want only the links that point to the files from 2015**:**

```
# Filter links that contain "JCMB_2015"
filenames <- links[str_detect(links, "JCMB_2015")]
# Convert to a list
filenames_list <- as.list(filenames)
print(filenames_list)
```

Step 3: Function to Download Files

We will create a **function** that:

- **Takes the main URL + file name**
- Downloads the file to the **local system**

```
downloadcsv <- function (mainurl, filename) {
  filedetails <- str_c(mainurl, filename)
```

```
    download.file(filedetails, filename)
}
```

Step 4: Download All Files

Use l_ply() from the **plyr package** to **download multiple files automatically:**

```
l_ply(filenames, downloadcsv, mainurl =
"http://www.geos.ed.ac.uk/~weather/jcmb_ws/")
```

Verify the File Download

After running the code, you should see these files in your **current R working directory:**

- JCMB_2015.csv
- JCMB_2015_Jan.csv
- JCMB_2015_Feb.csv
- JCMB_2015_Mar.csv
- JCMB_2015_Apr.csv
- ... and other 2015 files.

Web Scraping Using R

- **Web scraping** is a technique that allows us to automatically extract information from websites, especially when the data we need isn't available through downloadable datasets or public APIs (Application Programming Interfaces). Instead of manually copying and pasting content, web scraping uses code to fetch and parse the structure of a web page.

Install the Required Package

Install the rvest package in RStudio using the following command:

```
install.packages('rvest')
```

SelectorGadget: A Helpful Tool

To simplify web scraping, we can use an open-source browser extension called **SelectorGadget**. It helps identify CSS selectors for extracting specific elements from a webpage. Once installed (preferably on Google Chrome), SelectorGadget will appear in the browser's extension bar at the top right.

1 Import rvest Library

We begin by importing the rvest library:

```
library(rvest)
```

2. Read the Webpage

We read the HTML code from the webpage using read_html(). For this example, we'll scrape the following URL:

```
webpage = read_html("https://www.geeksforgeeks.org/data-structures-in-r-programming")
```

3.Scrape Data From the Webpage ,

Scraping the Heading Section Using SelectorGadget, we identify the CSS selector for the heading section and scrape it:

```
# Using CSS selectors to scrape the heading
 heading = html_node(webpage, '.entry-title')
 text = html_text(heading)
 print(text)
```

Output:

```
[1] "Data Structures in R Programming"
```

4. Scraping All Paragraphs

We scrape all paragraph (<p>) elements from the page:

```
# Using CSS selectors to scrape all paragraphs
paragraph = html_nodes(webpage, 'p')
pText = html_text(paragraph)
print(head(pText))
```

Output:

```
[1] "A data structure is a particular way of organizing data in a computer so that it can be used effectively. The idea is to reduce the space and time complexities of different tasks. Data structures in R programming are tools for holding multiple values."
[2] "R’s base data structures are often organized by their dimensionality (1D, 2D, or nD) and whether they’re homogeneous (all elements must be of the identical type) or heterogeneous (the elements are often of various types). This gives rise to the five data types which are most frequently utilized in data analysis."
[3] "The most essential data structures used in R include:"
```

3.1.4 – JSON FILES

JSON (JavaScript Object Notation) is a lightweight format for data exchange, human-readable, and easy to write and parse. R can **read from and write to JSON files** using the rjson package. Working with JSON files in R using the rjson package,

- Installing & loading the package
- Creating a JSON file
- Reading JSON data
- Writing data to JSON
- Converting JSON to a data frame
- Working with URLs & online JSON data

1 Install and Load the rjson Package

```
install.packages("rjson")      # Install the package
library("rjson")               # Load the package
```

2 Create a JSON File

Copy the following data into a text editor and save it as example.json (choose **"All Files"** when saving):

```
{
  "ID":["1","2","3","4","5"],
  "Name":["Mithuna","Tanushree","Parnasha","Arjun","Pankaj"],
  "Salary":["722.5","815.2","1611","2829","843.25"],
  "StartDate":["6/17/2014","1/1/2012","11/15/2014","9/23/2013","5/21/2013"],
  "Dept":["IT","IT","HR","Operations","Finance"]
}
```

3 Reading a JSON File

Example:

```
library("rjson")
# Read the JSON file
result <- fromJSON(file = "E:\\example.json")
# Print the result
print(result)
```

Output:

```
$ID
[1] "1" "2" "3" "4" "5"
$Name
[1] "Mithuna"   "Tanushree" "Parnasha"  "Arjun"     "Pankaj"
$Salary
[1] "722.5"  "815.2"  "1611"   "2829"   "843.25"
$StartDate
[1] "6/17/2014"  "1/1/2012"   "11/15/2014" "9/23/2013"  "5/21/2013"
```

```
$Dept
[1] "IT"         "IT"         "HR"         "Operations" "Finance"
```

4 Writing into a JSON File

Example:

```
library("rjson")
# Create a list
list1 <- vector(mode="list", length=2)
list1[[1]] <- c("sunflower", "guava", "hibiscus")
list1[[2]] <- c("flower", "fruit", "flower")
# Convert to JSON
jsonData <- toJSON(list1)
# Write JSON to a file
write(jsonData, "result.json")
# Read back the file
result <- fromJSON(file = "result.json")
print(result)
```

Output:

```
[[1]]
[1] "sunflower" "guava"     "hibiscus"
[[2]]
[1] "flower" "fruit"  "flower"
```

5 Converting JSON to Data Frame

Example:

```
library("rjson")
# Read JSON file
result <- fromJSON(file = "E://example.json")
# Convert to data frame
json_data_frame <- as.data.frame(result)
print(json_data_frame)
```

Output:

ID	Name	Salary	StartDate	Dept
1	Mithuna	722.5	6/17/2014	IT
2	Tanushree	815.2	1/1/2012	IT
3	Parnasha	1611	11/15/2014	HR
4	Arjun	2829	9/23/2013	Operations
5	Pankaj	843.25	5/21/2013	Finance

6 Working with URLs (Fetching JSON Data from the Web)

Example:

```
library(RJSONIO)
# Extract data from URL
Raw <- fromJSON("https://data.ny.gov/api/views/9a8c-
vfzj/rows.json?accessType=DOWNLOAD")
# Extract the 'data' node
food_market <- Raw[['data']]
# Extract store names
Names <- sapply(food_market, function(x) x[[14]])
# Preview the names
head(Names)
```

Output:

```
[1] "LUCKY MART"            "CUMBERLAND FARMS 1587"
[3] "K&M SPORTS"            "MASON&OLD RIDGE FARM"
[5] "HAMPTON CHUTNEY CO"    "CM - HUTCHINSON"
```

Example of JSON content:

```
{
  "name": "John",
  "age": 30,
  "city": "New York"
}
```

In R, we use the **jsonlite** package to work with JSON files.

2. Load the Library

First, make sure to load the jsonlite package:

```
library(jsonlite)
```

This package gives access to functions like read_json() and fromJSON().

3. Reading JSON Directly from a URL

Example:

```
json_data <- read_json(
  "https://raw.githubusercontent.com/datacarpentry/r-socialsci/main/episodes/data/SAFI.json"
)
```

- It reads the JSON file **directly** from the internet.
- You'll get a **list** object

4. Downloading the JSON File

If you want to **save** the file locally:

```
download.file(
  "https://raw.githubusercontent.com/datacarpentry/r-socialsci/main/episodes/data/SAFI.json",
  "data/SAFI.json", mode = "wb"
)
```

Check: This saves the JSON file as "data/SAFI.json" in your working directory.

5. Reading a Local JSON File

Now read the **downloaded JSON file**:

```
json_data <- read_json("data/SAFI.json")
```

By default, this **keeps the JSON structure as a list**.

6. **Explore the Data**

Example:

```
head(json_data)
```

You may find that the data looks **nested or messy**, with many lists inside lists.

7. Simplify into a Data Frame

To **simplify** and turn the JSON into a **data frame** (table format):

```
json_data <- read_json("data/SAFI.json", simplifyVector = TRUE)
```

Now it becomes a **data frame** with rows and columns.

8. Coerce to Tibble and Glimpse

If you're using tidyverse, you can **convert it to a tibble** and use glimpse():

```
library(tidyverse)
json_data <- json_data %>% as_tibble()
glimpse(json_data)
```

Example output:

```
Rows: 131
Columns: 74
$ C06_rooms      <int> 1, 1, 1, 1, 1, 1, 1, 3, 1, 5, ...
$ A08_ward       <chr> "ward2", "ward2", "ward2", ...
$ E01_water_use  <chr> "no", "yes", "no", ...
```

9. Accessing Columns and Values

Once it’s in data frame/tibble format, you can work with it **like any other data frame**.

Example: Get all the room numbers:

```
json_data$C06_rooms
```

Example: Filter rows where E01_water_use == "yes":

```
json_data %>% filter(E01_water_use == "yes")
```

3.1.5 – DATABASES

A **database** is an organized collection of data that allows fast access, management, and updating. Common databases:

- **SQLite** (lightweight, file-based)
- **MySQL / MariaDB**
- **PostgreSQL**
- **SQL Server**
- **Oracle**

Databases typically use **SQL (Structured Query Language)** to manage and query data.

Working with Databases in R

R provides **packages** that let you **connect to databases** and work with data just like with data frames.

Popular R Packages:

Package	Purpose
DBI	General interface for databases
RSQLite	Work with SQLite databases
RMariaDB	Work with MySQL/MariaDB
RPostgres	Work with PostgreSQL
Odbc	Connect to various databases via ODBC

Pool	Manage connections (good for shiny apps)

Example 1: **Using SQLite (No Server Needed!)**

1 Install and Load Libraries

```
install.packages(c("DBI", "RSQLite"))
library(DBI)
library(RSQLite)
```

2 Connect to a SQLite Database

```
# Create a new SQLite database (or connect to existing one)
con <- dbConnect(RSQLite::SQLite(), "my_database.sqlite")
```

3 Create a Table

Example: Create a table of people.

```
dbExecute(con, "
  CREATE TABLE people (
    id INTEGER PRIMARY KEY,
    name TEXT,
    age INTEGER
  )
")
```

4 Insert Data

```
dbExecute(con, "INSERT INTO people (name, age) VALUES ('Alice', 25)")
dbExecute(con, "INSERT INTO people (name, age) VALUES ('Bob', 30)")
```

OR insert data using a data frame:

```
people_df <- data.frame(name = c("Charlie", "David"), age = c(35, 40))
dbWriteTable(con, "people", people_df, append = TRUE)
```

5 List Tables

```
dbListTables(con)
# Output: [1] "people"
```

6 Query Data

```
result <- dbGetQuery(con, "SELECT * FROM people")
print(result)
```

Example output:

```
  id    name age
1  1   Alice  25
2  2     Bob  30
3  3 Charlie  35
4  4   David  40
```

7. Disconnect

```
dbDisconnect(con)
```

Example 2: Connect to MySQL (Replace with your own credentials)

```
library(DBI)
library(RMariaDB)

con <- dbConnect(
  RMariaDB::MariaDB(),
  user = "your_username",
  password = "your_password",
  dbname = "your_database",
  host = "localhost"
)
# List tables
dbListTables(con)
# Query
result <- dbGetQuery(con, "SELECT * FROM your_table LIMIT 5")
print(result)
# Disconnect
dbDisconnect(con)
```

Example 3: Connect to PostgreSQL

```
library(DBI)
library(RPostgres)
con <- dbConnect(
  RPostgres::Postgres(),
```

```
  dbname = "your_db",
  host = "localhost",
  port = 5432,
  user = "your_user",
  password = "your_password"
)
dbListTables(con)
dbDisconnect(con)
```

Why Use Databases?

- Store large datasets **efficiently**
- Perform **fast queries** (filter, join, aggregate)
- **Share data** across different apps/users
- Keep data **safe and structured**

Key Functions (DBI package)

Function	Description
dbConnect()	Connect to the database
dbDisconnect()	Close the connection
dbListTables()	List all tables
dbListFields()	List columns in a table
dbGetQuery()	Run a SQL query and fetch results
dbExecute()	Run a SQL command (no return)
dbWriteTable()	Write a data frame to a table
dbReadTable()	Read a whole table into R
dbRemoveTable()	Delete a table

EXAMPLE

```
# Install libraries
install.packages("DBI")
install.packages("RSQLite")
# Load libraries
library(DBI)
```

```
library(RSQLite)
# Connect to (or create) SQLite database
con <- dbConnect(RSQLite::SQLite(), "my_first_db.sqlite")
# Create a table
dbExecute(con, "
  CREATE TABLE students (
    id INTEGER PRIMARY KEY,
    name TEXT,
    grade REAL
  )
")
# Insert data
dbExecute(con, "INSERT INTO students (name, grade) VALUES ('Alice', 85.5)")
dbExecute(con, "INSERT INTO students (name, grade) VALUES ('Bob', 90.0)")
dbExecute(con, "INSERT INTO students (name, grade) VALUES ('Charlie',
78.0)")
# Query data
result <- dbGetQuery(con, "SELECT * FROM students")
# Print the result
print(result)
# Disconnect
dbDisconnect(con)
```

OUTPUT

```
  id    name grade
1  1   Alice  85.5
2  2     Bob  90.0
3  3 Charlie  78.0
```

3.1.6 – EXCEL FILES

Excel files (.xls, .xlsx) are spreadsheet documents that organize data into:

- **Rows & Columns**: Perfect for tabular data.
- **Sheets**: Each workbook can have many sheets (tabs).

Why use Excel with R?

- **Data Entry & Sharing:** Excel is **widely used** by non-programmers.
- **Data Analysis:** We often receive **real-world data** in Excel format.

- **Export Results:** We might **analyze data in R** and export results back to Excel for reporting.

How to Work with Excel in R?

The most popular R packages:

- readxl → **Read Excel files**
- openxlsx or writexl → **Write Excel files**

1 Reading and Importing Excel Files into R with readxl

- The readxl package is designed to **import Excel files (.xls, .xlsx)** into R **without needing Excel installed.**
- It handles **modern Excel files efficiently.**

Install & Load:

```
install.packages("readxl")
library(readxl)
```

2 Reading the First Workbook (Default Sheet)

- If you don't specify the sheet, read_excel() will **read the first sheet by default.**

Example:

```
data <- read_excel("students.xlsx")
print(data)
```

Output:

```
# A tibble: 3 × 3
     id name    grade
  <dbl> <chr>   <dbl>
1     1 Alice    85.5
```

```
2  2 Bob      90.0
3  3 Charlie  78.0
```

3 Reading Specific Rows

- Use range = to **specify rows & columns to read.**
- **Excel range notation:** "A2:C4" → reads from cell A2 to C4.

Example: Read only rows 2-4:

```
data_rows <- read_excel("students.xlsx", range = "A2:C4")
print(data_rows)
```

4 Reading Specific Cells

- To **read specific cells** (e.g., a single cell or a block), also use the range argument.

Example: Read just cell B2:

```
cell_value <- read_excel("students.xlsx", range = "B2")
print(cell_value)
```

This will return **just one cell** value.

5 Reading Data with No Header Row

- If your Excel file **doesn't have headers**, use:

```
Col_names = FALSE
```

- R will name columns as X__1, X__2, ...

Example:

```
data_no_header <- read_excel("students.xlsx", col_names = FALSE)
print(data_no_header)
```

6 Specifying Column Types

- By default, readxl guesses column types, but you can **force column types** using col_types.
- Common types: "text", "numeric", "date".

Example: Set id = text, name = text, grade = numeric

```
data_types <- read_excel(
  "students.xlsx",
  col_types = c("text", "text", "numeric")
)
print(data_types)
```

7 Deleting Content from Files

Note: readxl is a **read-only package**. To delete or modify Excel content, you need openxlsx.

Example: Clear sheet (overwrite it):

```
library(openxlsx)
# Create a blank workbook
wb <- loadWorkbook("students.xlsx")
# Overwrite Sheet1 with empty data
writeData(wb, sheet = 1, x = data.frame())
# Save
saveWorkbook(wb, "students_cleared.xlsx", overwrite = TRUE)
```

8 Modifying Files (Editing Content)

- Use openxlsx to **edit Excel files directly.**

Example: Add a new row

```
library(openxlsx)
# Load workbook & data
```

```
wb <- loadWorkbook("students.xlsx")
data <- read.xlsx("students.xlsx")
# Add a new student
data <- rbind(data, data.frame(id = 4, name = "David", grade = 88))
# Overwrite sheet
writeData(wb, sheet = 1, x = data)
# Save changes
saveWorkbook(wb, "students_modified.xlsx", overwrite = TRUE)
```

9 Merging Files (Combining Excel Files)

- To **combine multiple Excel files** into one data frame, read them separately and use rbind().

Example: Merge students1.xlsx and students2.xlsx

```
df1 <- read_excel("students1.xlsx")
df2 <- read_excel("students2.xlsx")
# Combine them
merged_data <- rbind(df1, df2)
print(merged_data)
```

Creating New Columns

- In R, you can **add a new column** by assignment:

```
data$new_column <- value
```

Example: Add a "pass/fail" column

```
students_data <- read_excel("students.xlsx")
# Add new column based on grade
students_data$status <- ifelse(students_data$grade >= 50, "Pass", "Fail")
print(students_data)
```

Output:

```
# A tibble: 3 × 4
     id name    grade status
  <dbl> <chr>   <dbl> <chr>
1     1 Alice    85.5  Pass
2     2 Bob      90.0  Pass
3     3 Charlie  78.0  Pass
```

3.1.7 – WORKING WITH R CHARTS AND GRAPHS

A **chart or graph** is a **visual representation** of data that helps to identify trends, patterns, and relationships easily.

R provides two main ways of plotting:

1. **Base R plotting system** (basic built-in functions)
2. **ggplot2** (advanced plotting system)

TABLE: BASIC CHART TYPES IN R

Chart Type	Function in R	Definition
Scatter Plot	plot()	Shows **dots** that represent data points for two numeric variables (X and Y).
Line Chart	plot(..., type = "l")	A **line connects** data points to show trends over time or sequences.
Bar Chart	barplot()	Displays data as **rectangular bars** proportional to the values.
Histogram	hist()	Shows the **distribution of continuous data** using bins (ranges).
Box Plot	boxplot()	Displays **data summary** (median, quartiles, outliers) in a box shape.
Pie Chart	pie()	Represents data in **circular sections** showing proportions.
Density Plot	plot(density(...))	Shows the **distribution shape** (smoothed curve) of numeric data.

Heatmap	heatmap()	A **color-coded matrix** showing relationships between two categorical variables.
Pair Plot	pairs()	Matrix of scatter plots for **multiple variables** to see pairwise relationships.
ggplot2 Graphs	ggplot() + layers	Advanced grammar of graphics to build **complex plots** in steps.

Example of Table Usage

Parameter	Meaning
Main	Sets the **title of the graph**.
xlab/ylab	Labels for the **X and Y axes.**
Col	Sets the **color** of lines, bars, or points.
Pch	Point shape (e.g., circle, triangle).
Type	"p" = points, "l" = lines, "b" = both.
breaks	Sets the **number of bins** in a histogram.
names.arg	Sets the **labels** for bars in a bar chart.

Quick Basic Examples:

Scatter Plot

```
plot(cars$speed, cars$dist, main = "Speed vs Distance", xlab = "Speed", ylab = "Distance", col = "blue", pch = 19)
```

Bar Chart

```
barplot(c(10, 20, 30), names.arg = c("A", "B", "C"), main = "Bar Chart Example", col = "green")
```

Histogram

```
hist(iris$Sepal.Length, main = "Histogram of Sepal Length", xlab = "Sepal Length", col = "purple")
```

Pie Chart

```
slices <- c(40, 30, 30)
labels <- c("Apple", "Banana", "Cherry")
```

```
pie(slices, labels = labels, main = "Fruit Pie Chart", col = c("red", "yellow", "pink")
```

3.1.8 – HISTOGRAMS

A **Histogram** is a **graphical representation** that organizes a group of **continuous numerical data** into **ranges (called bins)** and shows the **frequency (count)** of data points in each range.

Purpose:

- To **visualize the distribution** (shape, spread) of a dataset.
- Helps to understand if data is **normal, skewed, bimodal, etc**

Base R Function:

hist(x, ...)

- x is your **numeric vector** (data you want to plot).

Basic Syntax in R:

```
hist(x,
    breaks = "Sturges",
    main = NULL,
    xlab = NULL,
    xlim = NULL,
    ylim = NULL,
    col = NULL,
    border = NULL)
```

Main Parameters:

Parameter	Description
X	The **numeric data vector**.
breaks	Defines the **number of bins (intervals)** or gives exact breakpoints.
col	Sets the **color** of the bars.
main	The **main title** of the histogram.
xlab	**Label for the X-axis.**

ylab	**Label for the Y-axis.**
border	Color of the **border** of each bin/bar.
freq	TRUE: shows **frequency count** (default); FALSE: shows **density.**
labels	Logical: whether to **label each bar** with its count/density.
xlim, ylim	Set the **limits of X and Y axes.**
right	Logical: TRUE = include right endpoint of the bin (default).
density	Adds **shading lines** inside bars (if not solid color).
angle	Sets the **angle of shading lines** (if using density).

Example Program in R:

```
# Basic histogram of mtcars$mpg
data <- mtcars$mpg

hist(data,
    breaks = 10,          # Number of bins
    main = "Histogram of Miles Per Gallon",
    xlab = "MPG",
    col = "skyblue",
    border = "black")
```

1 **Change Number of Bins:**

```
hist(data, breaks = 5)
```

2 **Add Labels & Colors:**

```
hist(data, main = "MPG Distribution", xlab = "MPG", col = "lightgreen", border =
"darkgreen")
```

3 **Density Histogram:**

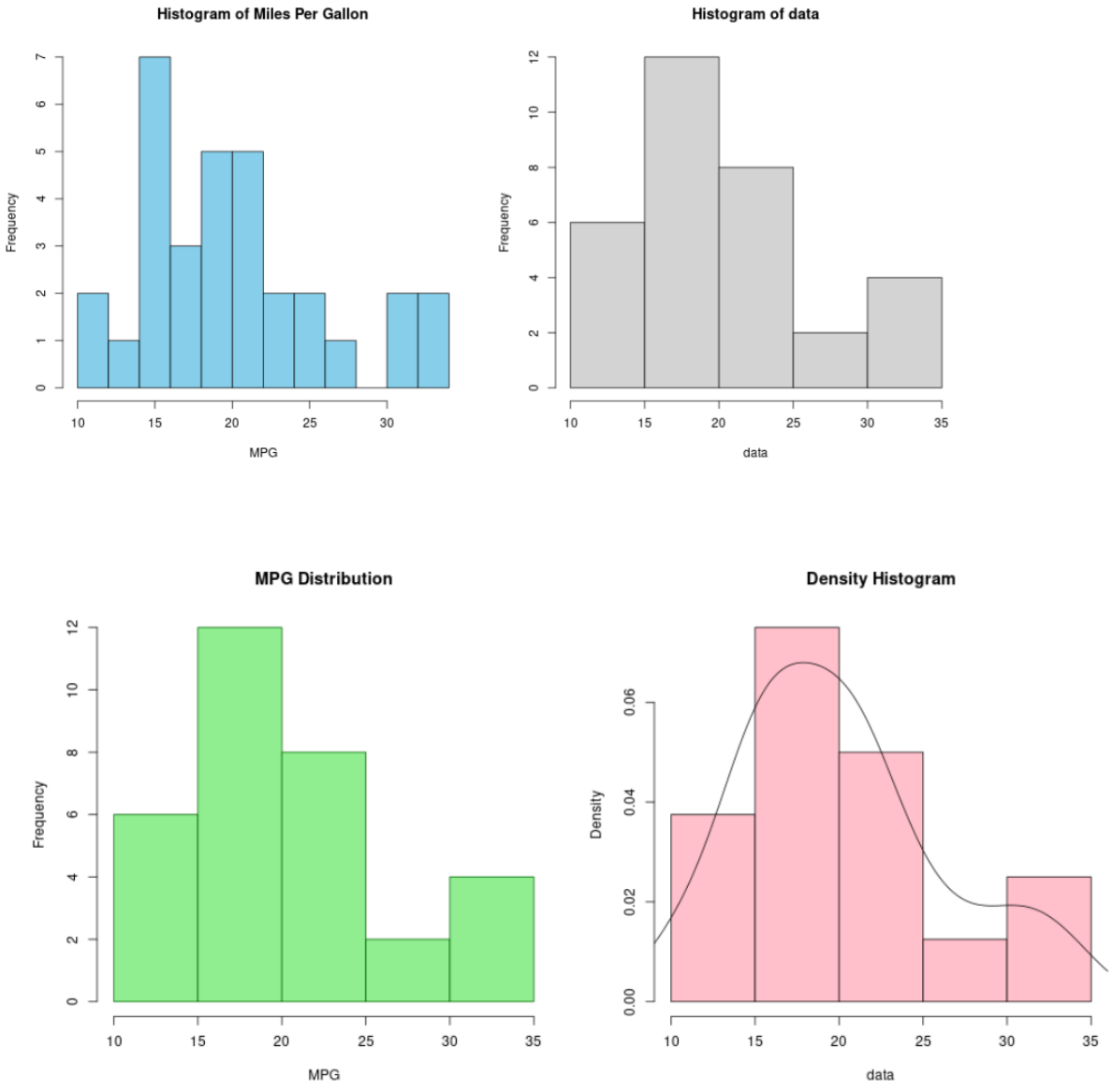
```
hist(data, freq = FALSE, main = "Density Histogram", col = "pink")
lines(density(data))
```

- **Bins:** Control how detailed the histogram is.
- **Color:** Makes the plot easier to read.
- **Density:** Shows probability density instead of counts.
- **Axis Labels:** Help explain the data shown.

Example 1: Basic Histogram

```
data <- iris$Sepal.Length
hist(data, main = "Histogram of Sepal Length", xlab = "Sepal Length", col = "skyblue")
```

Output



Example 2: Control Number of Bins

```
hist(data, breaks = 5, main = "Histogram with 5 bins", xlab = "Sepal Length", col = "lightgreen")
```

- breaks = 5 → splits data into 5 ranges.

Example 3: Density Instead of Frequency

```
hist(data, freq = FALSE, main = "Density Histogram", xlab = "Sepal Length", col = "orange")
lines(density(data), col = "blue", lwd = 2)
```

Adds a density curve over the histogram.

Example 4: Add Labels to Bars

```
hist(data, col = "purple", labels = TRUE, main = "Histogram with Labels", xlab = "Sepal Length")
```

Each bar is labeled with its count.

Example 5: Custom X & Y Limits

```
hist(data, col = "pink", xlim = c(4,8), ylim = c(0,20), main = "Custom Limits", xlab = "Sepal Length")
```

Advanced Example: Set Exact Bin Ranges

```
hist(data, breaks = seq(4,8,by=0.5), col = "gold", main = "Custom Bin Width (0.5)", xlab = "Sepal Length")
```

- breaks = seq(4,8,by=0.5) → makes bins from 4 to 8 with width of 0.5.

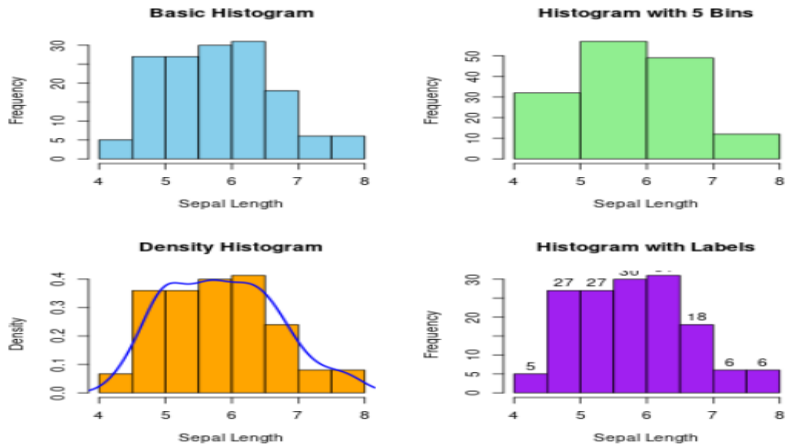
Extra Parameters:

Parameter	Use Example	What It Does
border	border = "red"	Sets border color of bars.
density	density = 20	Adds shading lines to bars.
angle	angle = 45	Controls shading line angle.
probability	probability = TRUE	Same as freq = FALSE (for density).

R HISTOGRAM EXAMPLES

```
# 1 Load Data
data <- iris$Sepal.Length
# 2 Set up 2x2 plotting area to show multiple histograms at once
par(mfrow = c(2, 2))
#  ◇  (1) Basic Histogram
hist(data,
     main = "Basic Histogram",
     xlab = "Sepal Length",
     col = "skyblue")
#  ◇  (2) Histogram with 5 Bins
hist(data,
     breaks = 5,
     main = "Histogram with 5 Bins",
     xlab = "Sepal Length",
     col = "lightgreen")
#  ◇  (3) Density Histogram + Curve
hist(data,
     freq = FALSE,
     main = "Density Histogram",
     xlab = "Sepal Length",
     col = "orange")
lines(density(data),
      col = "blue",
      lwd = 2)
#  ◇  (4) Histogram with Labels
hist(data,
     col = "purple",
     labels = TRUE,
     main = "Histogram with Labels",
     xlab = "Sepal Length")
#  ➕ NEW PAGE: Next set of histograms
dev.new()  # Open a new plot window
par(mfrow = c(2, 2))

#  ◇  (5) Custom Axis Limits
hist(data,
     col = "pink",
     xlim = c(4, 8),
```

```
    ylim = c(0, 20),
    main = "Custom Axis Limits",
    xlab = "Sepal Length")
# ◇ (6) Custom Bin Width (0.5)
hist(data,
    breaks = seq(4, 8, by = 0.5),
    col = "gold",
    main = "Custom Bin Width (0.5)",
    xlab = "Sepal Length")
# ◇ (7) Pattern Shading + Border
hist(data,
    col = "lightblue",
    border = "red",
    density = 20,
    angle = 45,
    main = "Pattern Shading",
    xlab = "Sepal Length")
# ◇ (8) mpg Histogram from mtcars
hist(mtcars$mpg,
    breaks = 8,
    col = "cyan",
    main = "MPG Histogram - mtcars",
    xlab = "Miles Per Gallon",
    labels = TRUE)
# Reset plotting layout to default (single plot)
par(mfrow = c(1, 1))
```



3.1.9 – BOXPLOTS

A **boxplot** in R, also known as a **box-and-whisker plot**, is a graphical representation of the distribution of a numerical variable. It displays key summary statistics that help in understanding the **central tendency, spread, and presence of outliers**.

Component	Description
Median (Q2)	The middle value of the dataset. Displayed as a horizontal line inside the box.
Box (IQR)	The box shows the interquartile range (IQR = Q3 - Q1), which contains the middle 50% of the data.
Whiskers	Lines extending from the box to the smallest and largest data points within 1.5 * IQR from Q1 and Q3.
Outliers	Data points outside the whiskers, often plotted as individual dots or stars.
IQR	Interquartile Range = Q3 - Q1; a robust measure of variability.

Default Rule for Whiskers and Outliers:

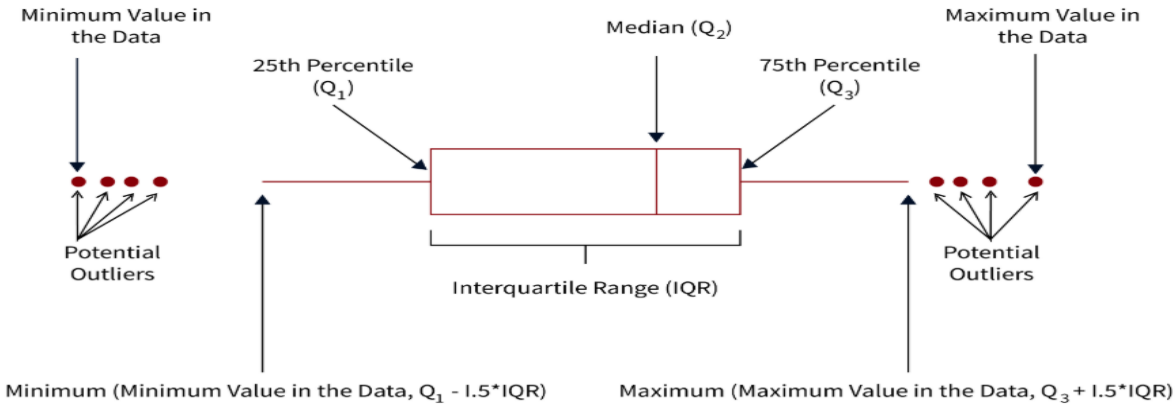
- **Upper whisker:** up to Q3 + 1.5 * IQR
- **Lower whisker:** down to Q1 - 1.5 * IQR
- Any point beyond these is treated as an **outlier**.

Basic Syntax of a Boxplot in R:

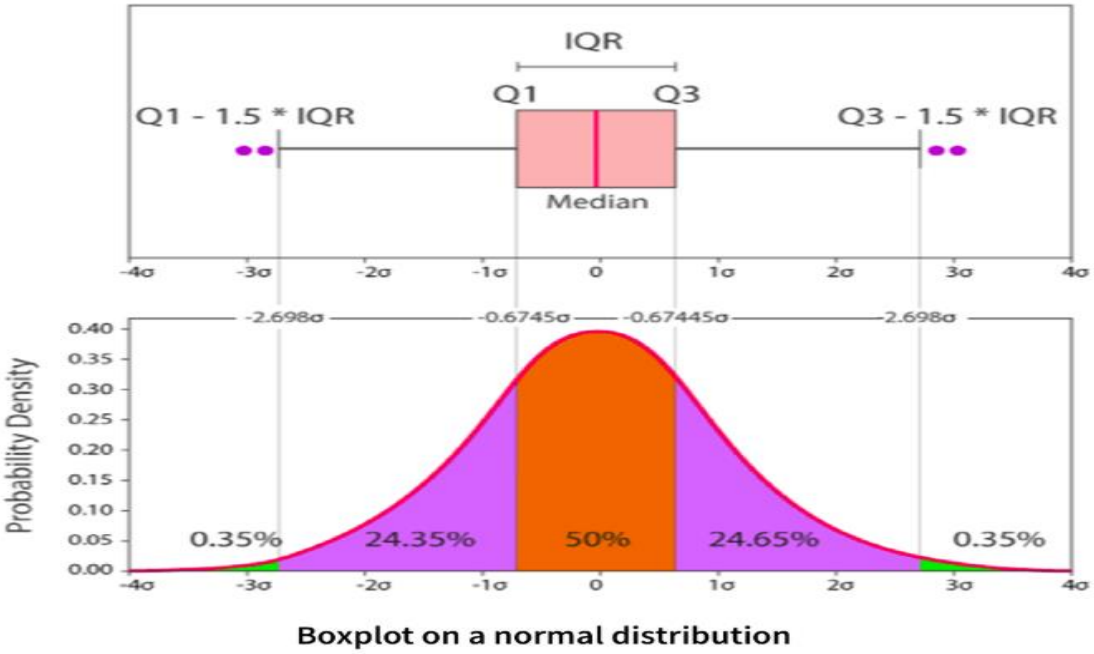
```
boxplot(x, data = NULL, main = "Title", xlab = "X-axis", ylab = "Y-axis", col = "color")
```

Parameter	Description
X	A numeric vector or formula.
Data	Optional dataset.
Main	Title of the boxplot.

xlab/ylab	Labels for axes.
Col	Color of the box.



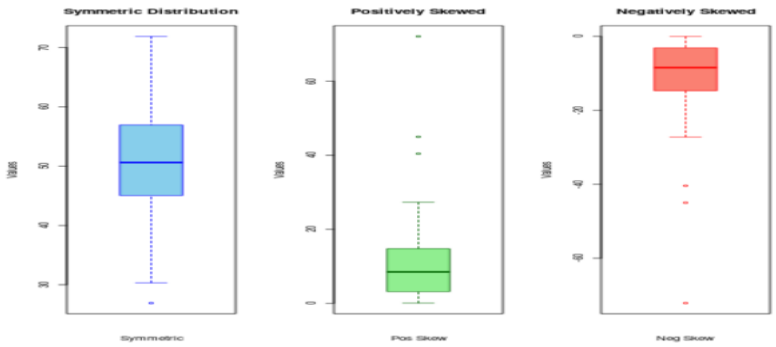
Type of Distribution	Description	Visual Clue in Boxplot
Positively Skewed	The right (upper) whisker is longer than the left (lower) whisker. The distance from the median to the maximum is greater than the distance from the median to the minimum.	The box leans **toward the lower side**; whisker **longer on the top side.**
Negatively Skewed	The left (lower) whisker is longer than the right (upper) whisker. The distance from the median to the minimum is greater than the distance from the median to the maximum.	The box leans **toward the upper side**; whisker **longer on the bottom side.**
Symmetric Distribution	The median is centered in the box and whiskers are of approximately equal length on both sides.	The box and whiskers are **balanced and even on both sides.**



Boxplot on a normal distribution

```
# Set up 1 row and 3 columns to show 3 plots together
par(mfrow = c(1, 3))
# 1 Symmetric distribution
set.seed(123)  # For reproducibility
symmetric_data <- rnorm(100, mean = 50, sd = 10)
boxplot(symmetric_data,
        main = "Symmetric Distribution",
        col = "skyblue",
        border = "blue",
        ylab = "Values",
        xlab = "Symmetric")
# 2 Positively Skewed distribution
set.seed(123)
positive_skew_data <- rexp(100, rate = 0.1)
boxplot(positive_skew_data,
        main = "Positively Skewed",
        col = "lightgreen",
        border = "darkgreen",
        ylab = "Values",
        xlab = "Pos Skew")
# 3 Negatively Skewed distribution
set.seed(123)
negative_skew_data <- -rexp(100, rate = 0.1)
boxplot(negative_skew_data,
```

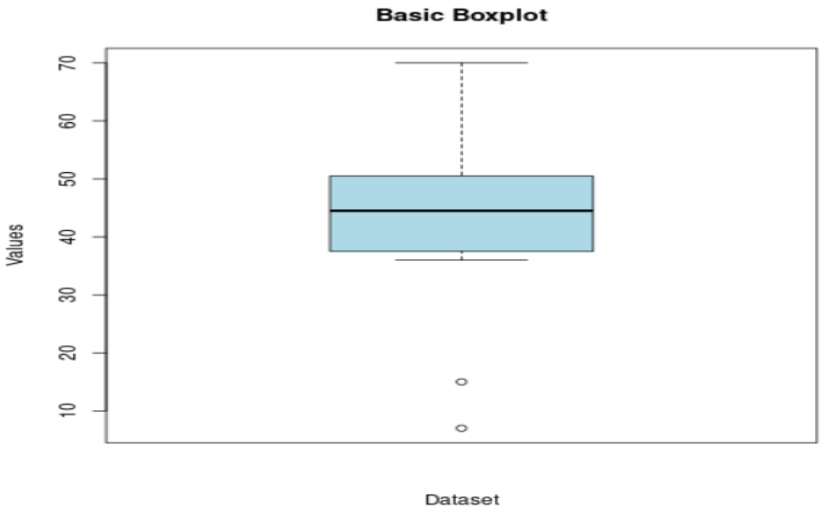
```
main = "Negatively Skewed",
col = "salmon",
border = "red",
ylab = "Values",
xlab = "Neg Skew")
```



Example Programs:

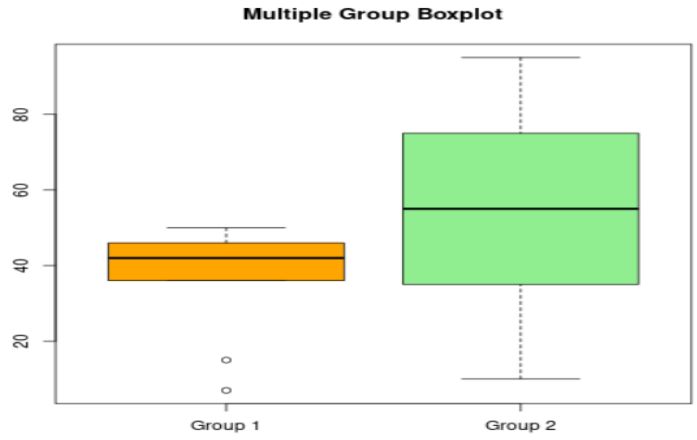
1 Basic Boxplot:

```
# Basic Boxplot
data <- c(7, 15, 36, 39, 42, 43, 46, 49, 50, 51, 53, 70)
boxplot(data, main = "Basic Boxplot", xlab = "Dataset", ylab = "Values", col =
"lightblue")
```



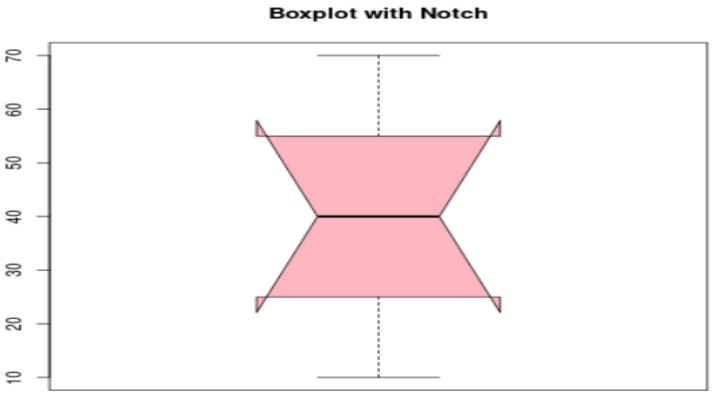
2 Boxplot for Multiple Groups:

```
# Boxplot with multiple groups
group1 <- c(7, 15, 36, 39, 42, 43, 46, 49, 50)
group2 <- c(10, 25, 35, 45, 55, 65, 75, 85, 95)
boxplot(group1, group2, names = c("Group 1", "Group 2"), main = "Multiple
Group Boxplot", col = c("orange", "lightgreen"))
```



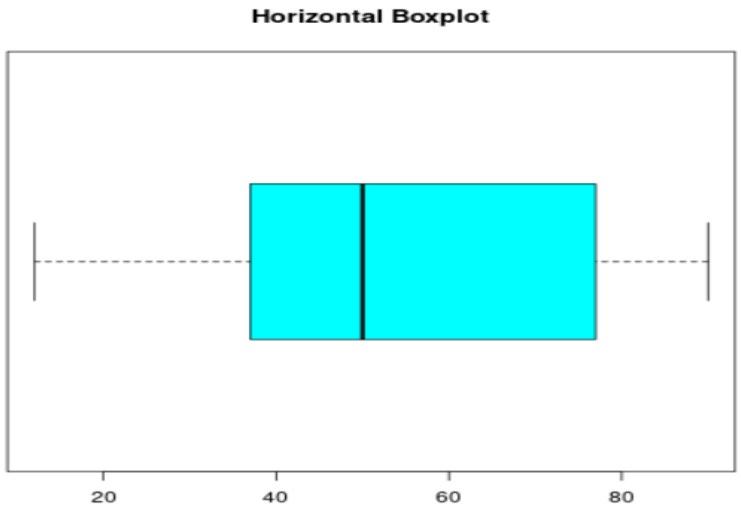
3 Boxplot with Notches:

```
# Boxplot with notch
data <- c(10, 20, 30, 40, 50, 60, 70)
boxplot(data, notch = TRUE, main = "Boxplot with Notch", col = "lightpink")
```



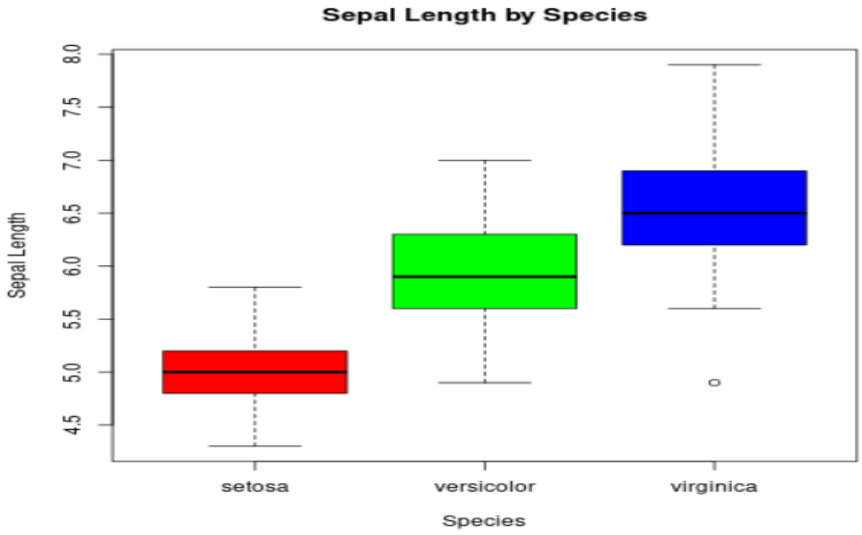
4 Horizontal Boxplot:

```
# Horizontal boxplot
data <- c(12, 25, 37, 45, 50, 65, 77, 85, 90)
boxplot(data, horizontal = TRUE, main = "Horizontal Boxplot", col = "cyan")
```



5 Boxplot Using a Dataset (Iris Example):

```
# Boxplot using Iris dataset
boxplot(Sepal.Length ~ Species, data = iris, col = c("red", "green", "blue"),
        main = "Sepal Length by Species", xlab = "Species", ylab = "Sepal Length")
```

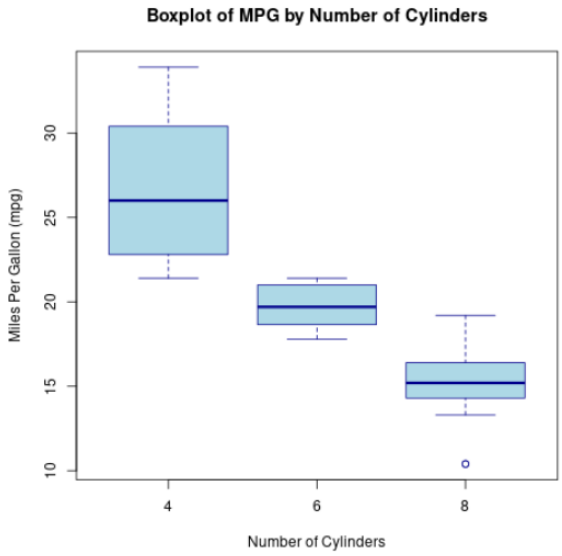


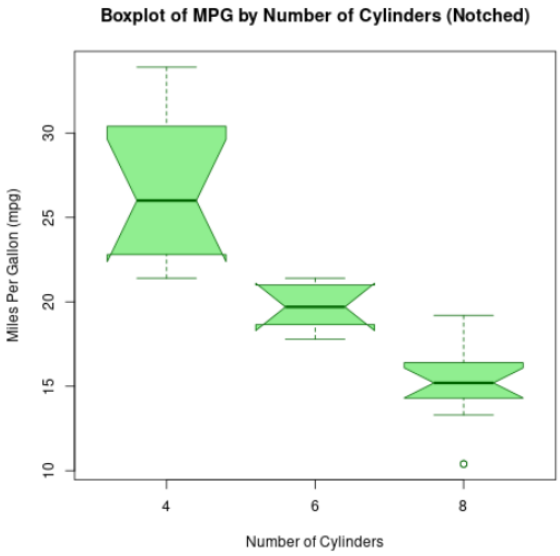
EXAMPLE PROGRAM

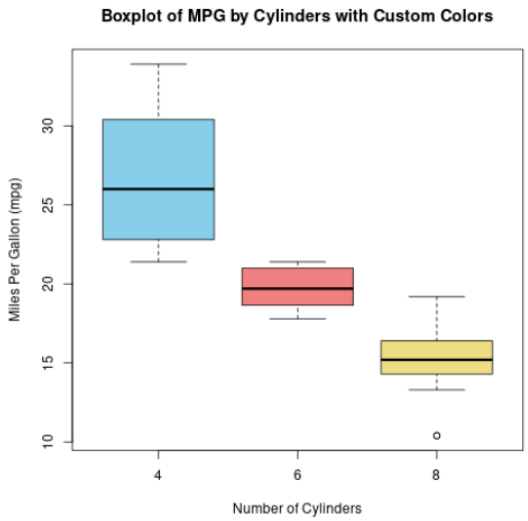
```
# ============================================
# Importing the Dataset
# ============================================
# Load the built-in mtcars dataset
data("mtcars")
# View the first few rows
head(mtcars)
```

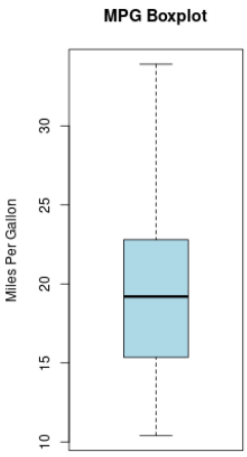
```
# ================================================
# Create a Basic Boxplot (mpg by cyl)
# ================================================
boxplot(mpg ~ cyl, data = mtcars,
        main = "Boxplot of MPG by Number of Cylinders",
        xlab = "Number of Cylinders",
        ylab = "Miles Per Gallon (mpg)",
        col = "lightblue",
        border = "darkblue")
# ================================================
# Boxplot with Notch
# ================================================
boxplot(mpg ~ cyl, data = mtcars,
        notch = TRUE,
        main = "Boxplot of MPG by Number of Cylinders (Notched)",
        xlab = "Number of Cylinders",
        ylab = "Miles Per Gallon (mpg)",
        col = "lightgreen",
        border = "darkgreen")
# ================================================
# Set Up Custom Colors
# ================================================
# Define custom colors for each box
my_colors <- c("skyblue", "lightcoral", "lightgoldenrod")
# Create boxplot with custom colors
boxplot(mpg ~ cyl, data = mtcars,
        main = "Boxplot of MPG by Cylinders with Custom Colors",
        xlab = "Number of Cylinders",
        ylab = "Miles Per Gallon (mpg)",
        col = my_colors,
        border = "black")
# ================================================
# Multiple Boxplots (mpg and hp)
# ================================================
# Set up the plotting layout (1 row, 2 columns)
par(mfrow = c(1, 2))
# Boxplot for mpg
boxplot(mtcars$mpg,
        main = "MPG Boxplot",
        ylab = "Miles Per Gallon",
```

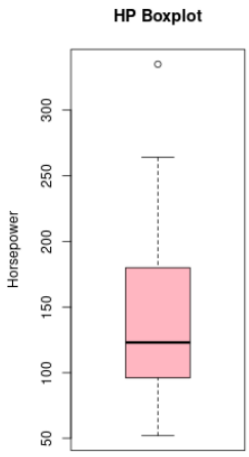
```
        col = "lightblue")
# Boxplot for hp
boxplot(mtcars$hp,
        main = "HP Boxplot",
        ylab = "Horsepower",
        col = "lightpink")
# Reset plotting layout to default (single plot)
par(mfrow = c(1, 1))
# ================================================
# Define Variables for Boxplots
# ================================================
# Define variables to boxplot
variables <- c("mpg", "hp", "wt", "qsec")
# Set up plotting layout: 2 rows x 2 columns
par(mfrow = c(2, 2))
# Loop through variables and create boxplots
for (var in variables) {
  boxplot(mtcars[[var]],
          main = paste("Boxplot of", var),
          ylab = var,
          col = "lightcyan",
          border = "blue")
}
# Reset plotting layout
par(mfrow = c(1, 1))
# ================================================
# Add Title, Labels, and Custom Colors to a Boxplot
# ================================================
boxplot(mpg ~ cyl, data = mtcars,
        main = "Enhanced Boxplot: MPG by Cylinders",
        xlab = "Cylinders",
        ylab = "Miles Per Gallon",
        col = c("orchid", "orange", "lightgreen"),
        border = "darkred",
        notch = TRUE,
        horizontal = TRUE)
# Add grid for better visualization
grid()
```

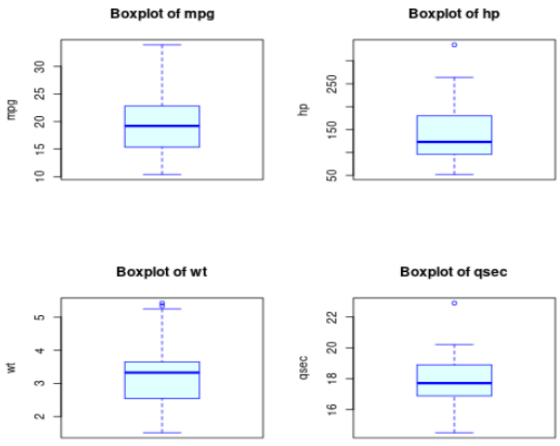

Boxplot of MPG by Number of Cylinders
Miles Per Gallon (mpg)
Number of Cylinders
10
15
20
25
30
4
6
8

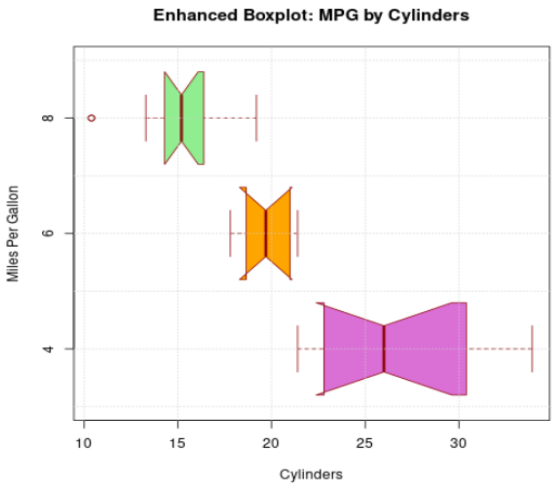

Boxplot of MPG by Number of Cylinders (Notched)
Miles Per Gallon (mpg)
Number of Cylinders
10
15
20
25
30
4
6
8


Boxplot of MPG by Cylinders with Custom Colors
Miles Per Gallon (mpg)
Number of Cylinders
10
15
20
25
30
4
6
8


MPG Boxplot
Miles Per Gallon
10
15
20
25
30


HP Boxplot
Horsepower
50
100
150
200
250
300


Boxplot of mpg
mpg
Boxplot of hp
hp
Boxplot of wt
wt
Boxplot of qsec
qsec


Enhanced Boxplot: MPG by Cylinders
Miles Per Gallon
Cylinders
4
6
8
10
15
20
25
30

3.1.10 – BAR CHARTS

A **bar chart** is used to **display categorical data** with rectangular bars. The **height/length of each bar** represents the value (like counts or means) of the corresponding category.

Basic Syntax of Bar Chart:

barplot(height, names.arg = NULL, col = NULL, main = NULL, xlab = NULL, ylab = NULL, horiz = FALSE)

Key Parameters:

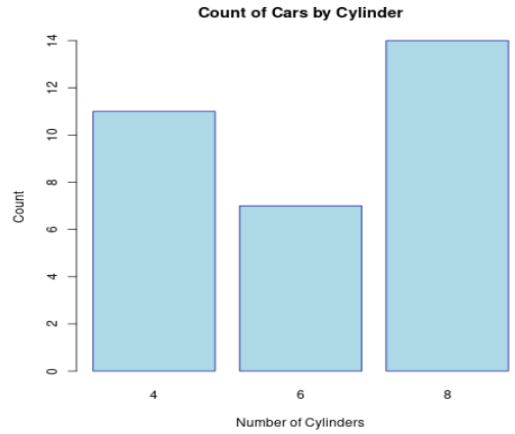
Parameter	Description
height	A vector or matrix of values to plot.
names.arg	Labels for the bars.
Col	Colors of the bars.
Main	The main title of the chart.
xlab, ylab	Labels for the x and y axes.
Horiz	If TRUE, bars are horizontal; by default, it's vertical.
border	Color of the bar borders.
legend.text	Adds a legend to the chart.

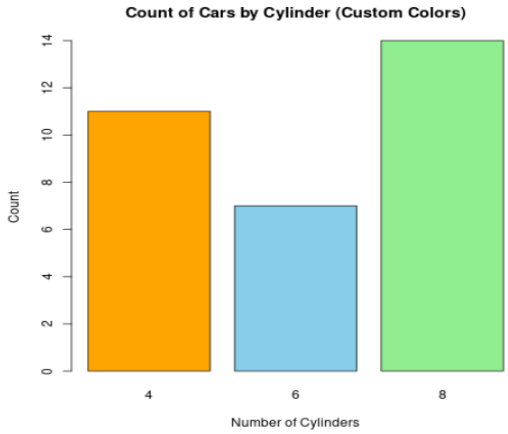
Complete Example Program: Bar Chart in R

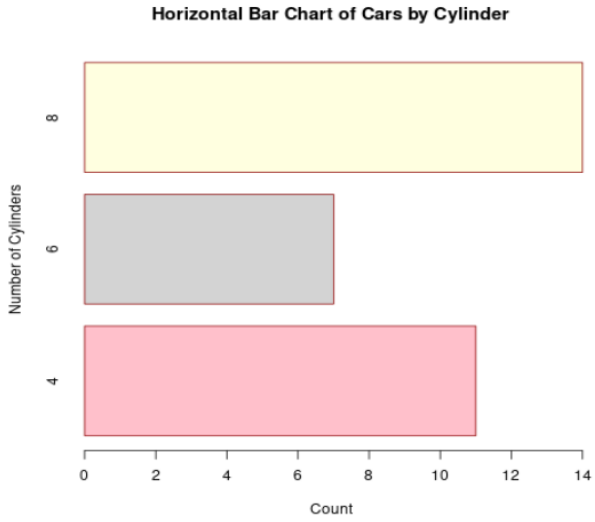
```
# ============================================
# Importing Dataset
# ============================================
# Load the built-in mtcars dataset
data("mtcars")
# View first few rows
head(mtcars)
# ============================================
# 1□ Basic Bar Chart: Count of Cars by Cylinders
# ============================================
# Get counts of cars per cylinder
cyl_counts <- table(mtcars$cyl)
# Create a basic bar chart
barplot(cyl_counts,
        main = "Count of Cars by Cylinder",
        xlab = "Number of Cylinders",
```

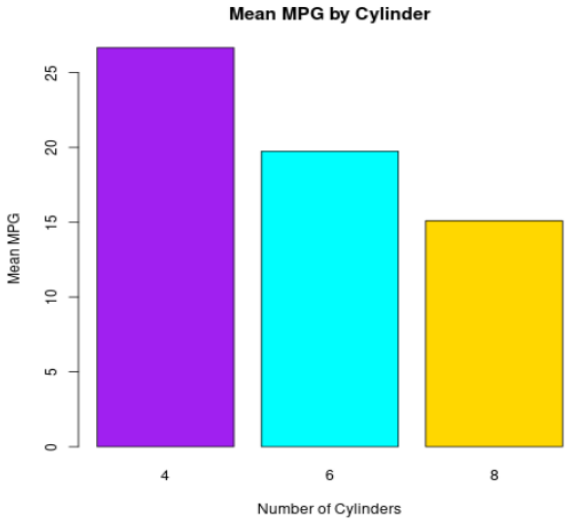
```
        ylab = "Count",
        col = "lightblue",
        border = "darkblue")
# ================================================
# 2 Bar Chart with Custom Colors
# ================================================
barplot(cyl_counts,
        main = "Count of Cars by Cylinder (Custom Colors)",
        xlab = "Number of Cylinders",
        ylab = "Count",
        col = c("orange", "skyblue", "lightgreen"),
        border = "black")
# ================================================
# 3 Horizontal Bar Chart
# ================================================
barplot(cyl_counts,
        main = "Horizontal Bar Chart of Cars by Cylinder",
        xlab = "Count",
        ylab = "Number of Cylinders",
        horiz = TRUE,
        col = c("pink", "lightgray", "lightyellow"),
        border = "darkred")
# ================================================
# 4 Bar Chart of Mean MPG by Cylinder
# ================================================
# Calculate mean mpg for each cylinder group
mean_mpg <- tapply(mtcars$mpg, mtcars$cyl, mean)
# Create barplot of mean mpg
barplot(mean_mpg,
        main = "Mean MPG by Cylinder",
        xlab = "Number of Cylinders",
        ylab = "Mean MPG",
        col = c("purple", "cyan", "gold"),
        border = "black")
# Add value labels on top of bars
bp <- barplot(mean_mpg,
              main = "Mean MPG by Cylinder with Values",
              xlab = "Number of Cylinders",
              ylab = "Mean MPG",
              col = c("purple", "cyan", "gold"),
```

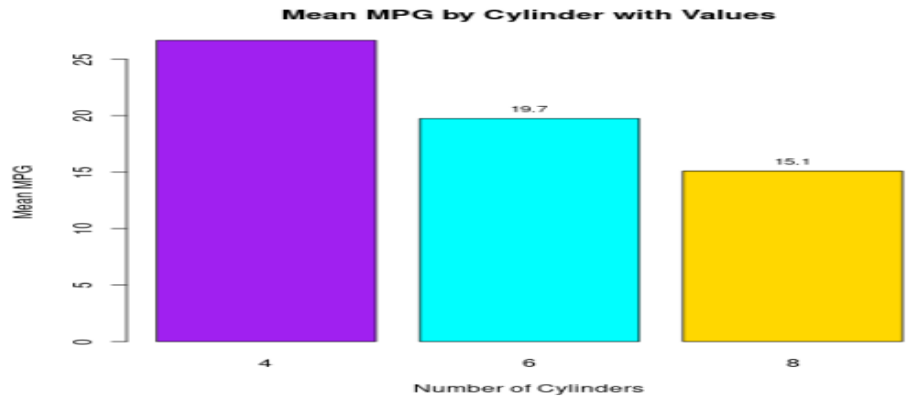
```
        border = "black")

# Add text to show exact mean mpg on bars
text(bp, mean_mpg, labels = round(mean_mpg, 1), pos = 3, cex = 0.8, col =
"black")
```











Bar Charts in R: Table of Features

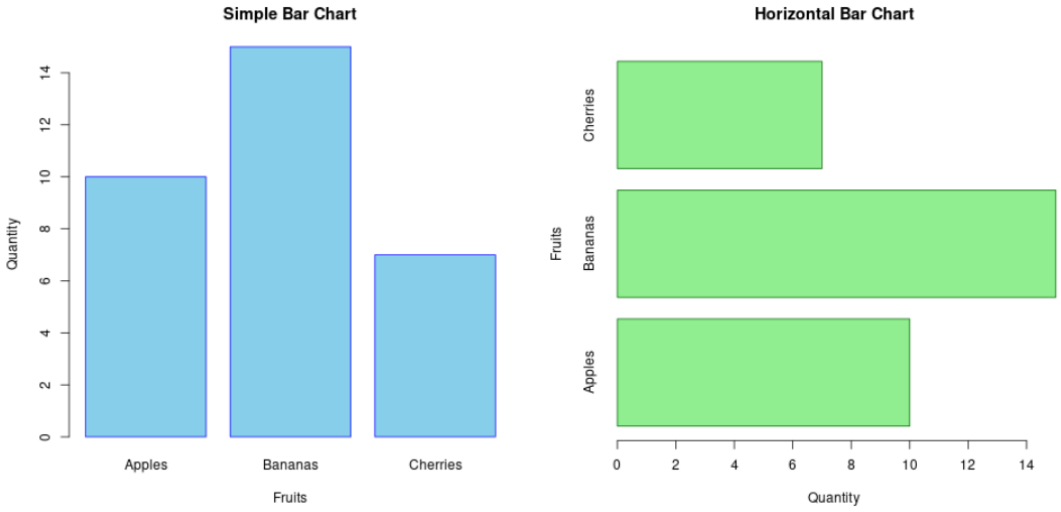
Feature	Description	Key Function/Syntax
Simple Bar Chart	Plots bars for categorical data.	barplot(height)
Horizontal Bar Chart	Bars are plotted horizontally.	barplot(height, horiz = TRUE)
Customizing the Bar Chart	Customize colors, labels, title, axis labels.	col, main, xlab, ylab, border
Adding Data Labels on Top of Each Bar	Adds numbers on top of bars to show their values.	text(bar_midpoints, data_values, labels = data_values, pos = 3)
Grouped Bar Chart	Groups bars side by side for comparison.	barplot(height_matrix, beside = TRUE)
Stacked Bar Chart	Stacks data of different groups on top of each other.	barplot(height_matrix, beside = FALSE)
Provide Names for Each Bar	Set names/labels below each bar.	names.arg = c("A", "B", "C")
Change Bar Color	Set specific colors for bars.	col = c("red", "blue", "green")
Bar Texture (Density & Angle)	Adds patterns (lines/dots) to the bars instead of solid colors.	density = c(10, 20), angle = c(45, 90)

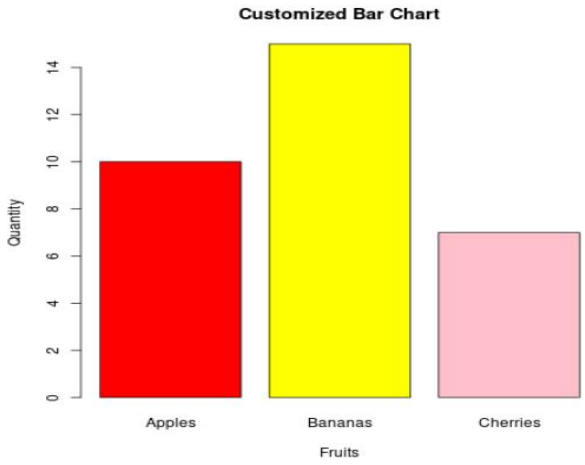
Complete R Program: Bar Charts with All Features

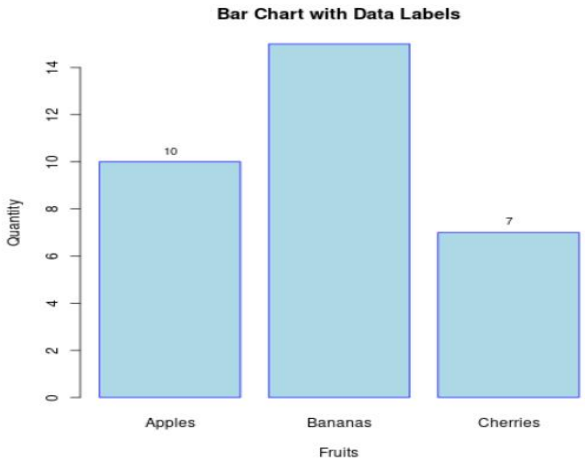
```
# ============================================
# Data Preparation
# ============================================
# Simple data vector
values <- c(10, 15, 7)
# Names for the bars
categories <- c("Apples", "Bananas", "Cherries")

# Matrix for grouped/stacked bar charts
group_data <- matrix(c(10, 20, 15, 25, 7, 14), nrow = 2, byrow = TRUE)
colnames(group_data) <- categories
rownames(group_data) <- c("2023", "2024")
# ============================================
# 1 Simple Bar Chart
# ============================================
bar_mid <- barplot(values,
                   names.arg = categories,
                   main = "Simple Bar Chart",
                   xlab = "Fruits",
                   ylab = "Quantity",
                   col = "skyblue",
                   border = "blue")
# ============================================
# 2 Horizontal Bar Chart
# ============================================
bar_mid_h <- barplot(values,
                     names.arg = categories,
                     main = "Horizontal Bar Chart",
                     xlab = "Quantity",
                     ylab = "Fruits",
                     col = "lightgreen",
                     horiz = TRUE,
                     border = "darkgreen")
# ============================================
# 3 Customized Bar Chart
# ============================================
barplot(values,
```

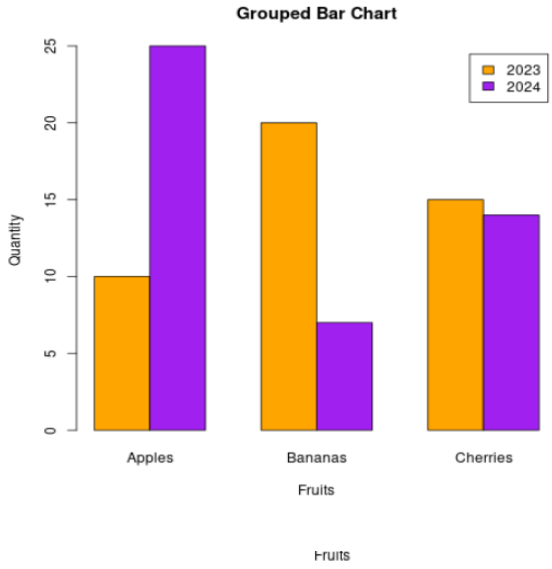
```
        names.arg = categories,
        main = "Customized Bar Chart",
        xlab = "Fruits",
        ylab = "Quantity",
        col = c("red", "yellow", "pink"),
        border = "black")
# ============================================
# 4 Adding Data Labels on Top of Bars
# ============================================
bar_mid_labels <- barplot(values,
                          names.arg = categories,
                          main = "Bar Chart with Data Labels",
                          xlab = "Fruits",
                          ylab = "Quantity",
                          col = "lightblue",
                          border = "blue")
# Add text labels on top of each bar
text(bar_mid_labels, values, labels = values, pos = 3, cex = 0.8, col = "black")
# ============================================
# 5 Grouped Bar Chart
# ============================================
barplot(group_data,
        beside = TRUE,
        main = "Grouped Bar Chart",
        col = c("orange", "purple"),
        legend.text = rownames(group_data),
        xlab = "Fruits",
        ylab = "Quantity")
# ============================================
# 6 Stacked Bar Chart
# ============================================
barplot(group_data,
        beside = FALSE,
        main = "Stacked Bar Chart",
        col = c("orange", "purple"),
        legend.text = rownames(group_data),
        xlab = "Fruits",
        ylab = "Quantity")
# ============================================
# 7 Provide Names for Each Bar & Change Colors
```

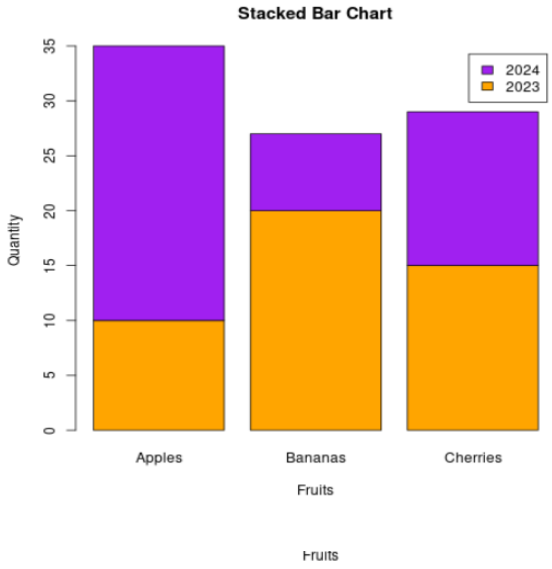
```
# ================================================
barplot(values,
        names.arg = c("Apple", "Banana", "Cherry"),
        main = "Named & Colored Bars",
        xlab = "Fruits",
        ylab = "Quantity",
        col = c("darkred", "gold", "deeppink"),
        border = "black")
# ================================================
# 8□ Bar Texture (Density & Angle)
# ================================================
barplot(values,
        names.arg = categories,
        main = "Bar Texture with Density & Angle",
        xlab = "Fruits",
        ylab = "Quantity",
        density = c(10, 20, 30),
        angle = c(45, 90, 135),
        col = "gray",
        border = "black")
```

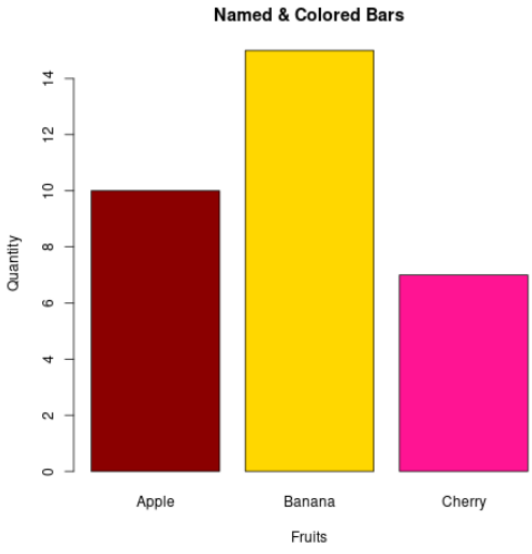


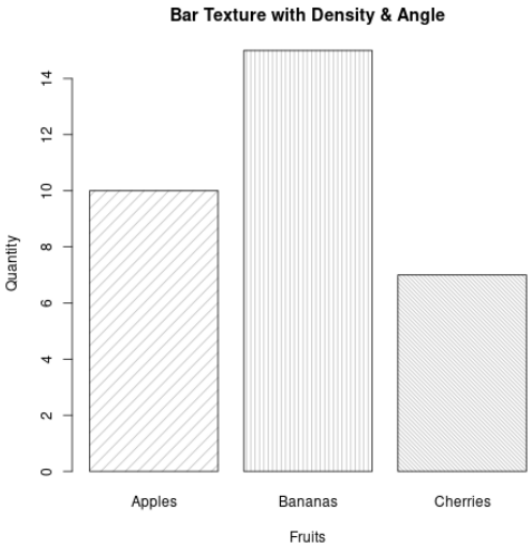

Customized Bar Chart
Quantity
0
2
4
6
8
10
12
14
Apples
Bananas
Cherries
Fruits


Bar Chart with Data Labels
Quantity
0
2
4
6
8
10
12
14
10
7
Apples
Bananas
Cherries
Fruits


Grouped Bar Chart
Quantity
0
5
10
15
20
25
2023
2024
Apples
Bananas
Cherries
Fruits
Fruits


Stacked Bar Chart
Quantity
0
5
10
15
20
25
30
35
2024
2023
Apples
Bananas
Cherries
Fruits
Fruits


Named & Colored Bars
Quantity
0
2
4
6
8
10
12
14
Apple
Banana
Cherry
Fruits


Bar Texture with Density & Angle
Quantity
0
2
4
6
8
10
12
14
Apples
Bananas
Cherries
Fruits

3.1.11 – LINE GRAPHS

A **line graph** (or **line chart**) is a type of plot used to display data points connected by straight lines. It is especially useful to:

- Show **trends** over time or continuous data.
- Visualize **relationships** between two numerical variables.
- Compare **multiple data series** on the same plot.

Feature	Description	Key Function/Syntax
Simple Line Graph	Basic line connecting data points.	plot(x, y, type = "l")
Line Graph with Points	Adds points on the line.	plot(x, y, type = "o")
Multiple Lines	Plot multiple lines on the same graph.	Use lines() function after the first plot.
Custom Colors & Line Types	Set colors, line types, and width.	col, lty, lwd parameters.
Add Title, Labels, Grid	Adds main title and axis labels.	main, xlab, ylab, grid()
Legend for Multiple Lines	Adds a legend to identify lines.	legend("topright", legend = ..., col = ..., lty = ...)
Line Graph from CSV	Reads data from CSV and plots.	data <- read.csv("file.csv") and plot(data$X, data$Y, type = "l")

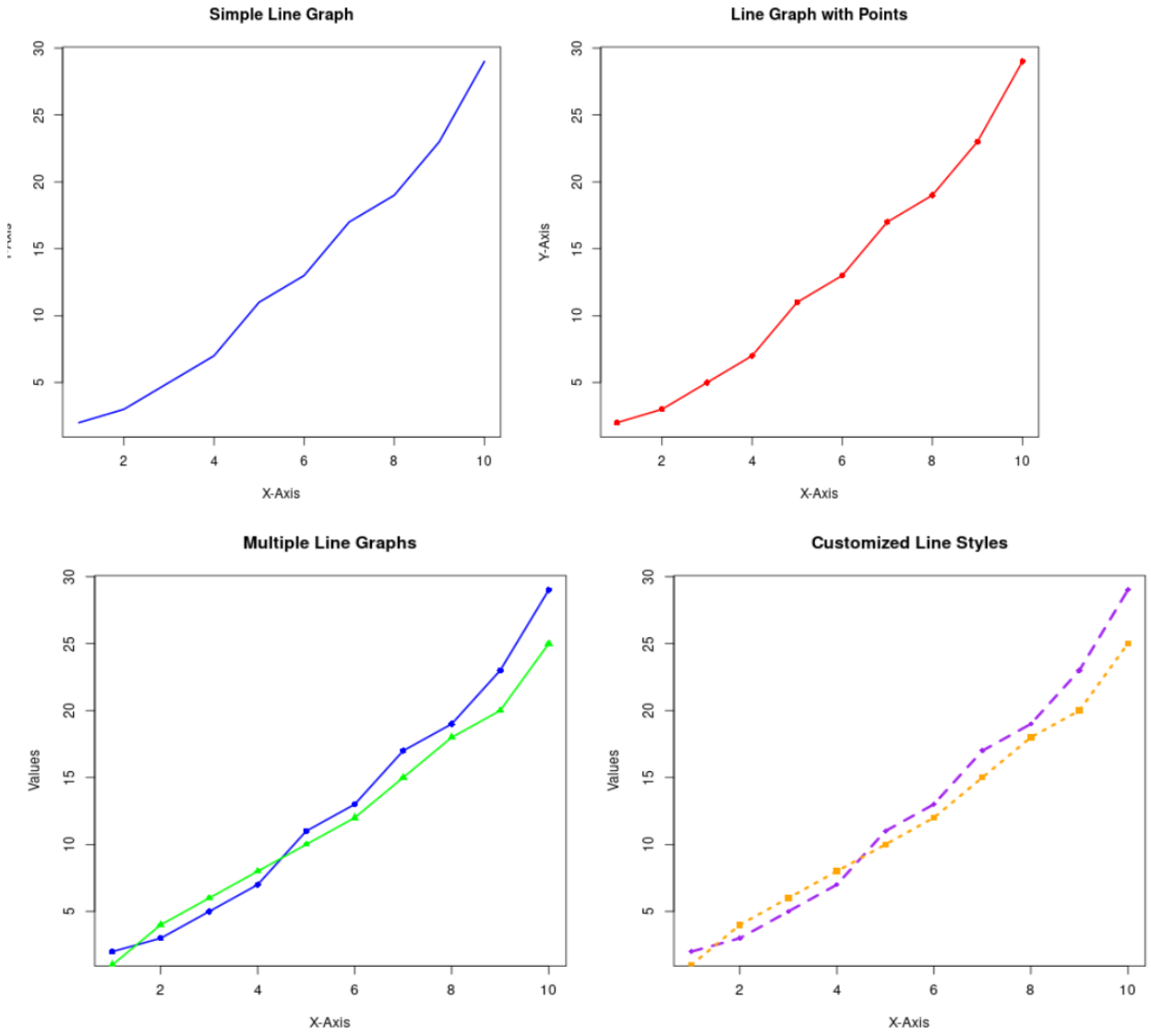
EXAMPLE PROGRAM

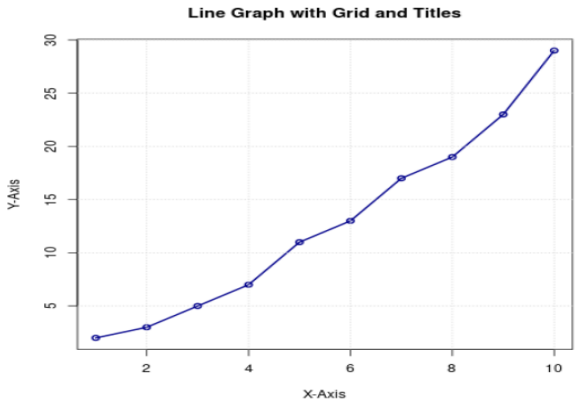
```
# ======================================================
# Example Data
# ======================================================
x <- 1:10
y1 <- c(2, 3, 5, 7, 11, 13, 17, 19, 23, 29)
y2 <- c(1, 4, 6, 8, 10, 12, 15, 18, 20, 25)
# ======================================================
```

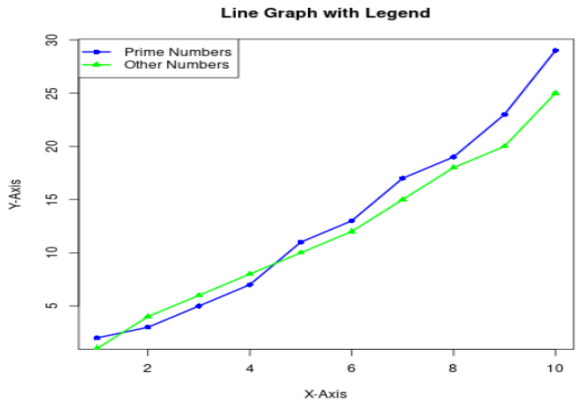
```
# 1 Simple Line Graph
# ======================================================
plot(x, y1, type = "l",
     main = "Simple Line Graph",
     xlab = "X-Axis",
     ylab = "Y-Axis",
     col = "blue",
     lwd = 2)
# ======================================================
# 2 Line Graph with Points
# ======================================================
plot(x, y1, type = "o",
     main = "Line Graph with Points",
     xlab = "X-Axis",
     ylab = "Y-Axis",
     col = "red",
     pch = 16,
     lwd = 2)
# ======================================================
# 3 Multiple Lines on Same Graph
# ======================================================
plot(x, y1, type = "o",
     main = "Multiple Line Graphs",
     xlab = "X-Axis",
     ylab = "Values",
     col = "blue",
     pch = 16,
     lwd = 2)
lines(x, y2, type = "o", col = "green", pch = 17, lwd = 2)
# ======================================================
# 4 Customize Colors, Line Types & Width
# ======================================================
plot(x, y1, type = "o",
     main = "Customized Line Styles",
     xlab = "X-Axis",
     ylab = "Values",
     col = "purple",
     lty = 2,     # dashed line
     lwd = 3,
     pch = 18)
```

```
lines(x, y2, type = "o", col = "orange", lty = 3, lwd = 3, pch = 15)
# ========================================================
# 5 Add Grid, Title, and Axis Labels
# ========================================================
plot(x, y1, type = "o",
     main = "Line Graph with Grid and Titles",
     xlab = "X-Axis",
     ylab = "Y-Axis",
     col = "darkblue",
     lwd = 2)
grid()
# ========================================================
# 6 Add Legend for Multiple Lines
# ========================================================
plot(x, y1, type = "o",
     main = "Line Graph with Legend",
     xlab = "X-Axis",
     ylab = "Y-Axis",
     col = "blue",
     pch = 16,
     lwd = 2)
lines(x, y2, type = "o", col = "green", pch = 17, lwd = 2)
legend("topleft",
       legend = c("Prime Numbers", "Other Numbers"),
       col = c("blue", "green"),
       lty = 1,
       pch = c(16, 17),
       lwd = 2)
# ========================================================
# 7 BONUS: Import Data from CSV and Plot
# ========================================================
# Assuming you have a CSV file named 'line_data.csv' with columns 'x' and 'y'
# Example CSV content:
# x,y
# 1,5
# 2,7
# 3,9
# 4,6
# 5,8
# Read data from CSV
```

```
data <- read.csv("line_data.csv")
# Plot from CSV
plot(data$x, data$y, type = "o",
    main = "Line Graph from CSV Data",
    xlab = "X-Axis",
    ylab = "Y-Axis",
    col = "darkred",
    pch = 19,
    lwd = 2
```







3.1.12 – SCATTERPLOTS

A **scatterplot** is a **graphical representation** used to display the **relationship between two numerical variables**. Each point (dot) on the plot represents one **observation** from the dataset.

Feature	Description	Key Function/Syntax
Simple Scatterplot	Plots data points as (x, y) on a 2D chart.	plot(x, y)
Custom Colors & Point Shapes	Customize color, shape, and size of points.	col, pch, cex parameters.
Add Titles & Labels	Add main title and axis labels.	main, xlab, ylab.
Add Regression Line	Adds a best-fit regression line.	abline(lm(y ~ x)).
Multiple Groups with Legend	Plot multiple groups with different colors and add legend.	legend() function.
Smooth Scatterplot (Density)	For large datasets, shows smoothed density plot.	smoothScatter(x, y).
Scatterplot Matrix	Displays scatterplots for multiple variables in matrix form.	pairs(dataframe).

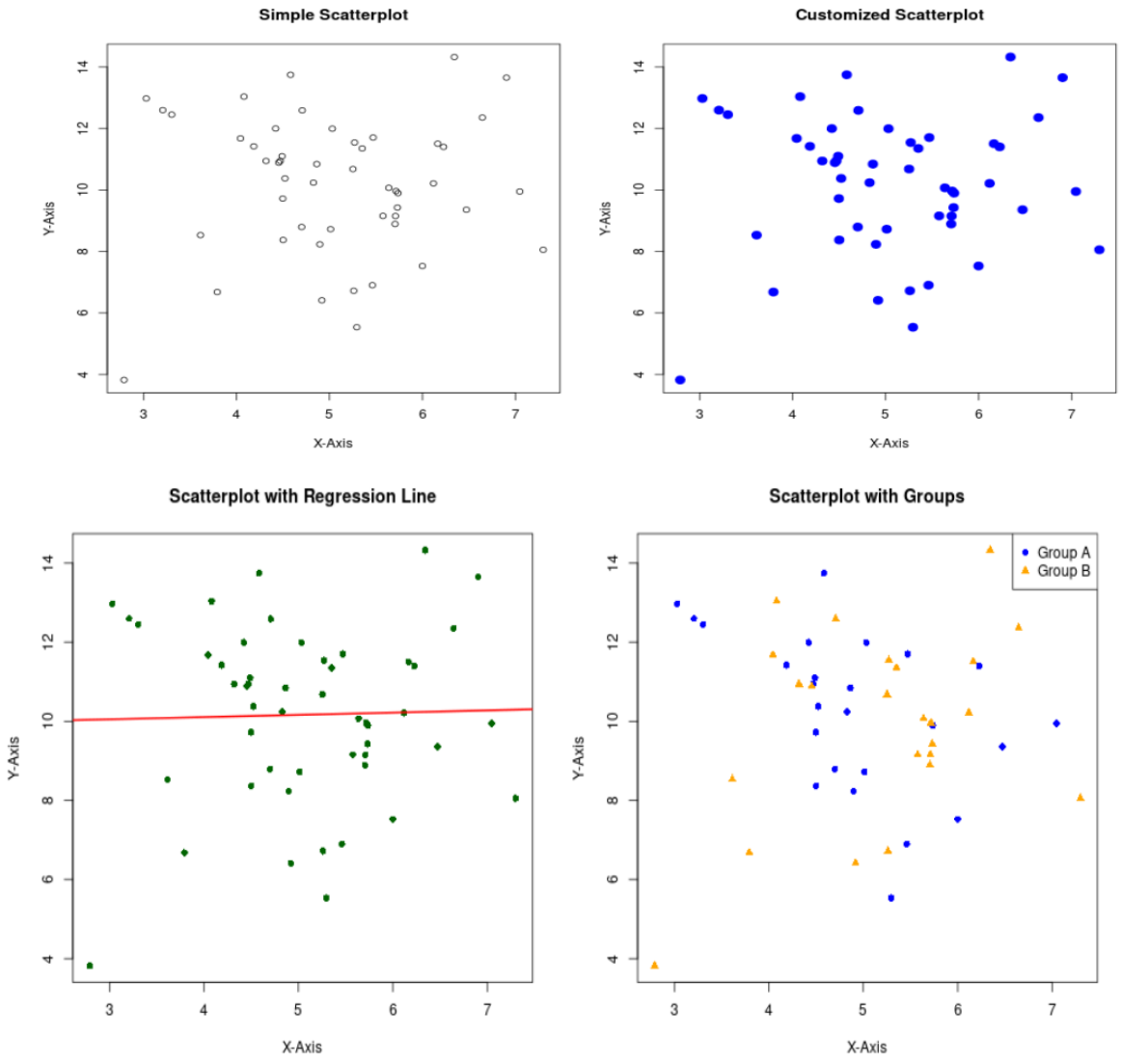
Scatterplot from CSV	Read data from CSV and plot.	data <- read.csv("file.csv"); plot(data$x, data$y).

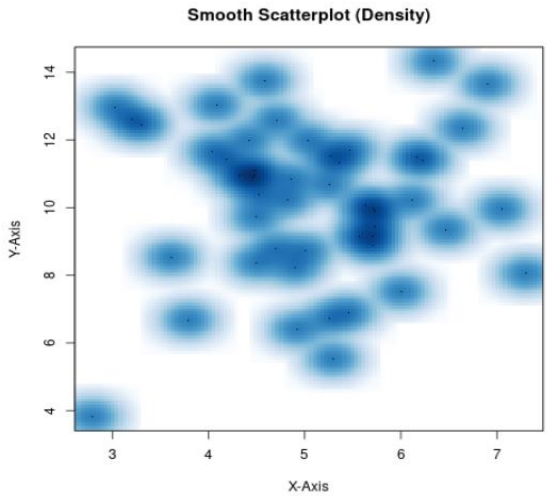
Complete R Program: Scatterplots with All Features

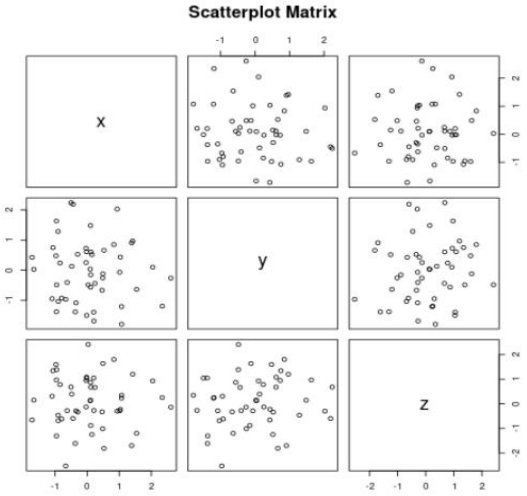
```
# ======================================================
# Example Data
# ======================================================
x <- rnorm(50, mean = 5, sd = 1)
y <- rnorm(50, mean = 10, sd = 2)
group <- rep(c("A", "B"), each = 25)

# ======================================================
# 1 Simple Scatterplot
# ======================================================
plot(x, y,
     main = "Simple Scatterplot",
     xlab = "X-Axis",
     ylab = "Y-Axis")
# ======================================================
# 2 Customized Scatterplot (Color, Shape, Size)
# ======================================================
plot(x, y,
     main = "Customized Scatterplot",
     xlab = "X-Axis",
     ylab = "Y-Axis",
     col = "blue",
     pch = 19,  # solid circle
     cex = 1.5) # point size
# ======================================================
# 3 Scatterplot with Regression Line
# ======================================================
plot(x, y,
     main = "Scatterplot with Regression Line",
     xlab = "X-Axis",
     ylab = "Y-Axis",
     col = "darkgreen",
     pch = 16)
abline(lm(y ~ x), col = "red", lwd = 2)
# ======================================================
```

```
# 4 Multiple Groups with Legend
# ======================================================
colors <- c("blue", "orange")
plot(x[group == "A"], y[group == "A"],
    main = "Scatterplot with Groups",
    xlab = "X-Axis",
    ylab = "Y-Axis",
    col = colors[1],
    pch = 16,
    xlim = range(x),
    ylim = range(y))
points(x[group == "B"], y[group == "B"], col = colors[2], pch = 17)
legend("topright",
      legend = c("Group A", "Group B"),
      col = colors,
      pch = c(16, 17))
# ======================================================
# 5 Smooth Scatterplot (Density)
# ======================================================
smoothScatter(x, y,
           main = "Smooth Scatterplot (Density)",
           xlab = "X-Axis",
           ylab = "Y-Axis")
# ======================================================
# 6  Scatterplot Matrix
# ======================================================
data_matrix <- data.frame(x = rnorm(50),
                      y = rnorm(50),
                      z = rnorm(50))
pairs(data_matrix,
     main = "Scatterplot Matrix")
# ======================================================
# 7  BONUS: Import Data from CSV and Scatterplot
# ======================================================
# Assuming 'scatter_data.csv' with columns 'x' and 'y'
# Example CSV:
# x,y
# 1,2
# 2,3
# 3,5
```

```
# 4,4
# 5,6
# Read data
data_csv <- read.csv("scatter_data.csv")
# Plot from CSV
plot(data_csv$x, data_csv$y,
    main = "Scatterplot from CSV",
    xlab = "X-Axis",
    ylab = "Y-Axis",
    col = "purple",
    pch = 19)
```







3.1.13 – PIE CHARTS

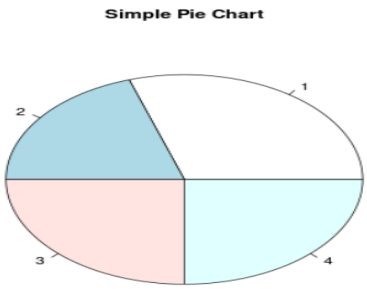
A **pie chart** is a **circular statistical graphic** divided into **slices (sectors)** to illustrate **numerical proportions**. Each slice of the pie shows the **relative size** or **percentage** of a category out of the total. It is **useful for displaying the parts-to-whole relationship**.

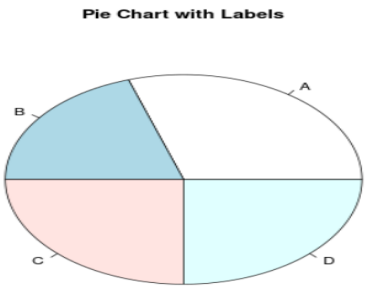
Feature	Description	Key Function/Syntax
Simple Pie Chart	Create a basic pie chart.	pie(values)
Labels	Add labels to slices.	labels parameter in pie().
Colors	Set custom colors for slices.	col parameter.
Percentage Labels	Show percentage instead of just names.	Use paste() with round() for percentage labels.
Exploded Pie (Pull-Out Slice)	Highlight a slice by "exploding" it.	explode logic via pie() with radius tweak (base R).
3D Pie Chart	Make a 3D pie chart.	Use plotrix::pie3D().
Legend	Add a legend for clarity.	legend() function.
Pie Chart from CSV	Import data from a CSV and plot a pie chart.	read.csv() + pie().

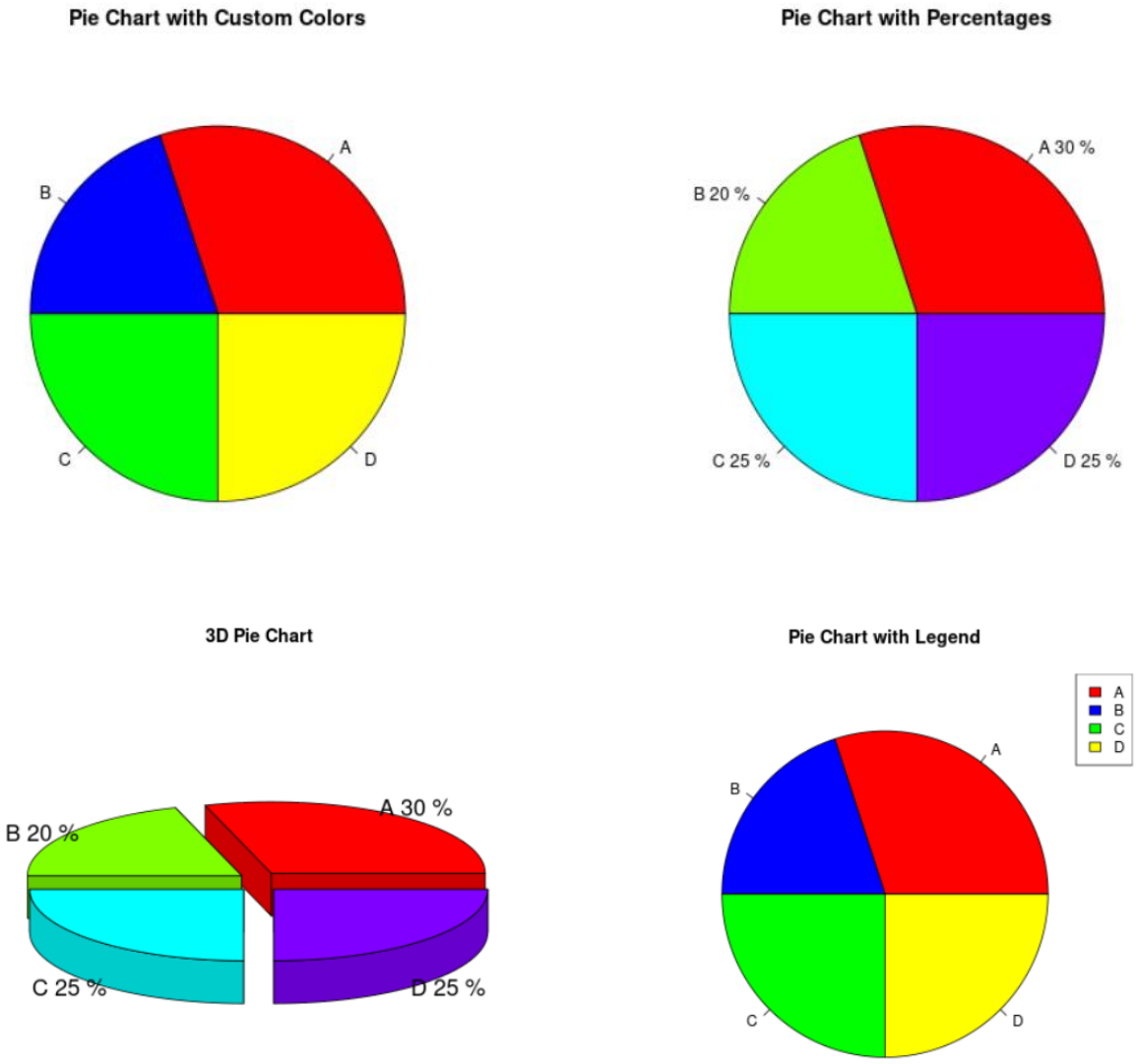
Complete R Program: Pie Charts with All Features

```
# ========================================================
# Example Data
# ========================================================
slices <- c(30, 20, 25, 25)
labels <- c("A", "B", "C", "D")
# ========================================================
# 1 Simple Pie Chart
# ========================================================
pie(slices,
    main = "Simple Pie Chart")
# ========================================================
# 2 Pie Chart with Labels
# ========================================================
pie(slices,
    labels = labels,
    main = "Pie Chart with Labels")

# ========================================================
# 3 Pie Chart with Custom Colors
# ========================================================
colors <- c("red", "blue", "green", "yellow")
pie(slices,
    labels = labels,
    col = colors,
    main = "Pie Chart with Custom Colors")
# ========================================================
# 4 Pie Chart with Percentages
# ========================================================
pct <- round(slices/sum(slices) * 100)
labels_pct <- paste(labels, pct, "%")
pie(slices,
    labels = labels_pct,
    col = rainbow(length(slices)),
    main = "Pie Chart with Percentages")
# ========================================================
# 5 3D Pie Chart (Requires plotrix package)
# ========================================================
# Install if needed: install.packages("plotrix")
library(plotrix)
```

```
pie3D(slices,
      labels = labels_pct,
      explode = 0.1,
      main = "3D Pie Chart")
# ========================================================
# 6 Pie Chart with Legend
# ========================================================
pie(slices,
    labels = labels,
    col = colors,
    main = "Pie Chart with Legend")
legend("topright",
       legend = labels,
       fill = colors)
# ========================================================
# 7 BONUS: Pie Chart from CSV
# ========================================================
# Example CSV: pie_data.csv
# Category,Value
# A,30
# B,20
# C,25
# D,25
# Read CSV
data_csv <- read.csv("pie_data.csv")
# Plot from CSV
pie(data_csv$Value,
    labels = data_csv$Category,
    col = rainbow(length(data_csv$Value)),
    main = "Pie Chart from CSV")
```







3.2 Let Us Sum Up

R provides powerful tools to import data from various sources like CSV files, XML, JSON, databases, web data, and Excel files using functions such as read.csv(), read_excel(), and database connectors. Once imported, R offers rich visualization options. Histograms display data distribution; boxplots show spread and outliers; bar charts represent categorical data; line graphs track trends over time; scatterplots reveal relationships between two variables; and pie charts illustrate proportions. Each chart type can be customized with colors, labels, and layouts. These tools make R an essential platform for data analysis and graphical representation.

3.3 Check Your Progress

1. Which function is used to read a CSV file in R?
   A) read.table()
   B) read.csv()
   C) read.xlsx()
   D) readRDS()
2. Which package is used for importing Excel files in R?
   A) jsonlite
   B) readxl
   C) RCurl
   D) XML
3. What does the function fromJSON() do?
   A) Writes data to JSON format
   B) Reads JSON files into R
   C) Parses XML data
   D) Reads Excel files
4. Which package helps fetch web data in R?
   A) dplyr
   B) ggplot2
   C) RCurl
   D) readr
5. Which function is used to read XML files?
   A) read.xml()
   B) xmlParse()
   C) read_json()
   D) xmlread()
6. To connect R with a MySQL database, which package is generally used?
   A) DBI
   B) readxl

C) jsonlite

D) XML

7. Which argument in read.csv() allows specifying if the file has headers?

   A) has.header

   B) head

   C) header

   D) csv.header

8. What type of data is imported using read_json()?

   A) CSV

   B) XML

   C) JSON

   D) Excel

9. Which is a correct way to import an Excel file named “data.xlsx”?

   A) read.xlsx("data.xlsx")

   B) read_excel("data.xlsx")

   C) read.csv("data.xlsx")

   D) readxl("data.xlsx")

10. What is the purpose of col_types in the read_excel() function?

    A) To rename columns

    B) To specify data types of columns

    C) To merge columns

    D) To delete columns

11. Which R function is used to view the first few rows of a dataset?

    A) head()

    B) top()

    C) show()

    D) view.head()

12. What is the default separator used in read.csv()?

    A) Space

    B) Tab

C) Comma

D) Semicolon

13. What is TRUE about read.table() in R?

A) It can only read CSV files.

B) It is more general than read.csv().

C) It is used for reading Excel files.

D) It only reads text files.

14. Which function is used to write data to a CSV file?

A) save.csv()

B) write.csv()

C) export.csv()

D) csv.write()

15. Which R function allows you to preview the structure of a dataset?

A) glimpse()

B) structure()

C) str()

D) view()

16. Which function is used to create a histogram in R?

A) barplot()

B) pie()

C) hist()

D) boxplot()

17. What is the main function of boxplot()?

A) Show frequency distribution

B) Visualize spread and outliers

C) Display proportions

D) Draw line graphs

18. Which R function creates a bar chart?

A) barplot()

B) hist()

C) pie()

D) boxplot()

19. What is the primary use of a scatterplot?

A) Display categorical data

B) Show relationships between two variables

C) Show proportions

D) Display frequency

20. What does the col parameter do in a plot?

A) Sets the size

B) Changes the shape

C) Specifies colors

D) Sets titles

21. Which function plots a pie chart?

A) piechart()

B) pie()

C) circle()

D) chart()

22. What argument in hist() sets the number of bins?

A) breaks

B) bins

C) size

D) width

23. Which plot type is best for showing time series data?

A) Pie chart

B) Line graph

C) Bar chart

D) Boxplot

24. What does main do in a plotting function?

A) Sets legend

B) Sets axis label

C) Adds main title

D) Changes color

25. What function is used to plot a line graph?

A) plot(type="o")

B) lineplot()

C) hist()

D) pie()

26. Which plot is best to display distribution and detect outliers?

A) Line graph

B) Scatterplot

C) Boxplot

D) Pie chart

27. In a barplot, what does names.arg specify?

A) Bar colors

B) Bar width

C) Labels for each bar

D) Number of bars

28. What is used to draw multiple boxplots for different groups?

A) boxplot(~~group)~~

~~B) boxplot(group~~variable)

C) boxplot(variable~group)

D) boxplot(group=variable)

29. In pie(), how can you start the pie at a specific angle?

A) start

B) start_angle

C) angle

D) init.angle

30. Which plot shows data as a circle divided into slices?

A) Bar chart

B) Pie chart

C) Scatterplot

D) Boxplot

3.4 Unit Summary

This unit focused on importing external data into R from sources like CSV files, Excel, XML, JSON, web data, and databases using packages such as readxl, jsonlite, and DBI. It also covered creating and customizing various charts and graphs in R, including histograms, boxplots, bar charts, line graphs, scatterplots, and pie charts, helping to visualize and interpret data effectively.

3.5 Glossary

- **CSV (Comma Separated Values):** A simple file format used to store tabular data in plain text, where each line represents a data record.
- **Excel File:** A spreadsheet file format (.xls or .xlsx) used for storing data in rows and columns.
- **JSON (JavaScript Object Notation):** A lightweight data-interchange format that is easy for humans to read and write and for machines to parse and generate.
- **XML (eXtensible Markup Language):** A markup language used to encode documents in a format that is both human-readable and machine-readable.
- **Database:** An organized collection of structured data, often accessed electronically.
- **Histogram:** A graphical display of data using bars of different heights to show the frequency distribution of a dataset.
- **Boxplot:** A graphical representation showing the median, quartiles, and outliers of a dataset, also called a box-and-whisker plot.
- **Bar Chart:** A graph that presents categorical data with rectangular bars proportional to the values they represent.
- **Line Graph:** A type of chart that displays information as a series of data points connected by straight lines, useful for showing trends over time.

- **Scatterplot:** A graph that shows the relationship between two continuous variables using dots.
- **Pie Chart:** A circular statistical graphic divided into slices to illustrate numerical proportions.
- **read.csv():** A function in R to read CSV files.
- **readxl:** An R package used to read Excel files.
- **jsonlite:** An R package for reading and writing JSON data.
- **DBI:** A database interface package in R used for connecting to databases.
- **ggplot2:** A powerful R package used for data visualization.
- **Frequency:** The number of times a particular value or range of values occurs in a dataset.
- **Outlier:** A data point that differs significantly from other observations.
- **Median:** The middle value of a dataset when sorted in order.
- **IQR (Interquartile Range):** The range between the first and third quartiles (middle 50% of data).

3.6 Self – Assessment Questions

1. What is the purpose of the read.csv() function in R?

2. How do you import an Excel file into R?

3. Name one R package used to read JSON files.

4. What type of data can be read using the XML package in R?

5. How do you connect R to a database?

6. What is a histogram used for in data visualization?

7. Which R function is commonly used to create a boxplot?

8. What type of data is best visualized using a bar chart?

9. How is a line graph different from a bar chart?

10. What is the main use of a scatterplot?

11. What does a pie chart represent?

12. What is the function of ggplot2 in R?

13. What is meant by "categorical data"?

14. How can you change the color of bars in a bar chart?

15. Define the term "outlier" in a boxplot.

16. Which function in R allows reading data without a header row?

17. What parameter is used to specify column types when reading data?

18. What is the Interquartile Range (IQR)?

19. How can you add labels to a pie chart in R?

20. What does a stacked bar chart show?

21. How do you read specific rows from an Excel file in R?

22. What is a grouped bar chart?

23. Explain the role of the "whiskers" in a boxplot.

24. What argument in a histogram function controls the number of bins?

25. How can you modify content in an Excel file using R?

26. What function allows you to merge two datasets in R?

27. How do you plot multiple boxplots side by side in R?

28. What is the role of the DBI package?

29. How can you customize the angle of a pie chart in R?

30. What is the purpose of using JSON data in R?

3.7Activities / Exercises / Case Studies

1 **Activity:**

Importing CSV Files

- Import the file student_scores.csv into R and display the first 10 rows.

2 **Exercise:**

Reading Excel Files

- Read an Excel file named sales_data.xlsx and plot a bar chart of total sales per region.

3**Activity:**

JSON Data Handling

- Import a JSON file containing employee records and extract the names and departments.

4**Exercise:**

Creating Histograms

- Use the mtcars dataset to create a histogram of the mpg column and customize its color.

5**Activity:**

Boxplots Comparison

- Create side-by-side boxplots for mpg across different cyl values from the mtcars dataset.

6**Exercise:**

Scatterplots

- Plot wt vs. mpg using a scatterplot and add a regression line.

7 **Activity:**

Line Graph

- Create a line graph showing sales trends over 12 months using dummy data.

8 **Exercise:**

Pie Chart

- Make a pie chart of market share for 5 companies and add labels and a custom color palette.

Case Studies:

Case Study 1: Sales Performance Analysis

- Scenario: You are provided with CSV and Excel files of a company's quarterly sales across different regions.
- Tasks:
  - Import the data.
  - Clean and merge the files.
  - Create histograms for sales distribution.
  - Use boxplots to compare sales across regions.
  - Create a bar chart showing top-performing regions.

Case Study 2: Social Media Engagement

- Scenario: A JSON dataset contains engagement data (likes, comments, shares) for various social media posts.
- Tasks:
  - Import the JSON file.
  - Visualize the data using bar charts and line graphs.
  - Use boxplots to analyze engagement variability.
  - Create a pie chart to show post types (photo, video, text) proportion.

Case Study 3: Vehicle Analysis

- Scenario: Using the built-in mtcars dataset.
- Tasks:
    - Explore relationships between horsepower and mpg using scatterplots.
    - Create boxplots grouped by the number of cylinders.
    - Summarize data with histograms.
    - Display pie charts showing the proportion of cars with different gear types.

3.8 Answers For Check Your Progress

1 B) read.csv()

2 B) readxl

3 B) Reads JSON files into R

4 C) RCurl

5 B) xmlParse()

6 A) DBI

7 C) header

8 C) JSON

9 B) read_excel("data.xlsx")

10 B) To specify data types of columns

11 A) head()

12 C) Comma

13 B) It is more general than read.csv().

14 B) write.csv()

15 C) str()

16 C) hist()

17 B) Visualize spread and outliers

18 A) barplot()

19 B) Show relationships between two variables

20 C) Specifies colors

21 B) pie()

22 A) breaks

23 B) Line graph

24 C) Adds main title

25 A) plot(type="o")

26 C) Boxplot

27 C) Labels for each bar

28 C) boxplot(variable~group)

29 D) init.angle

30 B) Pie chart

3.9 References and Suggested Readings

1. readr for CSV files – https://cran.r-project.org/web/packages/readr/
2. readxl for Excel – https://cran.r-project.org/web/packages/readxl/
3. jsonlite for JSON – https://cran.r-project.org/web/packages/jsonlite/
4. XML and xml2 for XML files – https://cran.r-project.org/web/packages/XML/
5. RODBC and DBI for databases – https://cran.r-project.org/web/packages/RODBC/
6. httr and rvest for Web data – https://cran.r-project.org/web/packages/rvest/
7. **R Graph Gallery** – https://r-graph-gallery.com/- A practical and visually rich resource for all types of charts and plots in R (histograms, boxplots, bar charts, line graphs, etc.)
8. **Quick-R by DataCamp** – https://www.statmethods.net/- Handy reference for quick implementation of plots using base R graphics.
9. **Murrell, P. (2011).** *R Graphics*. 2nd Edition, Chapman & Hall/CRC.

UNIT IV
Random Forest, Decision Tree, Normal and Binomial distributions, Time Series Analysis, Linear and Multiple Regression, Logistic Regression, Survival Analysis.

RANDOM FOREST

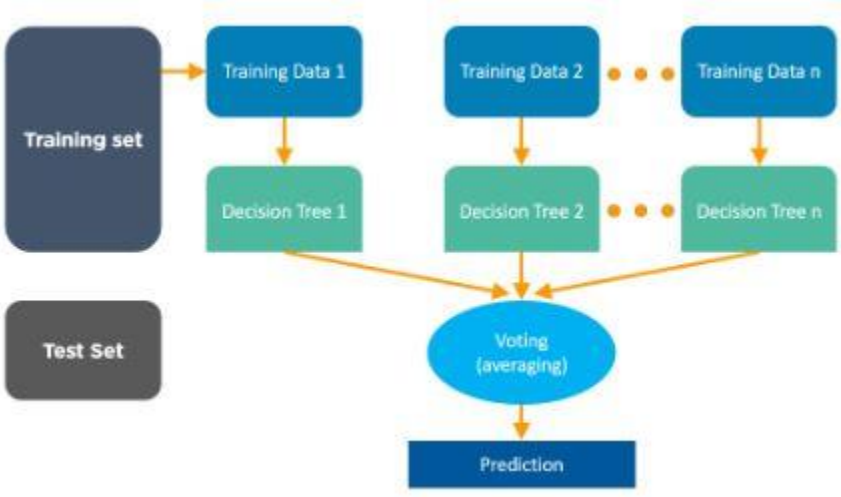
UNIT OBJECTIVE

This unit aims to provide an overview of key statistical and machine learning techniques including Random Forest, Decision Tree, Normal and Binomial distributions, Time Series Analysis, Linear and Multiple Regression, Logistic Regression, and Survival Analysis. Learners will understand the theory and practical application of these methods for data modeling, prediction, and interpretation, with a focus on implementing them in R programming for real-world data analysis tasks.

SECTION 4.1: RANDOM FOREST

4.1 RANDOM FOREST

Random Forest is an ensemble machine learning algorithm that builds multiple decision trees during training and merges their results (by averaging for regression or voting for classification) to improve accuracy and prevent overfitting. It is widely used for both classification and regression tasks.



Why We Use Random Forest:

- It improves prediction accuracy by combining multiple decision trees.
- It reduces the risk of overfitting that is common in single decision trees.

- It can handle large datasets with higher dimensionality (many features).
- It provides feature importance, helping to identify significant predictors.

Steps in Random Forest:

1. **Data Sampling:**
    - Create multiple random subsets (bootstrap samples) from the original dataset.
2. **Tree Construction:**
    - Build a decision tree for each sample using a random subset of features at each split.
3. **Splitting:**
    - Each tree splits the data based on the best feature among the randomly selected features.
4. **Tree Growth:**
    - Trees are grown to the maximum depth without pruning.
5. **Prediction:**
    - For **classification:** each tree votes for a class, and the majority vote becomes the final prediction.
    - For **regression:** the average of all tree outputs is the final prediction.
6. **Aggregation:**
    - Combine the results of all trees for the final output.

Important Terminologies:

- **Bootstrap Aggregation (Bagging):** Sampling data with replacement to train each tree.
- **Feature Subset:** Random selection of features for splitting at each node.
- **Ensemble Learning:** Combining multiple models (trees) to improve performance.
- **Out-of-Bag (OOB) Error:** Error estimation using data not included in the bootstrap sample.

- **Gini Index / Entropy:** Measures of impurity used to split nodes in classification tasks.
- **Variance Reduction:** Criteria used in regression trees to decide splits.
- **Majority Voting:** The method of choosing the class predicted by most trees.
- **Feature Importance:** A ranking of how useful each feature is for prediction.
- **Overfitting:** When a model is too closely fitted to the training data; Random Forest helps reduce this.
- **Hyperparameters:** Settings like the number of trees (n_estimators), tree depth, and number of features considered at each split.

Merits of Random Forest:

- High accuracy and robustness.
- Works well for both classification and regression.
- Handles missing values and maintains accuracy.
- Reduces overfitting through ensemble learning.
- Can measure feature importance.

EXAMPLE

```
# Load required package
library(randomForest)
# Load the iris dataset
data(iris)
# Set seed for reproducibility
set.seed(123)
# Create a Random Forest model
rf_model <- randomForest(Species ~ ., data = iris, ntree = 100)
# Print the model summary
print(rf_model)
# Predict using the model
predictions <- predict(rf_model, iris)
# Show first few predictions
head(predictions)
# Check accuracy using confusion matrix
confusion_matrix <- table(iris$Species, predictions)
```

```
print(confusion_matrix)
# Plot the error rate of Random Forest
plot(rf_model)
```

OUTPUT

```
randomForest(formula = Species ~ ., data = iris, ntree = 100)
               Type of random forest: classification
                     Number of trees: 100
No. of variables tried at each split: 2

        OOB estimate of  error rate: 4%
Confusion matrix:
            setosa versicolor virginica
setosa         50         0         0
versicolor      0        48         2
virginica       0         4        46
```

4.1.2 DECISION TREES

A **Decision Tree** is a supervised machine learning algorithm used for both classification and regression tasks. It splits the dataset into subsets based on the value of input features, and this process is represented in a tree-like structure with **decision nodes** and **leaf nodes**.

Why We Use Decision Trees:

- To **predict categorical** or **continuous outcomes**.
- Easy to **interpret and visualize**.
- Can handle **both numerical and categorical data**.

Steps:

1 **Choose the Best Attribute:** Select the attribute that best splits the data (e.g., using Gini index or Information Gain).

2 **Split the Dataset:** Divide the dataset based on the chosen attribute.

3 **Repeat Recursively:** Repeat the process for each child node until:

- All data points belong to the same class.
- There are no remaining attributes.
- A stopping condition is met (like max depth).

4 **Build the Tree:** Form decision nodes and leaf nodes.

5 **Make Predictions:** Traverse the tree from root to leaf for new data points.

Terminologies:

Term	Definition
Root Node	The first node that starts the tree and represents the entire dataset.
Decision Node	A node that splits into two or more branches based on a feature condition.
Leaf Node	The terminal node that gives the final output (class/target value).
Splitting	Dividing the dataset into subsets based on a feature.
Pruning	Removing unnecessary nodes to reduce overfitting.
Gini Index	A metric to measure impurity for classification tasks.
Information Gain	Measures how much "information" a feature gives about the class.
Entropy	Measures randomness or disorder in data.

Example Program:

```
# Load required packages
library(rpart)
library(rpart.plot)
# Load dataset
data(iris)
# Build the Decision Tree model
dt_model <- rpart(Species ~ ., data = iris, method = "class")
# Print model summary
print(dt_model)
# Plot the Decision Tree
rpart.plot(dt_model)
```

```
# Make predictions
predictions <- predict(dt_model, iris, type = "class")
# Display first 6 predictions
head(predictions)
# Create a confusion matrix
confusion_matrix <- table(iris$Species, predictions)
print(confusion_matrix)
```

Output:

```
n= 150
node), split, n, loss, yval, (yprob)
      * denotes terminal node
1) root 150 100 setosa (0.33 0.33 0.33)
  2) Petal.Length< 2.45 50   0 setosa (1.00 0.00 0.00) *
  3) Petal.Length>=2.45 100  50 versicolor (0.00 0.50 0.50)
    6) Petal.Width< 1.75 54   5 versicolor (0.00 0.91 0.09) *
    7) Petal.Width>=1.75 46   1 virginica (0.00 0.02 0.98) *
```

Predictions Example:

```
[1] setosa setosa setosa setosa setosa setosa
Levels: setosa versicolor virginica
```

Confusion Matrix Example:

```
           setosa versicolor virginica
setosa         50         0         0
versicolor      0        49         1
virginica       0         2        48
```

The plot will display a **decision tree** where:

- The **first split** is on Petal.Length < 2.45.
- Then further splits are based on Petal.Width < 1.75.
- Leaf nodes will indicate the final species classification.

4.1.3 NORMAL DISTRIBUTIONS AND BINOMIAL DISTRIBUTIONS

Normal Distribution:The **Normal Distribution** (also called the Gaussian distribution) is a continuous probability distribution that is **symmetrical** around its mean. It is characterized by the **bell-shaped curve**.

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Where:

- μ = Mean, σ = Standard deviation, e = Euler's number

R Example Code:

```
# Generate random numbers from a normal distribution
data <- rnorm(1000, mean = 50, sd = 10)
# Plot histogram
hist(data, main = "Normal Distribution", col = "lightblue", breaks = 30)
# Add a density curve
lines(density(data), col = "red", lwd = 2)
```

Binomial Distribution:

The **Binomial Distribution** is a **discrete probability distribution** of the number of successes in a sequence of **n independent experiments**, each asking a yes/no question (success/failure).

$$P(X = k) = \binom{n}{k} p^k (1-p)^{n-k}$$

Where:

- n = Number of trials, k = Number of successes, p = Probability of success

Key Features:

- Fixed number of trials.

- Each trial is independent.
- Two possible outcomes: success or failure.

R Example Code:

```
# Parameters
n <- 10      # number of trials
p <- 0.5     # probability of success
# Generate binomial data
data <- rbinom(1000, size = n, prob = p)
# Plot the distribution
barplot(table(data), main = "Binomial Distribution", col = "lightgreen",
        xlab = "Number of Successes", ylab = "Frequency")
```

Differences:

Feature	Normal Distribution	Binomial Distribution
Type	Continuous	Discrete
Shape	Bell-shaped	Varies (can be symmetric or skewed)
Outcomes	Infinite possibilities	Finite number of outcomes
Examples	Heights, IQ scores	Tossing a coin, Pass/fail tests

4.1.4 TIME SERIES ANALYSIS

Time Series Analysis in R is used to study how data behaves over time. Using the ts() function, we can create time series objects by linking data points with timestamps. This is especially useful in business and research for analyzing trends and forecasting future behavior—like sales, stock prices, or COVID-19 case counts.

Syntax:

```
objectName <- ts(data, start, end, frequency)
```

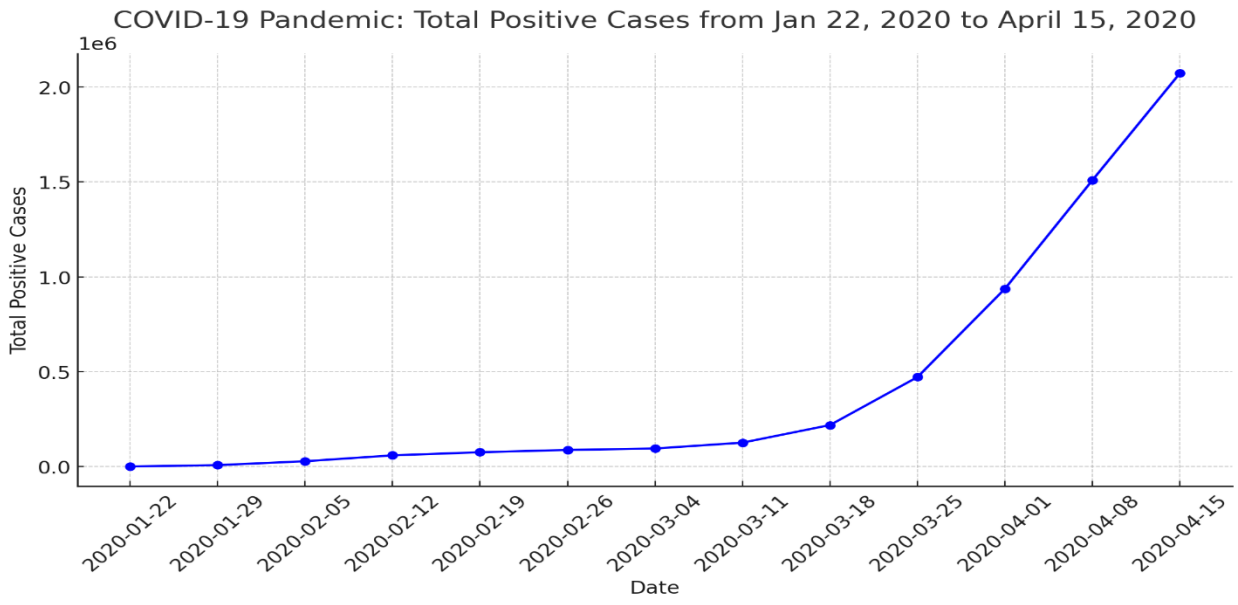
Where:

- **data** = vector of data,
- **start/end** = start and end times,
- **frequency** = observations per time unit.

Example: Weekly COVID-19 data was plotted and forecasted using:

- lubridate for date handling,
- forecast library for ARIMA models.

```
# Weekly data of COVID-19 positive cases from
# 22 January, 2020 to 15 April, 2020
x <- c(580, 7813, 28266, 59287, 75700,
       87820, 95314, 126214, 218843, 471497,
       936851, 1508725, 2072113)
# library required for decimal_date() function
library(lubridate)
# output to be created as png file
png(file = "timeSeries.png")
# creating time series object
# from date 22 January, 2020
mts <- ts(x, start = decimal_date(ymd("2020-01-22")),
          frequency = 365.25 / 7)
# plotting the graph
plot(mts, xlab = "Weekly Data",
     ylab = "Total Positive Cases",
     main = "COVID-19 Pandemic",
     col.main = "darkgreen")
# saving the file
dev.off()
```



4.1.5 SIMPLE LINEAR REGRESSION AND MULTIPLE LINEAR REGRESSION

Simple Linear Regression is the **most basic form** of linear regression. It models the relationship between **two variables:**

- **Independent Variable (X):** The predictor or input variable.
- **Dependent Variable (Y):** The response or output variable.

The relationship is represented by a **straight line** (the regression line), following this equation

Where:

$$Y = a + bX + \epsilon$$

- **Y** = Dependent variable (output)
- **X** = Independent variable (input)
- **a** = Intercept (the value of Y when X = 0)
- **b** = Slope of the line (how much Y changes when X increases by 1 unit)
- **ε (epsilon)** = Error term (captures the difference between actual and predicted Y)

Example: Predicting a person's **height (Y)** based on their **age (X).**

2 Multiple Linear Regression:

Multiple Linear Regression is an **extension** of Simple Linear Regression. It models the relationship between:

- **One dependent variable (Y)**
- **Two or more independent variables (X1, X2, X3, ..., Xn)**

The formula becomes: $$Y = a + b_1X_1 + b_2X_2 + b_3X_3 + \cdots + b_nX_n + \epsilon$$

Where:

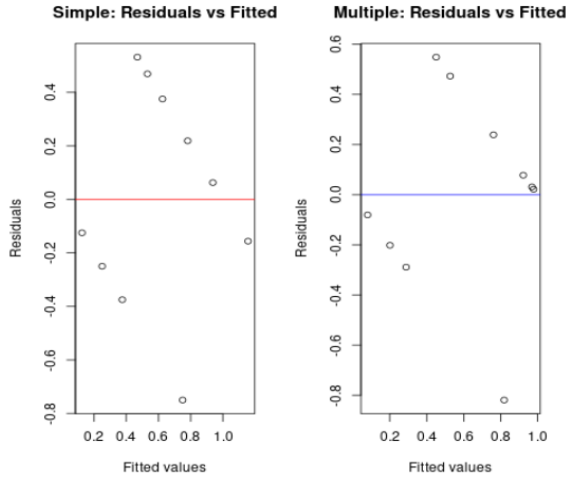
- **X_1, X_2, ..., Xn** = Multiple independent variables
- **b_1, b_2, ..., bn** = Coefficients (showing how each independent variable affects Y)

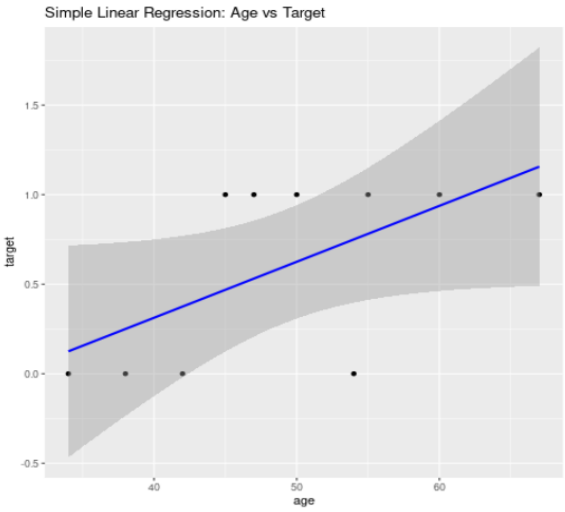
- **Example:** Predicting a person's **weight (Y)** based on their **height (X_1), age (X_2), and gender (X_3)**

Linear and Multiple Regression Analysis

```
# Step 1: Load data
data <- data.frame(
  age = c(45, 54, 67, 34, 50, 60, 42, 55, 38, 47),
  chol = c(230, 250, 240, 210, 245, 255, 220, 260, 215, 235),
  trestbps = c(140, 150, 160, 120, 130, 170, 138, 142, 125, 145),
  target = c(1, 0, 1, 0, 1, 1, 0, 1, 0, 1)
)
# Step 2: Check assumptions
print(sum(is.na(data)))  # Check missing values
# Step 3a: Simple linear regression (target ~ age)
model_simple <- lm(target ~ age, data = data)
summary(model_simple)
# Step 3b: Multiple linear regression (target ~ age + chol + trestbps)
model_multi <- lm(target ~ age + chol + trestbps, data = data)
summary(model_multi)
# Step 4: Check for homoscedasticity (plot residuals)
par(mfrow = c(1, 2))
plot(model_simple$fitted.values, model_simple$residuals,
     main = "Simple: Residuals vs Fitted",
     xlab = "Fitted values", ylab = "Residuals")
abline(h = 0, col = "red")
plot(model_multi$fitted.values, model_multi$residuals,
     main = "Multiple: Residuals vs Fitted",
     xlab = "Fitted values", ylab = "Residuals")
abline(h = 0, col = "blue")
# Step 5: Visualize the simple linear regression
library(ggplot2)
ggplot(data, aes(x = age, y = target)) +
  geom_point() +
  geom_smooth(method = "lm", col = "blue") +
  ggtitle("Simple Linear Regression: Age vs Target")
# Step 6: Print results
cat("Simple Linear Regression:\n")
print(coef(model_simple))
cat("\nMultiple Linear Regression:\n")
print(coef(model_multi))
```

```
Simple Linear Regression:
 (Intercept)           age
  -0.4019027    0.0235868

Multiple Linear Regression:
 (Intercept)           age          chol     trestbps
 -0.91329982  0.02245432  0.00088963  0.00325678
```





4.1.6 LOGISTIC REGRESSION

Logistic regression (also known as **binomial logistic regression**) is a classification algorithm used to estimate the probability of an event's success or failure. It is particularly applicable when the **dependent variable is binary** in nature (e.g., 0/1, True/False, Yes/No). At the core of logistic regression is the **logistic (or sigmoid) function**, which transforms any real-valued input into a value between 0 and 1, making it interpretable as a probability. This allows the model to describe the relationship between input features and the probability of a binary outcome. Logistic regression is a type of **Generalized Linear Model (GLM)** designed for classification tasks, especially when the response variable is binary. The main objective is to model the probability that a given input belongs to a specific category. The output probability always lies between 0 and 1.

The logistic regression model can be expressed as:

$$P = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_n x_n)}}$$

Where:

- P = probability of success,
- β_0= intercept,
- $\beta_1,\beta_2,...,\beta_n$ = coefficients for predictors $x_1,x_2,...,x_n$

Interpretation: Odds and Odds Ratio (OR)

In logistic regression:

- **Odds** represent the ratio of the probability of success to the probability of failure.
- The **odds ratio (OR)** is crucial for interpreting logistic regression coefficients. It shows how the odds change with a one-unit increase in a predictor.

$$\text{Odds} = \frac{P}{1-P}$$

Odd Ratio

Key interpretations:

- An **OR of 1**: Equal probability of success and failure.
- An **OR of 2**: Success is twice as likely as failure.
- An **OR of 0.5**: Failure is twice as likely as success.

EXAMPLE

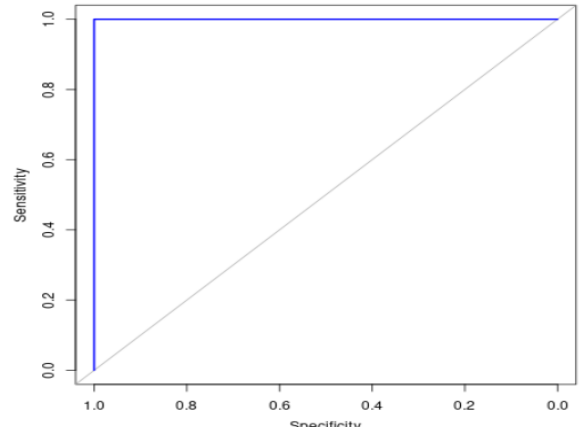
```
# Step 1: Load Data
data <- data.frame(
  age = c(29, 35, 45, 52, 61, 48, 50, 63, 39, 57),
  chol = c(210, 220, 240, 180, 300, 270, 250, 310, 200, 290),
  trestbps = c(120, 130, 140, 125, 150, 135, 145, 160, 128, 155),
  heart_disease = c(0, 0, 1, 1, 1, 0, 1, 1, 0, 1)
)
# Step 2: Check data
str(data)
summary(data)
# Step 3: Logistic Regression Model
```

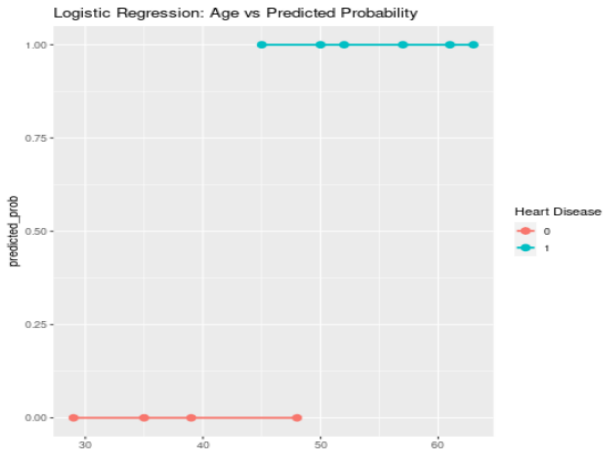
```
model <- glm(heart_disease ~ age + chol + trestbps, data = data, family =
"binomial")
summary(model)
# Step 4: Check for multicollinearity
library(car)
vif(model)
# Step 5: Predicted probabilities
data$predicted_prob <- predict(model, type = "response")
print(data)
# Step 6: Confusion Matrix at 0.5 threshold
predicted_class <- ifelse(data$predicted_prob > 0.5, 1, 0)
table(Predicted = predicted_class, Actual = data$heart_disease)
# Step 7: ROC Curve and AUC
library(pROC)
roc_curve <- roc(data$heart_disease, data$predicted_prob)
plot(roc_curve, col = "blue")
auc(roc_curve)
# Step 8: Visualization of predicted probabilities
library(ggplot2)
ggplot(data, aes(x = age, y = predicted_prob, color = as.factor(heart_disease))) +
  geom_point(size = 3) +
  geom_smooth(method = "glm", method.args = list(family = "binomial"), se =
FALSE) +
  labs(title = "Logistic Regression: Age vs Predicted Probability",
       color = "Heart Disease")
```

OUTPUT

```
'data.frame': 10 obs. of  4 variables:
 $ age          : num  29 35 45 52 61 48 50 63 39 57
 $ chol         : num  210 220 240 180 300 270 250 310 200 290
 $ trestbps     : num  120 130 140 125 150 135 145 160 128 155
 $ heart_disease : num  0 0 1 1 1 0 1 1 0 1

      age            chol          trestbps     heart_disease
 Min.   :29.00   Min.   :180.0   Min.   :120.0   Min.   :0.0
 1st Qu.:42.00   1st Qu.:215.0   1st Qu.:128.2   1st Qu.:0.0
 Median :48.50   Median :245.0   Median :137.5   Median :1.0
 Mean   :47.90   Mean   :247.0   Mean   :138.0   Mean   :0.6
 Max.   :63.00   Max.   :310.0   Max.   :160.0   Max.   :1.0
```

```
Call:
glm(formula = heart_disease ~ age + chol + trestbps, family = "binomial
    data = data)

Coefficients:
             Estimate Std. Error z value Pr(>|z|)
(Intercept)  -18.3212    10.4820  -1.747   0.0807 .
age            0.3012     0.1487   2.025   0.0430 *
chol           0.0195     0.0153   1.276   0.2021
trestbps       0.0156     0.0642   0.243   0.8080

(Dispersion parameter for binomial family taken to be 1)

Null deviance: 13.862  on 9  degrees of freedom
Residual deviance:  6.278  on 6  degrees of freedom
AIC: 14.278
```





Aspect	Simple Linear Regression	Multiple Linear Regression	Logistic Regression
Purpose	Predicts a **continuous** dependent variable using 1 predictor.	Predicts a **continuous** dependent variable using 2+ predictors.	Predicts a **binary** outcome (classification: 0/1, Yes/No).
Dependent Variable Type	Continuous (numeric)	Continuous (numeric)	Categorical (binary: 0/1)
Independent Variables	One (X)	Two or more (X1, X2, X3, ...)	One or more (X1, X2, X3, ...)

Equation	$Y = \beta_0 + \beta_1 X + \varepsilon$	$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + ... + \beta_n X_n + \varepsilon$	$\log(p/(1-p)) = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + ... + \beta_n X_n$
Prediction Output	A number (e.g., house price, height)	A number (e.g., sales, blood pressure)	A probability between 0 and 1 (e.g., chance of disease)
Example Use Case	Predicting weight based on height.	Predicting blood pressure based on age, weight, and diet.	Predicting disease presence (Yes/No) based on symptoms.
Error Metric	Mean Squared Error (MSE), R^2	Mean Squared Error (MSE), R^2	Accuracy, Precision, Recall, ROC AUC
Key Assumption	Linear relationship between X and Y.	Linear relationship between X's and Y.	Log-odds (logit) is linearly related to predictors.
Graph Type	Straight line (2D scatter + line)	Plane/multidimensional hyperplane	S-curve (sigmoid curve)

4.1.7 SURVIVAL ANALYSIS

Survival Analysis is a set of statistical methods used to analyze the time until an **event of interest** occurs. It's widely used in medicine, biology, engineering, and social sciences.

Key feature:

It deals with **"time-to-event"** data, such as:

- Time to death,

- Time to relapse,
- Time to machine failure.

Key Concepts:

Concept	Description
Survival Time	The time from a defined starting point to the occurrence of a given event.
Event/Censoring	- Event: The outcome of interest (e.g., death, failure).- Censoring: When the event hasn't happened by the end of the study or is lost to follow-up.
Survival Function S(t)	Probability of surviving beyond time t: S(t) = P(T > t).
Hazard Function h(t)	The instantaneous rate at which the event occurs at time t, given survival up to time t.

Popular Methods:

1 **Kaplan-Meier Estimator:**

- Estimates the survival function **S(t)**.
- Provides a **step plot** showing survival probability over time.

2 **Log-Rank Test:**

- Compares the survival distributions of two or more groups.

3 **Cox Proportional Hazards Model:**

- A regression model that relates predictors (age, treatment, etc.) to survival time.
- Assumes hazard ratios are constant over time.

Example Use Cases:

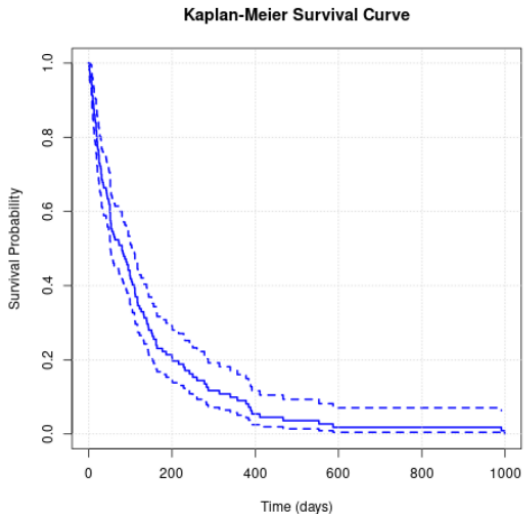
- Time until cancer relapse after treatment.
- Time to failure of mechanical components.
- Duration of unemployment in labor studies.

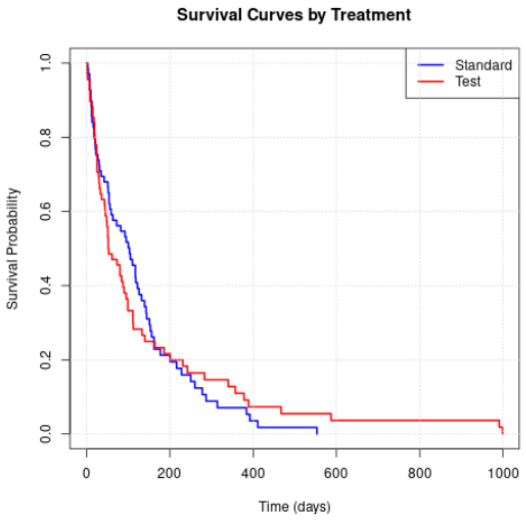
PROGRAM

```
# Install the survival package if needed
# install.packages("survival")
library(survival)
# Load the veteran dataset
data("veteran")
# View first few rows of data
print("First few rows of the veteran dataset:")
print(head(veteran))
# Check the structure of the dataset
print("Structure of the veteran dataset:")
print(str(veteran))
# Kaplan-Meier survival curve
print("Kaplan-Meier survival curve summary:")
km_fit <- survfit(Surv(time, status) ~ 1, data = veteran)
print(summary(km_fit))
# Plot the Kaplan-Meier survival curve
plot(km_fit, xlab = "Time (days)", ylab = "Survival Probability", main = "Kaplan-
Meier Survival Curve", col = "blue", lwd = 2)
grid()
# Kaplan-Meier curve by treatment group
print("Kaplan-Meier survival curve by treatment group:")
km_fit_trt <- survfit(Surv(time, status) ~ trt, data = veteran)
print(summary(km_fit_trt))
# Plot by treatment group
plot(km_fit_trt, col = c("blue", "red"), lwd = 2, xlab = "Time (days)", ylab =
"Survival Probability", main = "Survival Curves by Treatment")
legend("topright", legend = c("Standard", "Test"), col = c("blue", "red"), lwd = 2)
grid()
# Cox proportional hazards model
print("Cox proportional hazards model summary:")
cox_fit <- coxph(Surv(time, status) ~ trt + age + celltype + karno + diagtime +
prior, data = veteran)
print(summary(cox_fit))
```

```
[1] "Kaplan-Meier survival curve summary:"
Call: survfit(formula = Surv(time, status) ~ 1, data = veteran)

   time n.risk n.event survival std.err lower 95% CI upper 95% CI
1     1    137       1    0.993 0.00707        0.9798        1.000
2     2    136       2    0.978 0.01117        0.9562        1.000
3     3    134       1    0.971 0.01275        0.9463        0.997
...
```

```
[1] "Cox proportional hazards model summary:"
Call:
coxph(formula = Surv(time, status) ~ trt + age + celltype + karno +
    diagtime + prior, data = veteran)

  n= 137, number of events= 128

                          coef  exp(coef)   se(coef)      z Pr(>|z|)
trt                   2.946e-01  1.343e+00  2.075e-01  1.419  0.15577
age                  -8.706e-03  9.913e-01  9.300e-03 -0.936  0.34920
celltypesmallcell     8.616e-01  2.367e+00  2.753e-01  3.130  0.00175 **
celltypeadeno         1.196e+00  3.307e+00  3.009e-01  3.975 7.05e-05 ***
celltypelarge         4.013e-01  1.494e+00  2.827e-01  1.420  0.15574
karno                -3.282e-02  9.677e-01  5.508e-03 -5.958 2.55e-09 ***
diagtime              8.132e-05  1.000e+00  9.136e-03  0.009  0.99290
prior                 7.159e-03  1.007e+00  2.323e-02  0.308  0.75794
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

                     exp(coef) exp(-coef) lower .95 upper .95
trt                     1.3426     0.7448    0.8939    2.0166
age                     0.9913     1.0087    0.9734    1.0096
celltypesmallcell       2.3669     0.4225    1.3799    4.0597
celltypeadeno           3.3071     0.3024    1.8336    5.9647
celltypelarge           1.4938     0.6695    0.8583    2.5996
karno                   0.9677     1.0334    0.9573    0.9782
diagtime                1.0001     0.9999    0.9823    1.0182
prior                   1.0072     0.9929    0.9624    1.0541

Concordance= 0.736  (se = 0.021 )
Likelihood ratio test= 62.1  on 8 df,   p=2e-10
Wald test            = 62.37  on 8 df,   p=2e-10
Score (logrank) test = 66.74  on 8 df,   p=2e-11
```





4.2 LET US SUM UP

Random Forest and Decision Tree are machine learning algorithms used for classification and regression, where Random Forest builds multiple trees for better accuracy, while Decision Tree uses a single tree structure. Normal Distribution is a bell-

shaped curve showing data spread, and Binomial Distribution models success/failure outcomes. Time Series Analysis examines data over time to identify trends and make forecasts. Linear Regression predicts a continuous outcome using one predictor, and Multiple Regression uses multiple predictors. Logistic Regression is used for binary classification problems. Survival Analysis studies the time until an event occurs, such as failure or death.

4.3 Check Your Progress

1. What type of algorithm is Random Forest?
   A) Clustering
   B) Regression
   C) Ensemble Learning
   D) Dimensionality Reduction
2. Which algorithm works by creating a model that predicts the value of a target variable by learning simple decision rules?
   A) Random Forest
   B) K-Means
   C) Decision Tree
   D) Naive Bayes
3. What is the mean of a standard normal distribution?
   A) 1
   B) 0
   C) -1
   D) 100
4. What is the shape of a normal distribution curve?
   A) Skewed
   B) Bell-shaped
   C) Flat
   D) U-shaped

5. In a binomial distribution, trials are:
   A) Dependent
   B) Independent
   C) Always equal
   D) Skewed
6. Which R function is used for linear regression?
   A) lm()
   B) glm()
   C) regress()
   D) linreg()
7. Which R function is used for logistic regression?
   A) lm()
   B) glm()
   C) logit()
   D) regress()
8. What type of variable is predicted by logistic regression?
   A) Continuous
   B) Binary
   C) Ordinal
   D) Interval
9. What is the key assumption of linear regression?
   A) Non-linearity
   B) Homoscedasticity
   C) Multicollinearity
   D) Skewness
10. Which plot is useful to check residuals in linear regression?
    A) Histogram
    B) Scatterplot
    C) Residual vs Fitted plot
    D) Line chart

11. What is the main difference between linear and multiple regression?
    A) Number of dependent variables
    B) Number of independent variables
    C) Type of dataset
    D) Type of regression line
12. Which of the following is true for Random Forest?
    A) Uses a single tree
    B) Reduces overfitting
    C) Works only for classification
    D) Always underfits
13. What does ACF stand for in time series analysis?
    A) AutoCorrelation Function
    B) Average Curve Function
    C) Auto Classification Factor
    D) Average Cross Function
14. What is the main goal of time series analysis?
    A) Classification
    B) Forecasting
    C) Clustering
    D) Dimensionality Reduction
15. What is the hazard function in survival analysis?
    A) Probability of survival
    B) Rate of event occurrence
    C) Mean time to event
    D) Time series data
16. What does the Kaplan-Meier curve show?
    A) Correlation
    B) Regression line
    C) Survival probability
    D) Time trend

17. In Random Forest, what is the default method for choosing features at each split?
    A) Random
    B) Fixed
    C) All features
    D) Only categorical
18. In decision trees, what is Gini impurity used for?
    A) To measure data quality
    B) To split the data
    C) To calculate accuracy
    D) To prune the tree
19. Which distribution is used in logistic regression?
    A) Poisson
    B) Normal
    C) Binomial
    D) Uniform
20. Which method is best for handling non-linear data?
    A) Linear regression
    B) Polynomial regression
    C) Logistic regression
    D) Cox regression
21. What is the log-rank test used for?
    A) Comparing mean values
    B) Comparing two survival curves
    C) Testing for normality
    D) Testing homoscedasticity
22. In time series, what is stationarity?
    A) Trendless data
    B) Data with constant mean and variance
    C) Data with high correlation
    D) Random data

23. Which metric is used to evaluate logistic regression?
    A) R-squared
    B) Accuracy
    C) Residual sum of squares
    D) F-statistic
24. In linear regression, multicollinearity occurs when:
    A) Independent variables are independent
    B) Independent variables are highly correlated
    C) Dependent variables are binary
    D) Residuals are homoscedastic
25. What is the output of logistic regression?
    A) Real numbers
    B) Probability values (0–1)
    C) Categorical labels
    D) Counts
26. Which term describes overfitting prevention in decision trees?
    A) Pruning
    B) Splitting
    C) Gini index
    D) Bootstrapping
27. Which model is suitable for censored data?
    A) Linear regression
    B) Logistic regression
    C) Cox proportional hazards model
    D) Random Forest
28. What is a key difference between Random Forest and Decision Tree?
    A) Decision tree uses bagging
    B) Random Forest is a single tree
    C) Random Forest combines multiple trees
    D) Decision tree is more accurate

29. What is ARIMA used for?
    A) Classification
    B) Clustering
    C) Time series forecasting
    D) Regression

4.4 UNIT SUMMARY

This unit covers key statistical and machine learning concepts used in data analysis and predictive modeling. It introduces essential algorithms such as Random Forest and Decision Trees, explaining their roles in classification and regression tasks. The unit explores fundamental statistical distributions like Normal and Binomial, crucial for understanding data behavior. Techniques such as Linear and Multiple Regression are discussed for predicting continuous outcomes, while Logistic Regression is highlighted for binary classification problems. Additionally, Time Series Analysis methods, including ARIMA, are introduced for forecasting, and Survival Analysis is explained for modeling time-to-event data using tools like the Kaplan-Meier estimator and Cox proportional hazards model. Throughout, the focus is on understanding, applying, and interpreting these methods effectively.

4.5 GLOSSARY

- **Random Forest:** An ensemble learning method that builds multiple decision trees and merges their results for better accuracy and robustness.
- **Decision Tree:** A flowchart-like model that splits data into branches to make decisions based on input features.
- **Normal Distribution:** A symmetric, bell-shaped distribution representing data that clusters around a mean.
- **Binomial Distribution:** A discrete probability distribution of the number of successes in a fixed number of independent experiments.
- **Time Series Analysis:** A method of analyzing data points collected or recorded at specific time intervals to identify trends and make forecasts.

- **Linear Regression:** A statistical method that models the relationship between a dependent variable and one independent variable using a straight line.
- **Multiple Regression:** An extension of linear regression that models the relationship between a dependent variable and two or more independent variables.
- **Logistic Regression:** A classification algorithm used when the dependent variable is binary, predicting the probability of a particular outcome.
- **Survival Analysis:** A statistical approach used to analyze the expected duration until one or more events happen, like failure or death.
- **Kaplan-Meier Estimator:** A non-parametric statistic used to estimate the survival function from lifetime data.
- **Cox Proportional Hazards Model:** A regression model used in survival analysis to examine the effect of several variables on survival time.

4.6 SELF – ASSESSMENT QUESTIONS

1. What is the main goal of a Random Forest algorithm?
2. How does a Decision Tree decide where to split the data?
3. What is the shape of a Normal distribution?
4. Give an example of a situation modeled by a Binomial distribution.
5. What type of data is used in Time Series Analysis?
6. What is the dependent variable in Linear Regression?
7. How does Multiple Regression differ from Simple Linear Regression?
8. In Logistic Regression, what is the typical range of output probabilities?
9. What is the main purpose of Survival Analysis?
10. What is censored data in the context of Survival Analysis?
11. What is an ensemble method in machine learning?
12. What metric is commonly used to evaluate a Decision Tree’s performance?
13. Why is the mean important in a Normal distribution?
14. What does the term “event” mean in a Binomial distribution?
15. Name one common method to decompose a Time Series.
16. What does the slope represent in Linear Regression?
17. What does the R-squared value indicate in regression models?

18. Why do we use the sigmoid function in Logistic Regression?
19. What is a hazard ratio in Survival Analysis?
20. What is multicollinearity in Multiple Regression?
21. How does pruning help in Decision Trees?
22. What is the Central Limit Theorem?
23. How can seasonality affect Time Series data?
24. What is the difference between overfitting and underfitting?
25. What does a confusion matrix show in classification?
26. What role does the intercept play in Linear Regression?
27. In Random Forest, why do we use multiple trees?
28. What is the proportional hazards assumption in Cox models?
29. What is the output of a Logistic Regression model?
30. Give one application of Survival Analysis in healthcare.

4.7 ACTIVITIES / EXERCISES / CASE STUDIES

Activities:

1. **Build a Decision Tree:**
   Use the rpart package in R to build a decision tree model using a sample dataset (e.g., the iris dataset).
2. **Implement a Random Forest:**
   Apply the randomForest package to classify heart disease data and evaluate the accuracy.
3. **Plot a Normal Distribution:**
   Generate 1000 random numbers from a normal distribution and plot the histogram with a density curve.
4. **Simulate a Binomial Experiment:**
   Simulate tossing a coin 100 times and plot the distribution of successes.
5. **Time Series Visualization:**
   Use the AirPassengers dataset to visualize monthly airline passenger data and identify trends and seasonality.

6. **Simple Linear Regression:**
   Build a simple regression model to predict weight based on height using a sample dataset.
7. **Multiple Regression Model:**
   Fit a multiple regression model to predict house prices using factors like size, location, and number of bedrooms.
8. **Logistic Regression Analysis:**
   Classify whether a tumor is malignant or benign using logistic regression on a cancer dataset.
9. **Survival Analysis with Kaplan-Meier:**
   Perform Kaplan-Meier survival analysis using the survival package with the veteran dataset.
10. **Check for Multicollinearity:**
    Use the car package's VIF function to detect multicollinearity in a multiple regression model.

Exercises:

1. Explain the difference between Random Forest and Decision Trees.
2. List the assumptions of Linear Regression.
3. What is the purpose of the confusion matrix?
4. Describe the shape of a Normal distribution curve.
5. How can logistic regression be used in marketing?
6. Interpret an odds ratio of 0.7 in logistic regression.
7. Describe one method to handle missing values in Time Series data.
8. What is meant by censored data in Survival Analysis?
9. Explain the significance of the hazard function.
10. Discuss one real-life application of Random Forest in healthcare.

Case Studies:

1. **Heart Disease Prediction:**
   Use a heart disease dataset to compare the performance of Decision Tree and Random Forest classifiers. Analyze feature importance and accuracy.
2. **Housing Price Prediction:**
   Perform Multiple Linear Regression on housing data to predict prices. Evaluate the model and suggest improvements.
3. **Cancer Diagnosis:**
   Apply Logistic Regression to breast cancer data to classify tumors. Analyze the ROC curve and AUC.
4. **Patient Survival Time:**
   Conduct a Survival Analysis of lung cancer patients using the veteran dataset. Plot the survival curves and interpret the results.
5. **Retail Sales Forecasting:**
   Analyze sales data of a retail store using Time Series Analysis to forecast future sales. Identify trends, seasonality, and cyclic patterns.

4.8 Answers For Check Your Progress

1. C) Ensemble Learning
2. C) Decision Tree
3. B) 0
4. B) Bell-shaped
5. B) Independent
6. A) lm()
7. B) glm()
8. B) Binary
9. B) Homoscedasticity
10. C) Residual vs Fitted plot
11. B) Number of independent variables
12. B) Reduces overfitting
13. A) AutoCorrelation Function

14. B) Forecasting
15. B) Rate of event occurrence
16. C) Survival probability
17. A) Random
18. B) To split the data
19. C) Binomial
20. B) Polynomial regression
21. B) Comparing two survival curves
22. B) Data with constant mean and variance
23. B) Accuracy
24. B) Independent variables are highly correlated
25. B) Probability values (0–1)
26. A) Pruning
27. C) Cox proportional hazards model
28. C) Random Forest combines multiple trees
29. C) Time series forecasting

4.9 REFERENCES

1. **Breiman, L. (2001).** Random Forests. *Machine Learning*, 45(1), 5–32.
2. **Rice, J. A. (2006).** *Mathematical Statistics and Data Analysis*. Cengage Learning.
3. **CRAN Survival Package Documentation** – https://cran.r-project.org/web/packages/survival/
4. **R for Data Science** by Hadley Wickham & Garrett Grolemund A comprehensive guide to importing, tidying, transforming, and visualizing data in R. https://r4ds.hadley.nz/
5. **The R Book** by Michael J. Crawley An extensive reference covering data input methods and statistical modeling in R. https://onlinelibrary.wiley.com/doi/book/10.1002/9781118448908

UNIT V

Creating data for analytics through designed experiments, Creating data for analytics through active learning, Creating data for analytics through reinforcement learning

UNIT OBJECTIVE

The objective of this unit is to provide a comprehensive understanding of how data for analytics can be created through various advanced techniques such as designed experiments, active learning, and reinforcement learning. Through designed experiments, learners will explore how systematic experimentation can generate controlled and meaningful data for analysis. Active learning will be explored as a strategy for selecting the most informative data points to maximize learning efficiency while minimizing labeling costs. Reinforcement learning, on the other hand, will be examined for its ability to generate data by allowing an agent to interact with its environment, learn from trial and error, and optimize its behavior to gather relevant data. By the end of the unit, learners will grasp the theoretical foundations and practical applications of these techniques to create robust datasets, essential for building and training high-performing models in various domains.

SECTION 5.1: SEARCH TREES

5.1.1 CREATING DATA ANALYTICS THROUGH DESIGNED EXPERIMENTS

Creating data for analytics through designed experiments involves carefully planning and structuring your data collection process to ensure that the results are reliable, meaningful, and can be used to draw valid conclusions. Below is a step-by-step outline of how to create data for analytics using designed experiments:

1. Define the Objective

- Clearly define the goal of your experiment. What do you want to learn or prove through the experiment? For example, in the context of medical imaging or machine learning, the objective could be to evaluate the performance of a denoising technique or compare different segmentation algorithms.

2. Identify Variables

- **Independent Variables**: These are the factors that you manipulate during the experiment. For example, in a medical imaging experiment, the independent variables might include the type of denoising filter (Gaussian, Wiener, etc.) or the segmentation technique used.
- **Dependent Variables**: These are the outcomes you measure, such as accuracy, PSNR, SSIM, or Dice similarity index in a segmentation task.
- **Control Variables**: These are variables that are kept constant throughout the experiment to ensure that the changes in the dependent variables are due to the independent variables and not other factors.

3. Select the Experimental Design

- **Completely Randomized Design (CRD)**: Randomly assign treatments (independent variables) to subjects (data points). This is a simple and flexible design, suitable for experiments with relatively few variables.
- **Factorial Design**: This design involves multiple factors (independent variables) tested simultaneously, allowing for the analysis of interaction effects. This is ideal if you have multiple variables that may influence the outcome.
- **Split-Plot Design**: This is used when some factors are difficult to vary and others can be easily manipulated. It involves grouping experimental units into blocks based on one factor.
- **Randomized Block Design (RBD)**: Group the subjects into blocks that are similar to each other based on a certain characteristic, and then randomize the treatment assignment within each block.

4. Choose the Sample Size

- The sample size is crucial for the reliability of the experiment. You can use statistical power analysis to determine the minimum number of samples needed to detect an effect with a certain level of confidence. The larger the sample size, the more reliable the results, but it also increases costs and complexity.

5. Randomization

- Randomize the allocation of treatments or manipulations to the experimental units (data points) to eliminate bias and ensure that the results are generalizable.

6. Collect the Data

- **Data Collection Method**: Define how you will collect the data (e.g., from CT scans, medical records, or sensor data). Ensure that your data collection method aligns with the experimental design.
- **Consistency**: Make sure that the data collection process is consistent for all subjects or samples, ensuring that any differences in the results are due to the experimental manipulations rather than variations in how the data is collected.

7. Perform the Experiment

- Execute the experiment according to the design, ensuring that the manipulations are done correctly and that all variables are controlled as needed.

8. Data Analysis

- After collecting the data, use appropriate statistical or analytical methods to analyze the data. Techniques like ANOVA (Analysis of Variance), regression analysis, or machine learning models can be used, depending on the nature of the data and the objectives of the experiment.
- Analyze the effects of different independent variables on the dependent variables and assess any interactions between the variables.

9. Interpret Results

- Based on your analysis, draw conclusions about the effects of different treatments or factors. For example, if you're comparing denoising techniques for medical images, you might conclude which filter provides the best trade-off between image quality and computation time.

10. Validate and Verify

- Cross-check the results with previous studies or validate them through replication if possible. For example, validate the findings of your experiment on a separate test dataset to confirm their robustness.

5.1.2 CREATING DATA ANALYTICS THROUGH ACTIVE LEARNING

Creating Data for Analytics through Active Learning involves a process where the model iteratively selects the most informative or uncertain data points to be labeled by an oracle (typically a human annotator). This approach is especially useful when labeled data is scarce or expensive to obtain. The goal is to create a high-quality labeled dataset with fewer labeled examples by focusing on uncertain or complex cases.

Here's a detailed process for creating data for analytics through active learning:

1. Understand the Problem Context

- **Identify the Task**: Whether it's classification, regression, segmentation, or another task, active learning can be applied to any supervised learning problem. The first step is to define the task clearly.
- **Define the Data Requirements**: Understand what type of data is required. For example, in medical imaging, the data might include labeled CT scan images of patients with specific diseases (e.g., pneumonia, lung cancer).
- **Set Performance Metrics**: Decide which metrics (accuracy, precision, recall, F1-score, etc.) will define success and how they will be evaluated during the process.

2. Choose an Active Learning Strategy

Active learning involves the model selecting the most informative data points for labeling. Several strategies can be used:

- **Uncertainty Sampling**: The model queries the data points where it is most uncertain (i.e., those where the model's predictions have low confidence).

Techniques like entropy-based uncertainty or margin-based uncertainty can be used here.

- **Query by Committee**: Involves training multiple models (the committee) and querying instances where the models disagree the most. This approach assumes that disagreements between models represent areas where the model is uncertain.
- **Expected Model Change**: The model queries instances that would most change the current model if they were labeled. This approach focuses on data points that are expected to provide the most information about the decision boundary.
- **Representative Sampling**: Selects data points that are most representative of the entire dataset, which can help improve generalization.
- **Diversity Sampling**: Chooses diverse data points that cover different areas of the feature space. This ensures that the model is exposed to various scenarios, improving the overall learning.

3. Initial Model Training

- **Start with a Small Labeled Dataset**: Begin with a small, randomly selected labeled dataset. This could be a small set of images, text samples, or sensor data, depending on the task. This initial dataset can be manually labeled or taken from a partially labeled dataset.
- **Train a Preliminary Model**: Use the initial labeled dataset to train an initial model (classifier, regressor, etc.). This model will guide the selection of the most informative data points for the next round of labeling.

4. Data Selection and Querying

- **Apply Active Learning Strategy**: The trained model is used to identify the most uncertain or diverse data points from the unlabeled pool. These are the data points that the model “doesn’t understand well” or that are most likely to improve its performance if labeled.

- **Labeling**: The selected data points are sent to the oracle (human annotator or labeling service) for labeling. In the case of image segmentation, a domain expert would annotate the areas of interest in the images.
- **Batch Selection**: It's often efficient to request labeling in batches rather than one instance at a time. This can help reduce labeling costs and time.

5. Model Retraining

- **Incorporate Labeled Data**: Once the new data points have been labeled, they are added to the labeled dataset.
- **Retrain the Model**: The model is retrained using the expanded labeled dataset, which now includes the newly labeled instances. This process ensures that the model improves its performance with each round of active learning.
- **Evaluate Model Performance**: Use a separate validation set to evaluate how well the model is performing after each round of training.

6. Iterative Process

- **Repeat the Active Learning Cycle**: After retraining, the model queries the next set of uncertain or diverse data points, and the labeling process continues. This cycle is repeated iteratively until the model's performance reaches a satisfactory level or the labeling budget is exhausted.
- **Stop Criteria**: Define when to stop the active learning process. This can be when:
    - Performance metrics reach a satisfactory threshold.
    - The model shows diminishing returns from additional labeled data.
    - The labeling budget (time, cost) is exhausted.

7. Exploit Expert Knowledge

- **Expert Feedback**: In tasks like medical imaging or complex decision-making, you may involve domain experts who provide more informative labels, especially for ambiguous or difficult cases.

- **Active Learning with Domain Expertise**: Integrating expert knowledge can enhance the process of selecting uncertain or complex examples for labeling. In medical imaging, a radiologist's expertise can guide the model to focus on particularly difficult areas of scans (e.g., identifying rare types of lung lesions).

8. Data Augmentation

- **Data Augmentation**: Use data augmentation techniques (e.g., rotation, flipping, scaling for images) to artificially expand the dataset. This can be especially useful in image or signal processing tasks where variations in the data can be generated synthetically, increasing diversity without requiring additional labeling.
- **Synthetic Data**: Another option is to generate synthetic data using generative models like GANs (Generative Adversarial Networks) or simulations to further augment the dataset.

9. Balancing the Dataset

- **Address Data Imbalance**: If the data contains imbalances between classes (e.g., more normal images than pneumonia images), active learning can help focus on underrepresented or rare cases. Active learning strategies like uncertainty sampling can help ensure that the model sees these underrepresented cases.
- **Cost-sensitive Active Learning**: This is a variant of active learning that takes into account the cost of labeling different types of data. For example, some cases may require more expert effort to label (e.g., complex medical diagnoses) and may be prioritized for labeling.

10. Performance Monitoring and Adjustment

- **Monitor Model Performance**: During the active learning process, it's important to continuously monitor model performance. If the model performance is stagnating or the queries are not improving accuracy, consider adjusting the active learning strategy or exploring alternative data labeling approaches.

- **Adapting Active Learning Strategy**: Depending on how the model is performing, you might need to adjust the strategy. For example, if the model starts to show low uncertainty, you may switch to a more diversity-focused strategy or use a different querying method.

11. Handling Noisy Labels

- **Label Quality Control**: When using human annotators, it's important to ensure the quality of the labels. Implementing a double-check system (having two annotators label the same data point) or using consensus approaches can help in cases where labeling errors may be introduced.
- **Active Learning with Noisy Labels**: If noisy labels are unavoidable, techniques like noise filtering or robust learning algorithms can be incorporated to reduce the negative impact of inaccurate labels on model training.

12. Cost Efficiency

- **Budget Management**: Active learning helps reduce labeling costs, but it's important to consider the cost-efficiency of the process. You can assess the cost-benefit ratio of labeling, evaluating whether the improvement in model performance justifies the expense of labeling additional data.

13. Automated Active Learning Tools

- There are several open-source libraries and tools for implementing active learning efficiently, such as:
    - **ALiPy**: An active learning library for Python that provides various querying strategies.
    - **ModAL**: A Python library for modular active learning with built-in support for uncertainty sampling and query-by-committee methods.
    - **scikit-activeml**: A library built on top of scikit-learn for active learning tasks in Python.

Benefits of Active Learning

- **Cost Reduction**: Reduces the need for a large number of labeled data points, thus saving time and costs associated with data labeling.
- **Improved Model Performance**: By focusing on uncertain or complex cases, active learning ensures that the model learns the most valuable information for better performance.
- **Efficient Data Usage**: Helps maximize the utility of available labeled data and improves the model's generalization with fewer labeled instances.

Challenges of Active Learning

- **Initial Model Performance**: The model must be reasonably well-optimized to identify uncertain instances in the early stages, which may require a basic understanding of the task even before starting the active learning process.
- **Labeling Costs**: While active learning reduces the number of labeled instances needed, each label still requires time and expertise, especially in domains like medical imaging.

By applying active learning, you can create a high-quality dataset for analytics, ensuring that the model is trained on the most informative and valuable examples, ultimately improving the efficiency and performance of the analytics process.

5.1.3 CREATING DATA ANALYTICS THROUGH REINFORCEMENT LEARNING

Creating Data for Analytics through Reinforcement Learning involves leveraging the principles of reinforcement learning (RL) to generate and improve data, typically for decision-making tasks, optimization problems, or systems where an agent learns through interaction with an environment. In the context of data creation, RL can be used to generate data in a variety of ways, often focusing on exploration, action-reward feedback loops, and optimizing decision-making in complex, dynamic environments.

Here’s a detailed approach for creating data for analytics through reinforcement learning:

1. Understanding Reinforcement Learning (RL)

Reinforcement Learning is a type of machine learning where an agent learns to make decisions by interacting with an environment, receiving feedback in the form of rewards or penalties. The key components of RL are:

- **Agent**: The learner or decision maker.
- **Environment**: The external system or context with which the agent interacts.
- **State**: The current situation or configuration of the environment.
- **Action**: The choices made by the agent that influence the environment.
- **Reward**: The feedback the agent receives after taking an action in a particular state.
- **Policy**: The strategy or mapping from states to actions that the agent follows to maximize cumulative reward.

2. Defining the Problem

- **Task Setup**: The first step is to define the task or problem that needs to be solved using reinforcement learning. This could be anything from optimizing a business process, robotic control, or even generating synthetic data in a simulation environment.
- **Environment Design**: Create or select an environment in which the agent will learn. This could be a physical or simulated environment (like a game, manufacturing process, or healthcare simulation) where the agent interacts with and gathers data.
- **State and Action Space Definition**: Define the state space (set of all possible states) and the action space (set of possible actions the agent can take). This forms the foundation of the RL setup.

3. Using RL to Generate Data

In the creating data, reinforcement learning can help by generating new data points based on exploration and interaction with the environment. This can be useful for tasks such as:

- **Simulated Data Generation**: In some cases, real-world data may be limited or expensive to obtain. An RL agent can interact with a simulated environment and generate synthetic data based on the agent's actions and the subsequent feedback.
- **Data Augmentation**: RL can be used to create variations of existing data. For example, if you're working on a task like image segmentation or classification, RL agents can explore different ways to alter or augment the original data (e.g., rotating images, modifying pixel values) based on their learning to increase the variety of data available for training.
- **Optimizing Data Collection**: In environments where data collection is expensive or slow (e.g., sensor readings or experimental data), RL can be used to decide when and how to collect data. The agent can learn an optimal policy to minimize the cost or time involved in gathering data while maximizing the diversity and usefulness of the dataset.

4. Environment Interaction and Data Feedback Loop

- **Exploration vs. Exploitation**: One of the central concepts in RL is the balance between exploration (trying new actions to learn more about the environment) and exploitation (taking actions known to yield high rewards). This exploration-exploitation trade-off can be used to generate diverse and informative data for analytics.
- **Reward Structure**: Design the reward system so that the agent is incentivized to gather useful, diverse, or high-quality data. For example, the agent might receive higher rewards for generating novel data points or data that covers less-explored areas of the data space.
- **Data Generation through Interaction**: As the agent interacts with the environment, it collects states, actions, and rewards, which can form the basis of a rich dataset. In RL, each interaction (state-action-reward) is a data point, and multiple interactions lead to the creation of a large dataset that reflects the dynamics of the environment.

5. Exploration Strategies for Data Creation

RL agents can use several exploration strategies to gather diverse and informative data:

- **Epsilon-Greedy**: The agent takes random actions with probability ε (exploration) and takes the best-known action with probability 1 - ε (exploitation). This helps to balance exploration and exploitation, generating a variety of data while still improving the model's performance.
- **Boltzmann Exploration**: Actions are selected probabilistically based on their expected reward. Actions with higher rewards are chosen more often, but less rewarding actions are still occasionally explored, leading to diverse data generation.
- **Thompson Sampling**: A Bayesian approach where the agent samples actions based on their posterior probability, which encourages the exploration of actions with uncertain outcomes, leading to diverse data points.

6. Data Augmentation via Policy Exploration

- **Data Transformation**: By using RL in environments like image processing or text generation, the agent can explore transformations of data that might be underrepresented in the original dataset. For instance, in image data, RL agents might discover new ways to augment images (e.g., through different lighting, background changes, or geometric distortions).
- **Policy Exploration for Data Generation**: In tasks like natural language processing (NLP), RL agents can explore different linguistic structures or word sequences, generating diverse sentence structures that can be used for training models.

7. Leveraging RL for Synthetic Data Generation

RL is increasingly being used in generating synthetic data, particularly in scenarios where real data is hard to come by or is limited. Examples include:

- **Game Simulations**: In gaming or virtual environments, an RL agent can explore and generate synthetic data by playing the game and interacting with its components. This synthetic data can be used to train models for tasks like game testing, game AI, or strategy optimization.
- **Robotics**: In robotics, RL can be used to generate data for training robotic models. For instance, a robot can explore different configurations or motions in a simulated environment to generate training data for tasks like object manipulation, navigation, or control.
- **Healthcare**: In medical imaging or healthcare, RL can be used to generate synthetic patient data or medical scenarios based on learned models of disease progression or diagnostic tasks.

8. Data Creation through Reward Shaping

- **Shaping Rewards**: By adjusting the reward function, RL can be guided to focus on specific types of data. For example, in medical data generation, the agent could be rewarded more for generating data that is difficult for the model to interpret, which could provide useful information for improving diagnostics or analysis.
- **Custom Reward Functions**: Customizing the reward function can influence the type of data generated. In RL for analytics, the reward structure could prioritize high-quality, diverse, or edge-case data that is important for training robust models.

9. Evaluation of the Generated Data

- **Testing Model Performance**: After generating data, it is essential to evaluate how well the models trained on this data perform. This can be done by testing on validation sets and comparing performance metrics such as accuracy, precision, recall, F1-score, and others.
- **Quality Control**: Evaluate the quality of generated data by assessing its usefulness for analytics. For instance, is the synthetic data representative of real-world scenarios? Does it help the model generalize well on unseen data?

10. Applications of Data Creation through RL

Reinforcement learning for data generation is particularly useful in several applications:

- **Robotic Data Generation**: In robotics, RL agents can create varied data by performing different tasks in simulated environments, contributing to training data for robotic control models.
- **Synthetic Data for Simulations**: RL is often used to create synthetic data in scenarios such as financial modeling, autonomous driving, and healthcare.
- **Personalized Recommendations**: RL can generate personalized data for recommendation systems by simulating interactions between users and content, allowing the system to learn from diverse user behavior.

11. Challenges

- **Exploration vs. Exploitation in Data Creation**: While exploration generates new data, it may lead to redundant or less informative data points, which might need to be carefully managed.
- **Computational Complexity**: RL often requires substantial computational resources for training, especially in complex environments or with large state and action spaces.
- **Reward Function Design**: Defining an appropriate reward function that accurately reflects the utility of the generated data can be challenging, especially in tasks where the benefits of certain data points are not immediately obvious.

Reinforcement learning offers a unique approach to generating data for analytics, particularly in situations where traditional data collection methods may be costly, time-consuming, or impractical. By using RL to interact with dynamic environments and reward exploration, an agent can create high-quality, diverse datasets for a wide range of tasks, from simulation and robotics to healthcare and personalized recommendations. Through careful design of the environment, reward structures, and exploration strategies, RL can

be a powerful tool for improving the quality and quantity of training data used in machine learning and analytics.

Let Us Sum Up

In this unit, we learned about three approaches for creating data for analytics: **designed experiments**, **active learning**, and **reinforcement learning**. Designed experiments involve systematically manipulating variables in controlled environments to gather reliable data for analysis. Active learning allows models to selectively query the most informative data points for labeling, reducing the need for large labeled datasets. Reinforcement learning generates data by enabling an agent to interact with its environment, learn from feedback, and optimize its actions over time. Each technique plays a vital role in generating high-quality data for training models and improving decision-making in various fields.

Check Your Progress

1. Which of the following experimental designs helps in testing multiple factors simultaneously in a controlled way?
   A) Factorial Design
   B) Randomized Block Design
   C) Latin Square Design
   D) Complete Block Design
2. What is the main goal of **active learning**?
   A) To generate large datasets from unlabeled data
   B) To reduce the cost of data labeling by selecting the most informative data points
   C) To create synthetic data through exploration
   D) To learn from feedback in a trial-and-error fashion
3. Which of the following is NOT a key component of reinforcement learning (RL)?
   A) Agent
   B) Environment

C) Data Labeling

D) Reward

4. **Response Surface Methodology** is primarily used in:

   A) Data classification

   B) Optimization of processes

   C) Active learning

   D) Dimensionality reduction

5. In **active learning**, which technique selects data points for labeling where the model is most uncertain?

   A) Uncertainty Sampling

   B) Random Sampling

   C) Representative Sampling

   D) Query Synthesis

6. What is the role of the **oracle** in active learning?

   A) To generate new synthetic data

   B) To label the data points selected by the model

   C) To optimize the model's performance

   D) To test the final model on unseen data

7. In designed experiments, **randomization** is important because it:

   A) Ensures that all factors are controlled

   B) Reduces bias by randomly assigning treatments

   C) Maximizes the number of factors involved

   D) Guarantees an optimal outcome

8. Which type of design is used in experiments to group units based on a specific characteristic (e.g., age) to control variability?

   A) Randomized Block Design

   B) Factorial Design

   C) Latin Square Design

   D) Full Factorial Design

9. **Reinforcement Learning** is best suited for:

   A) Supervised learning tasks with labeled data

B) Data labeling tasks in image classification

C) Decision-making problems where agents interact with an environment

D) Clustering tasks with no feedback

10. What is **uncertainty sampling** in active learning?

A) Selecting data points at random

B) Selecting the least labeled data points

C) Selecting data points with the highest model uncertainty

D) Selecting data points based on a predefined criterion

11. In a **reinforcement learning** setup, what does the agent receive after performing an action in a given state?

A) Data labels

B) A reward or penalty

C) Unlabeled data

D) A new environment state

12. The **factorial design** in experiments typically helps to:

A) Test the effects of one factor at a time

B) Evaluate the interactions between multiple factors

C) Minimize the variability in data collection

D) Increase the number of control groups

13. In **active learning**, which of the following methods generates new, synthetic data points through query formulation?

A) Uncertainty Sampling

B) Query Synthesis

C) Data Augmentation

D) Random Sampling

14. **Reinforcement learning** can be used for:

A) Solving classification problems with labeled data

B) Generating synthetic data through trial and error

C) Performing clustering on unlabeled datasets

D) Mapping input-output relationships for supervised learning

15. In **designed experiments**, which of the following methods aims to optimize the effect of multiple variables at different levels?
    A) Full Factorial Design
    B) Response Surface Methodology
    C) Randomized Block Design
    D) Latin Square Design

5.4 UNIT SUMMARY

Creating data through designed experiments is a systematic approach to collecting data for analysis by controlling variables in an experiment. It involves setting up controlled conditions to test hypotheses and make informed decisions based on empirical evidence. Key methods include factorial designs, response surface methodology, and randomized block designs. Designed experiments allow for a structured and efficient way of gathering data that is critical for analysis, especially in fields like manufacturing, healthcare, and market research.

5.5 GLOSSARY:

- **Active Learning**: A machine learning technique where the model selects the most informative data points to be labeled, reducing the cost of labeling and improving model performance.
- **Agent (Reinforcement Learning)**: An entity that interacts with the environment, takes actions, and learns from the feedback (rewards or penalties) received.
- **Reinforcement Learning (RL)**: A type of machine learning where an agent learns to make decisions by interacting with its environment to maximize cumulative rewards.
- **Feedback (Reinforcement Learning)**: The reward or penalty that an agent receives after performing an action, used to adjust future behavior.
- **Reward (Reinforcement Learning)**: A scalar value that is given to the agent after it performs an action, which the agent uses to evaluate its actions and learn from them.

- **Synthetic Data**: Data that is artificially generated instead of being collected from real-world sources, often used to supplement real data in cases where data is scarce or privacy concerns arise.

5.6 SELF – ASSESSMENT QUESTIONS

1. What is the primary objective of using designed experiments in data analytics?
2. How does active learning reduce the cost of labeling data, and what are its key benefits?
3. What role does uncertainty sampling play in active learning, and how does it help improve the model's performance?
4. In reinforcement learning, explain the interaction between the agent, environment, and feedback loop.
5. Describe how a factorial design helps in evaluating the interactions between multiple variables in an experiment.
6. What is the purpose of randomization in experimental designs, and how does it help in controlling bias?
7. What is the difference between exploration and exploitation in reinforcement learning, and how do they influence decision-making?
8. How does synthetic data contribute to machine learning tasks, and in what scenarios is it used?
9. Explain the significance of the oracle in active learning and its function in the data labeling process.
10. How can reinforcement learning be applied in real-world applications, and what are some examples where it is used effectively?
11. What is the key difference between active learning and traditional machine learning in terms of data labeling?
12. How do agents in reinforcement learning optimize their actions, and what does it mean to maximize cumulative rewards?
13. In a randomized block design, why is it important to group experimental units based on specific characteristics?

14. How does response surface methodology help in optimizing a process, and what types of experiments benefit from its use?
15. What is the purpose of using factorial designs in experiments, and how does it help in determining the effects of multiple variables simultaneously?

5.6 ACTIVITIES / EXERCISES / CASE STUDIES

Activity 1: Designing an Experiment for Data Collection

- **Objective**: To design a controlled experiment to understand the effects of two variables on a specific outcome.
- **Instructions**:
    1. Select two variables (e.g., temperature and pressure) that you believe will influence a particular outcome (e.g., reaction time in a chemical process).
    2. Use a **factorial design** to test the effects of these variables. Vary the levels of each variable (e.g., high, low) and measure the effect on the outcome.
    3. Document your experiment's setup, including the number of trials, the randomization process, and how you plan to control for other variables.
    4. After completing the experiment, analyze the results and determine the main effects and interactions between the variables.

Exercise 1: Active Learning Implementation

- **Objective**: To demonstrate the process of active learning by selecting informative samples for labeling.
- **Instructions**:
    1. You are given a dataset with a large number of unlabeled images. Initially, you have a small set of labeled images.
    2. Use an **uncertainty sampling** strategy to choose the most uncertain data points for labeling (e.g., those with the highest model uncertainty).

3. Label the selected data points and train your model with the newly labeled data.
4. Repeat the process iteratively by selecting new uncertain samples and retraining the model until you achieve satisfactory performance with fewer labeled data.

Activity 2: Applying Reinforcement Learning to Optimize a Simple Game

- **Objective**: To understand how reinforcement learning can optimize decisions based on rewards.
- **Instructions**:
    1. Set up a simple game, such as Tic-Tac-Toe or a grid-based movement game, where an agent (player) can take different actions (e.g., move left, right, up, down).
    2. Define a reward system (e.g., +1 for winning, -1 for losing, and 0 for a draw).
    3. Program a **reinforcement learning agent** to play the game by exploring different actions, learning from the rewards it receives, and gradually optimizing its strategy.
    4. After training, observe how the agent's strategy evolves to maximize the cumulative rewards over time.

Case Study 1: Applying Active Learning in Medical Imaging

- **Objective**: To demonstrate how active learning can be applied to a medical imaging task (e.g., tumor detection in CT scans).
- **Instructions**:
    1. You are tasked with developing a model to detect tumors in CT scans. You have access to a large set of unlabeled images and a small labeled set.
    2. Use **active learning** to iteratively label the most uncertain images. Start by training the model with the initial labeled set.

3. Apply an **uncertainty sampling** strategy to select the most informative CT scans for labeling. Retrain the model on the updated dataset after each iteration.
4. Measure the performance of the model after each round of labeling and training, comparing it to a baseline model trained with the full labeled dataset.

Exercise 2: Evaluating Factorial Design in an Experiment

- **Objective**: To evaluate the use of **factorial design** in understanding the interactions between different variables.
- **Instructions**:
    1. Choose a process or product (e.g., baking time and temperature for cookies) and identify the variables that could impact the outcome (e.g., taste, texture).
    2. Design a **factorial experiment** with at least two variables and two levels for each (e.g., baking temperature: low, high; baking time: short, long).
    3. Record the results of each trial and analyze the effects of each variable, both individually and in combination.
    4. Use statistical software to analyze the results and determine the significant effects and any interactions between the variables.

Activity 3: Synthetic Data Generation for Model Training

- **Objective**: To generate synthetic data for a machine learning model when real data is scarce.
- **Instructions**:
    1. Choose a dataset where real data is limited or privacy concerns prevent its use.
    2. Use a **data generation technique**, such as **GANs (Generative Adversarial Networks)**, to generate synthetic data that mimics the real dataset's properties.

3. Train a machine learning model using both synthetic and real data and compare its performance to a model trained solely on real data.
4. Assess whether synthetic data can effectively supplement the real data to improve model performance.

Case Study 2: Reinforcement Learning in Robotics

- **Objective**: To explore the application of reinforcement learning in robot motion planning.
- **Instructions**:
    1. Imagine you're tasked with programming a robot to navigate through a maze.
    2. Define a **reward system**: +10 for reaching the goal, -1 for hitting an obstacle, and -5 for moving in circles.
    3. Use a **reinforcement learning algorithm** (e.g., Q-learning) to enable the robot to learn optimal movement strategies by trial and error.
    4. Observe the robot's learning process, and analyze how the agent balances exploration and exploitation to improve its pathfinding.

5.8 Answers For Check Your Progress

1. A) Factorial Design
2. B) To reduce the cost of data labeling by selecting the most informative data points
3. C) Data Labeling
4. B) Optimization of processes
5. A) Uncertainty Sampling
6. B) To label the data points selected by the model
7. B) Reduces bias by randomly assigning treatments
8. A) Randomized Block Design
9. C) Decision-making problems where agents interact with an environment
10. C) Selecting data points with the highest model uncertainty

11. B) A reward or penalty
12. B) Evaluate the interactions between multiple factors
13. B) Query Synthesis
14. B) Generating synthetic data through trial and error
15. B) Response Surface Methodology.

5.9 REFERENCES

1. https://www.studocu.com/in/document/i-k-gujral-punjab-technical-university/data-analytics-using-r/unit5-r-creating-data-for-analytics-through-designed-experiments-creating-data-for/94598100
2. https://www.studocu.com/in/quiz/unit5-r-creating-data-for-analytics-through-designed-experiments-creating-data-for/5105161
3. https://encord.com/blog/active-learning-machine-learning-guide

CDOE – OLP M.B.A – SEMESTER III DATA ANALYTICS WITH R PROGRAMMING