R4DataScience

A Course offered by Department of Statistics, University of Calicut

Mahesh Divakaran

2026-01-14

Table of contents

1 Introduction

1.1 R for Data Analysis: From Fundamentals to Advanced Applications

A Course Offered by the Department of Statistics, University of Calicut.

1.2 About This Book

This book, *R for Data Analysis: From Fundamentals to Advanced Applications*, has been developed by Mahesh Divakaran as part of an certificate course offered by the **Department of Statistics, University of Calicut**. It is designed to serve as a **comprehensive learning resource** for students who wish to build strong practical and theoretical skills in data analysis using the R programming language.

The material is written in the form of a **Quarto book**, enabling seamless integration of explanations, executable R code, figures, and outputs in a single reproducible document. This approach reflects modern statistical practice, where analysis, interpretation, and reporting are inseparable.

1.3 Motivation for the Course

In today's data-driven world, statistics is no longer confined to theory alone. Statisticians are expected to:

- Handle real-world datasets
- Perform exploratory and confirmatory data analysis
- Apply appropriate statistical models
- Communicate results clearly and reproducibly

R has emerged as one of the most powerful and widely used tools to meet these demands. This course is motivated by the need to **bridge the gap between statistical theory and practical data analysis**, enabling students to translate classroom concepts into real analytical workflows.

1.4 Why R for Data Analysis?

R is a language developed by statisticians, for statisticians. Its design philosophy aligns closely with statistical thinking and data-centric problem solving.

Key reasons for choosing R as the primary tool in this course include:

- A rich ecosystem for statistical methods and modeling
- Strong support for data visualization and exploratory analysis
- Open-source nature and global community support
- Integration of computation, graphics, and reporting
- Wide acceptance in academia, research, and industry

By learning R, students gain a **transferable skill** that is relevant across disciplines such as economics, health sciences, social sciences, environmental studies, and machine learning.

1.5 Course Philosophy and Learning Approach

This course adopts a **hands-on, example-driven learning approach**. Rather than treating R as a collection of commands to memorize, the emphasis is on:

- Understanding data structures and workflows
- Writing clear, readable, and reusable code
- Interpreting results in a statistical context
- Developing reproducible analysis habits

Each concept is introduced with intuition, followed by practical examples and gradually extended to more complex applications. Errors, warnings, and unexpected outputs are treated as learning opportunities rather than obstacles.

1.6 Intended Audience

This book is primarily intended for:

- Undergraduate and postgraduate students of Statistics
- Students from allied disciplines with a quantitative background
- Beginners with little or no prior programming experience

A basic understanding of statistics will be helpful, but **no prior knowledge of R or programming is assumed**. The material progresses from fundamental concepts to advanced analytical techniques in a structured manner.

1.7 Course Structure and Progression

The book is organized to support **progressive learning**, moving from foundational concepts to advanced applications.

Broadly, the course covers:

1. Introduction to R, RStudio, and the computing environment
2. Core data structures and data handling in R
3. Data import, cleaning, and transformation
4. Exploratory data analysis and visualization
5. Statistical modeling and inference
6. Advanced topics and applied case studies
7. Reproducible reporting using Quarto

Each chapter builds on previous knowledge, ensuring continuity and conceptual clarity.

1.8 Reproducible Data Analysis with Quarto

A defining feature of this course is the emphasis on **reproducible research**. All analyses in this book are written using Quarto, allowing:

- Code and results to coexist in one document
- Automatic generation of tables and figures
- Easy export to PDF, HTML, and Word formats

This practice encourages transparency, accuracy, and professional documentation—skills essential for academic research and industry projects alike.

1.9 Learning Outcomes

By the end of this course, students will be able to:

- Use R confidently for data analysis tasks
- Understand and manipulate different data structures
- Perform exploratory and inferential statistical analysis
- Visualize data effectively
- Write reproducible and well-documented analysis reports
- Apply R to real-world datasets and research problems

1.10 How to Use This Book

To get the most out of this book:

- Read the explanations carefully and run the code yourself
- Modify examples and observe how results change
- Practice regularly using the exercises provided
- Focus on understanding workflows rather than isolated commands

Active engagement is essential for mastering data analysis with R.

1.11 Concluding Remarks

R for Data Analysis: From Fundamentals to Advanced Applications is more than a programming guide—it is an introduction to **modern statistical practice**. Through this course, students will develop the skills needed to analyze data rigorously, think statistically, and communicate findings effectively using R.

The chapters that follow begin with the basics of R and RStudio, laying the foundation for advanced analytical techniques explored later in the book.

2 About

2.1 About the Author

Mahesh Divakaran is a Data Scientist, Statistical Programmer, Trainer, and Public Speaker with over **eight years of professional experience** in statistics, data science, and clinical research. He is the author of this course material *R for Data Analysis: From Fundamentals to Advanced Applications*, offered by the **Department of Statistics, University of Calicut**.

He specializes in **clinical data analysis using R**, supported by a strong foundation in statistical theory and applied analytics. He is currently pursuing a **Ph.D. in Statistics at Amity University**, with a research focus on advanced data analysis techniques.

Mahesh has worked extensively in the clinical research and healthcare analytics domain. He is currently a **Senior Statistical Programmer at IQVIA**, where he leads the development and validation of R packages for clinical studies. Previously, he served as **Associate Lead – Clinical Data Analytics at Genpro Research**, contributing to the development of R Shiny dashboards and automated reporting systems for clinical trials.

His technical expertise includes **R, Python, SAS, R Shiny, Tableau, and Power BI**, with particular strength in ADaM dataset creation, statistical reporting, and data visualization. Alongside industry work, he is deeply committed to teaching and capacity building in **Statistics, Data Science, and Clinical Data Analytics**, and has delivered invited talks and presentations at national and international conferences.

2.2 About the Department of Statistics, University of Calicut

The **Department of Statistics, University of Calicut**, was established in **1988** with the objective of developing into a **centre of excellence** in statistics and its applications in the region. The department was founded under the leadership of **Dr. K. Kumaranakutty**, the founder professor, and is rooted in a philosophy that emphasizes the close relationship between the University and the community, promoting a scientifically motivated society.

Since its inception, the department has been actively involved in **teaching, training, and research**, with a strong emphasis on modeling real-life problems using rigorous statistical methods. The faculty members engage in collaborative research with scientists and institutions within India and abroad. Several faculty members have been recipients of **Commonwealth Scholarships and Fellowships** tenable in the United Kingdom.

The department also plays a vital service role within the University by advising and assisting faculty members and research scholars from other departments in the **statistical analysis of research data**.

As of 2010, **27 research scholars** from the department have been awarded the **Ph.D. degree**. Research output from the department is nationally and internationally recognized, as reflected in peer-reviewed publications, funded projects, and invited lectures at academic conferences.

2.2.1 Thrust Areas

The major thrust areas of the department include:

- Stochastic Processes and Applications
- Reliability and Survival Analysis
- Extreme Value Theory
- Time Series Analysis and Modelling
- Applied Probability
- Distribution Theory
- Sampling Theory
- Novelty Detection and Neural Networks

2.2.2 Courses Offered

The department offers the following academic programmes:

1. **M.Sc. Statistics** – 4 Semesters (Choice Based Credit Semester System – CCSS)
2. **M.Phil.** – 2 Semesters
3. **Ph.D. Programme**

2.2.2.1 M.Sc. Statistics Programme

The M.Sc. Statistics programme is designed to provide **intensive training** covering the core and applied areas of statistics. The curriculum equips students with technical and analytical skills required for immediate employment as statisticians, while also preparing them for advanced research.

- **Current Specialization Module**: Advanced Statistical Analysis
- **Total Intake**: 25 students
- **Admission Criteria**: Candidates with a B.Sc. degree securing at least 50% marks, with either (i) Statistics as the main subject, or (ii) Mathematics as the main subject with Statistics as a subsidiary. Admission is based on performance in the common university entrance examination.

Graduates of the programme find employment opportunities in **research organizations, government agencies, industry, healthcare, analytics, and education**.

2.2.3 Faculty

- **Dr. K. Jayakumar** – Senior Professor
- **Dr. Krishnarani S. D.** – Associate Professor & Head
- **Dr. Dileepkumar M.** – Assistant Professor

Former Faculty Members:

- Dr. C. Chandran
- Dr. M. Manoharan
- Dr. N. Raju
- Dr. K. Kumaranakutty

2.2.4 Infrastructure and Facilities

The department is well-equipped with academic and research infrastructure. It has its own **departmental library**, housing over **2,100 books** and several volumes of classical statistical journals. The **computer laboratory** is equipped with more than **twenty personal computers** and statistical software such as **SPSS and SYSTAT**, along with teaching aids including OHP and LCD projectors. The department also maintains an active **Alumni Association**.

2.2.5 Research and Collaborations

Research in the department is held in high regard at national and international levels. Faculty members have received prestigious fellowships, including Commonwealth Scholarships and research fellowships in Canada. The department maintains active research collaborations with academic institutions, industry partners, and government organizations.

Major research interests include **Stochastic Processes, Reliability and Survival Analysis, Distribution Theory, Time Series Modelling, Applied Probability, Extreme Value Theory, Novelty Detection, and Neural Networks**.

Further details about the department and its academic programmes are available at **http://www.cuonline.ac.in/**.

3 Introduction to R and RStudio

3.1 What is R?

R is a**programming language and software environment**designed primarily for**statistical computing, data analysis, and visualization**. It is widely used in academia, research institutions, government organizations, and industry for tasks ranging from simple data summaries to advanced machine learning and reproducible research.

Key characteristics of R include:

- **Open-source and free**: Anyone can use, modify, and distribute R.
- **Strong statistical foundation**: Built by statisticians for statistical work.
- **Extensive package ecosystem**: Thousands of user-contributed packages extend R's capabilities.
- **Excellent graphics**: R is known for high-quality, publication-ready visualizations.
- **Reproducibility**: Scripts and documents ensure analyses can be repeated and verified.

In this workshop, R will be used as a**tool for thinking with data**, not just a calculator. You will learn how to write clear, reusable code that performs data analysis systematically.

3.2 Why Use R for Data Analysis?

R is especially well-suited for data analysis because it combines**data manipulation, statistical modeling, and visualization**in one environment.

3.2.1 Advantages of R

- Handles**small to very large datasets**efficiently
- Supports**classical statistics, modern data science, and machine learning**
- Encourages**script-based analysis**, reducing manual errors
- Integrates well with reports (PDF, Word, HTML) through**Quarto and R Markdown**
- Strong community support and documentation

3.2.2 Typical Use Cases

- Exploratory Data Analysis (EDA)
- Statistical inference and hypothesis testing
- Regression and multivariate analysis
- Time series and forecasting
- Data visualization and dashboards
- Academic research and thesis work

3.3 Installing R

R itself is the**core engine**that performs computations.

3.3.1 Steps to Install R

1. Visit the official Comprehensive R Archive Network (CRAN):https://cran.r-project.org
2. Choose your operating system (Windows / macOS / Linux)
3. Download and install the latest stable version

After installation, R can be run from the system console. However, working directly in the R console is not ideal for beginners or large projects. This is where**RStudio**becomes essential.

3.4 What is RStudio?

RStudio is an**Integrated Development Environment (IDE)**for R. It makes working with R easier, faster, and more organized.

RStudio does**not replace R**. Instead:

> R does the computation, and RStudio provides a user-friendly interface to work with R.

3.4.1 Why Use RStudio?

- Code editor with syntax highlighting
- Easy file and project management
- Built-in help and documentation
- Integrated plotting and visualization
- Seamless support for Quarto documents

3.5 Understanding the RStudio Interface

When you open RStudio, you typically see**four panes**:

3.5.1 1. Script Editor (Top-Left)

- Where you write and save R code
- Supports`.R`scripts and`.qmd`Quarto files
- Code is written once and executed many times

3.5.2 2. Console (Bottom-Left)

- Where R executes commands
- Displays output, messages, and errors
- Useful for quick calculations and testing code

3.5.3 3. Environment / History (Top-Right)

- Shows objects currently in memory
- Displays datasets, variables, and functions
- Helps track what data is loaded

3.5.4 4. Files / Plots / Help / Viewer (Bottom-Right)

- File browser for project files
- Plot display window
- Help documentation for functions
- Viewer for Quarto and HTML outputs

Understanding these panes is crucial for efficient workflow in R.

3.6 Your First Interaction with R

Let us start with very simple commands.

3.6.1 Using R as a Calculator

```
2 + 3
```

```
[1] 5
```

```
10 / 2
```

```
[1] 5
```

```
5^2
```

```
[1] 25
```

R immediately evaluates expressions and returns results.

3.6.2 Assigning Values to Objects

```
x <- 10
y <- 5
x + y
```

```
[1] 15
```

Here:

- `<-`is the**assignment operator**
- `x`and`y`are objects stored in memory

Object-based thinking is fundamental in R.

3.7 Basic Data Types in R

R works with different types of data. The most common ones are:

- **Numeric**: Numbers (e.g., 10, 3.5)
- **Character**: Text (e.g., "Statistics")
- **Logical**: TRUE or FALSE

Examples:

```
age <- 21
name <- "Anita"
passed <- TRUE
```

You can check the type of an object using:

```
class(age)
```

```
[1] "numeric"
```

```
class(name)
```

```
[1] "character"
```

```r
class(passed)
```

```
[1] "logical"
```

3.8 Vectors: The Core Data Structure

A vector is a**collection of values of the same type**.

```r
marks <- c(78, 85, 90, 88)
names <- c("A", "B", "C")
```

Key points:

- `c()`stands for*combine*
- All elements in a vector must be of the same type
- Vectors are the building blocks of most R data structures

3.9 Working Directory and Projects

The**working directory**is the default location where R reads and saves files.

```r
getwd()
setwd("path/to/your/folder")
```

However, changing directories manually is discouraged.

3.9.1 RStudio Projects (Recommended)

- Create a project for each workshop, assignment, or research work
- Keeps scripts, data, and outputs organized
- Ensures reproducibility across systems

3.10 Introduction to Quarto

Quarto is a**next-generation scientific and technical publishing system**that allows you to combine:

- R code
- Text explanation
- Tables and figures

into a**single, reproducible document**.

3.10.1 Why Quarto for This Workshop?

- Generates**PDF, HTML, and Word**outputs
- Ideal for teaching, reports, and books
- Supports executable R code chunks

Example of an R code chunk in Quarto:

summary(c(10, 20, 30, 40))

3.11 How This Book Is Structured

This Quarto book is designed for**beginners**and progresses gradually:

1. Basics of R and RStudio
2. Data structures and data import
3. Data manipulation and visualization
4. Statistical analysis
5. Reproducible reporting with Quarto

Each chapter includes:

- Clear explanations
- Annotated examples
- Practice exercises

3.12 Best Practices for Learning R

- Type the code yourself instead of copy–paste
- Read error messages carefully
- Experiment with small examples
- Save scripts regularly
- Ask*why*a result occurs, not just*what*it is

Learning R is not about memorizing commands, but about developing**data analysis thinking**.

3.13 Getting Help

R has extensive built-in help and a supportive community. Here are some ways to get help: - **Help files**: Use `?function_name` or `help(function_name)` to access documentation for any function. For example:

```
?mean
```

- **RStudio Help Pane**: Use the Help pane in RStudio to search for functions and topics.
- **Online Resources**: Websites like Stack Overflow, R-bloggers, and the RStudio Community are great places to ask questions and find solutions.
- **Books and Tutorials**: There are many free and paid resources available for learning R. Remember, seeking help is a normal part of learning programming!

4 R Environment, Syntax, and Operators

4.1 Introduction

Before performing any data analysis, it is essential to ensure that the R environment is correctly installed and functioning. This chapter introduces the basic diagnostics for the R environment, followed by a detailed discussion of R syntax, assignment operators, variable naming conventions, and arithmetic operations. These concepts form the foundation for writing clear and correct R programs.

4.2 R Environment Diagnostic Checks

A properly configured R environment ensures smooth execution of scripts and reproducible results.

4.2.1 Checking the R Version

```
R.version.string
```

```
[1] "R version 4.5.2 (2025-10-31)"
```

This command displays the installed R version. Using an up-to-date version of R is recommended for compatibility with modern packages.

4.2.2 Testing Package Installation

```
install.packages("ggplot2")
```

```
The following package(s) will be installed:
- ggplot2 [4.0.1]
These packages will be installed into "~/Documents/Personal/R/R4Data Science_CU/R4DataScience_C
4.5/aarch64-apple-darwin20".

# Installing packages --------------------------------------------------------
```

```
- Installing ggplot2 ...                        OK [linked from cache]
Successfully installed 1 package in 4.1 milliseconds.
```

Successful installation confirms that R can access CRAN repositories and install external packages.

4.2.3 Verifying Basic Computation

```
(50 + 50) / 2
```

```
[1] 50
```

This simple arithmetic check confirms that the R interpreter is functioning correctly.

4.2.4 Checking TinyTeX Installation

```
tinytex::is_tinytex()
```

```
[1] TRUE
```

TinyTeX is required for generating PDF outputs from Quarto documents. This command verifies its availability.

4.3 Assignment Operators in R

R provides multiple ways to assign values to objects.

```
x <- 5
y = 10
11 -> z
```

- `<-` is the **recommended assignment operator** in R
- `=` is commonly used but should be avoided in complex expressions
- `->` assigns values from right to left

4.3.1 Global Assignment Operator

```
z <<- 15
```

The`<<-`operator assigns a value to a variable in the global environment. Its use should be limited, as it can make code harder to debug.

4.4 Variable Naming Conventions

Valid variable names in R:

```
var1 <- 10
var_2 <- 20
var.3 <- 30
Var4 <- 40
varFive <- 50
```

Key rules:

- Variable names are**case-sensitive**
- They may contain letters, numbers, dots (`.`), and underscores (`_`)
- They must not begin with a number

Invalid variable names contain spaces or special characters and will produce errors.

4.5 Fundamental R Syntax

R syntax is flexible and generally ignores whitespace.

```
x <- 10
y <- 20
z <- x + y

a = 5; b = 15; c = a * b
d=x-y
e = x * y; f = x / y
```

Although whitespace is optional,**consistent formatting**improves readability and maintainability.

4.6 Basic Arithmetic Operations

R supports all standard arithmetic operations.

```
5 + 3      # Addition
```

```
[1] 8
```

```
10 - 4     # Subtraction
```

```
[1] 6
```

```
6 * 7      # Multiplication
```

```
[1] 42
```

```
20 / 5     # Division
```

```
[1] 4
```

```
2 ^ 3      # Exponentiation
```

```
[1] 8
```

```
10 %% 3    # Modulus
```

```
[1] 1
```

```
10 %/% 3   # Integer division
```

```
[1] 3
```

These operations are frequently used in data transformation and statistical computation.

4.7 Summary

In this chapter, you learned how to:

- Verify and diagnose the R environment
- Use different assignment operators
- Apply valid variable naming conventions
- Understand fundamental R syntax
- Perform arithmetic operations in R

These basics are essential for writing correct and efficient R programs. The next chapter explores**data types and type conversion**, which are crucial for understanding how R stores and processes data.

5 Data Types, Type Conversion, and Basic Functions

5.1 Introduction

Understanding data types is fundamental to effective data analysis in R. This chapter introduces the core data types supported by R, explains how R stores and interprets data, and demonstrates type conversion and commonly used built-in functions.

5.2 Numeric and Integer Data Types

5.2.1 Numeric

```
num_var <- 42.5
class(num_var)
typeof(num_var)
```

By default, numeric values in R are stored as **double-precision numbers**.

5.2.2 Integer

```
int_var <- 42L
class(int_var)
typeof(int_var)
```

Appending `L` explicitly creates an integer. Although integers and numerics behave similarly, they differ in memory representation.

5.3 Character Data Type

```
char_var <- "Hello, R!"
class(char_var)
typeof(char_var)
```

R supports both single and double quotes for character strings. Escape characters allow quotes to be included within strings.

5.4 Logical Data Type

```
log_var <- TRUE
log_var2 <- FALSE
```

Logical values are essential for conditional operations and filtering data. Shorthand forms `T` and `F` exist but are not recommended in professional code.

5.5 Complex and Raw Data Types

5.5.1 Complex

```
comp_var <- 3 + 4i
class(comp_var)
typeof(comp_var)
```

Complex numbers include an imaginary component and are mainly used in specialized mathematical computations.

5.5.2 Raw

```
raw_var <- charToRaw("Hello")
rawToChar(raw_var)
```

Raw data represents bytes and is rarely used in routine data analysis.

5.6 Type Conversion in R

R provides explicit functions for converting between data types.

```
as.character(123.45)
as.numeric("678.90")
as.integer(99.99)
as.logical(1)
```

Type conversion is common when importing data from external sources.

5.7 Date, Factor, and NULL

5.7.1 Date

```
date_var <- as.Date("2024-01-01")
class(date_var)
```

Dates are crucial for time-based analysis.

5.7.2 Factor

```
factor_var <- factor(c("Red", "Blue", "Green"))
levels(factor_var)
```

Factors are used to represent categorical variables and play an important role in statistical modeling.

5.7.3 NULL

```
null_var <- NULL
is.null(null_var)
```

`NULL` represents the absence of a value and is commonly used in programming logic.

5.8 Special Numeric Values

```
Inf
-Inf
NaN
```

These values represent infinity and undefined numerical results. Functions such as `is.nan()` and `is.infinite()` help identify them.

5.9 Basic Functions in R

Frequently used built-in functions include:

```
length(vec)
class(vec)
summary(vec)
str(vec)
mean(vec)
sum(vec)
sd(vec)
```

These functions are essential for exploratory data analysis.

5.10 Sequences and Repetition

```
seq(1, 10, by = 2)
rep(1:3, times = 2)
```

Sequences and repetition are commonly used for simulations and indexing.

5.11 Accessing Help and Documentation

```
?mean
help.search("regression")
vignette()
```

R provides extensive documentation that should be consulted regularly.

5.12 Working Directory Management

```
getwd()
list.files()
```

Managing the working directory ensures that data files and outputs are correctly located.

5.13 Summary

In this chapter, you learned about:

- Core data types in R
- Differences between `class()` and `typeof()`
- Explicit type conversion
- Special numeric values
- Essential built-in functions
- Help and file management utilities

These concepts are critical for data handling and analysis. The next chapter will focus on **data structures such as vectors, lists, and data frames**, which build upon the data types introduced here.

6 Vectors, Lists, Matrices, and Data Frames

6.1 Introduction

Data structures are the backbone of data analysis in R. While Chapter 5 introduced basic data types, this chapter focuses on **how data is organized and stored** using R's fundamental data structures: **vectors, lists, matrices, and data frames**. Understanding these structures is essential for efficient data manipulation, analysis, and modeling.

6.2 Vectors

A **vector** is the most basic data structure in R. It is a collection of elements of the **same data type**.

6.2.1 Creating Vectors

```
num_vec <- c(10, 20, 30, 40)
char_vec <- c("A", "B", "C")
log_vec <- c(TRUE, FALSE, TRUE)
```

The function `c()` stands for *combine*.

6.2.2 Properties of Vectors

- All elements must be of the **same type**
- Vectors are **one-dimensional**
- R performs **type coercion** if mixed types are used

```
mix_vec <- c(1, "A", TRUE)
class(mix_vec)
```

6.3 Accessing Vector Elements

Elements in a vector are accessed using **indexing**, which starts at 1 in R.

```
num_vec[1]
num_vec[2:4]
num_vec[c(1, 3)]
```

6.3.1 Logical Indexing

```
num_vec[num_vec > 20]
```

6.4 Vector Operations

Vectors support element-wise operations.

```
x <- c(1, 2, 3)
y <- c(4, 5, 6)

x + y
x * y
```

6.4.1 Recycling Rule

```
x <- c(1, 2, 3, 4)
y <- c(10, 20)
x + y
```

R recycles the shorter vector. A warning appears if lengths are incompatible.

6.5 Lists

A **list** is a flexible data structure that can contain elements of **different types and lengths**.

6.5.1 Creating Lists

```
my_list <- list(
  numbers = c(1, 2, 3),
  text = "Statistics",
  logical = TRUE,
  data = c("A", "B")
)
```

6.5.2 Accessing List Elements

```
my_list[[1]]
my_list$numbers
```

- `[[ ]]` extracts a single element
- `$` accesses elements by name

6.6 Matrices

A **matrix** is a two-dimensional data structure where all elements are of the **same type**.

6.6.1 Creating a Matrix

```
mat <- matrix(1:6, nrow = 2, ncol = 3)
mat
```

6.6.2 Matrix by Row

```
mat_row <- matrix(1:6, nrow = 2, byrow = TRUE)
mat_row
```

6.6.3 Accessing Matrix Elements

```
mat[1, 2]
mat[, 1]
mat[2, ]
```

6.7 Matrix Operations

Matrices support mathematical operations.

```
A <- matrix(1:4, nrow = 2)
B <- matrix(5:8, nrow = 2)

A + B
A * B        # Element-wise multiplication
A %*% B     # Matrix multiplication
```

6.8 Data Frames

A **data frame** is the most commonly used data structure in data analysis. It is a table-like structure where:

- Columns can have **different data types**
- All columns must have the **same length**

6.8.1 Creating a Data Frame

```
df <- data.frame(
  ID = 1:4,
  Name = c("A", "B", "C", "D"),
  Marks = c(78, 85, 90, 88),
  Passed = c(TRUE, TRUE, TRUE, FALSE)
)
```

6.9 Accessing Data Frame Elements

```
df$Marks
df[1, ]
df[, 2]
df[ , "Name"]
```

6.9.1 Structure of a Data Frame

```
str(df)
summary(df)
```

6.10 Adding and Removing Columns

```
df$Grade <- c("B", "B", "A", "B")
df$Passed <- NULL
```

6.11 Converting Between Data Structures

```
as.list(num_vec)
as.matrix(df)
as.data.frame(mat)
```

Understanding conversions helps avoid common errors in analysis.

6.12 Summary

In this chapter, you learned:

- How vectors store homogeneous data
- How lists store heterogeneous data
- How matrices represent two-dimensional numeric data
- How data frames organize real-world datasets
- Methods to access, modify, and convert data structures

These data structures form the foundation for data manipulation and analysis in R. In the next chapter, we will focus on **importing external data and preparing it for analysis**.

7 Data Import in R (CSV, Excel, SPSS, SAS)

7.1 Introduction

In real-world data analysis, data rarely originates inside R. Instead, datasets are typically stored in external files such as spreadsheets, text files, or statistical software formats. This chapter introduces the most common methods for **importing data into R**, with a focus on formats widely used in statistics, research, and industry: **CSV, Excel, SPSS, and SAS**.

The ability to correctly import data is a critical skill, as errors at this stage can propagate throughout the analysis.

7.2 Working Directory and File Paths

Before importing data, it is important to understand where R looks for files.

```
getwd()
```

The working directory is the default location from which R reads files and saves outputs. Using **RStudio Projects** is strongly recommended to manage file paths consistently.

Files can be referenced using:

- **Relative paths** (recommended)
- **Absolute paths** (system-specific, less portable)

7.3 Importing CSV Files

Comma-Separated Values (CSV) files are one of the most common data formats.

7.3.1 Using Base R

```r
data_csv <- read.csv("data/sample.csv")
head(data_csv)
```

Important arguments:

- `header = TRUE` – first row contains column names
- `stringsAsFactors = FALSE` – prevents automatic factor conversion

```r
data_csv <- read.csv("data/sample.csv", stringsAsFactors = FALSE)
```

7.3.2 Using `readr` Package (Recommended)

```r
library(readr)
data_csv <- read_csv("data/sample.csv")
```

Advantages of `readr`:

- Faster for large datasets
- Better handling of column types
- Clearer warnings and messages

7.4 Importing Excel Files

Excel files are widely used in academic and administrative settings.

7.4.1 Using `readxl`

```r
library(readxl)
data_excel <- read_excel("data/sample.xlsx")
```

To read a specific sheet:

```r
data_excel <- read_excel("data/sample.xlsx", sheet = "Sheet1")
```

To view available sheets:

```
excel_sheets("data/sample.xlsx")
```

Excel files do not require external dependencies, making `readxl` suitable for most systems.

7.5 Importing SPSS Files

SPSS files (`.sav`) are commonly used in social sciences and health research.

7.5.1 Using `haven`

```
library(haven)
data_spss <- read_sav("data/sample.sav")
```

SPSS value labels are preserved and imported as labelled variables.

```
str(data_spss)
```

To convert labelled variables to factors:

```
data_spss[] <- lapply(data_spss, as_factor)
```

7.6 Importing SAS Files

SAS datasets are widely used in clinical research and regulatory environments.

7.6.1 Using `haven`

```
data_sas <- read_sas("data/sample.sas7bdat")
```

SAS transport files (`.xpt`) can also be imported:

```
data_sas_xpt <- read_xpt("data/sample.xpt")
```

Metadata such as variable labels and formats are preserved during import.

7.7 Checking Imported Data

After importing data, it is essential to inspect its structure and content.

```
str(data_csv)
summary(data_csv)
head(data_csv)
```

Key checks include:

- Correct variable types
- Missing values
- Unexpected factor levels
- Column names and labels

7.8 Common Data Import Issues

Some common problems encountered during data import include:

- Incorrect file paths
- Encoding issues
- Automatic type conversion
- Missing or malformed values

Careful inspection and cleaning immediately after import is considered best practice.

7.9 Saving Imported Data in R Format

Once data is imported and cleaned, it can be saved in R's native formats.

```
save(data_csv, file = "data/sample.RData")
saveRDS(data_csv, "data/sample.rds")
```

R-specific formats load faster and preserve object structure.

7.10 Summary

In this chapter, you learned how to:

- Set and manage file paths
- Import CSV and Excel files
- Read SPSS and SAS datasets using `haven`
- Inspect and validate imported data
- Save datasets in R-native formats

Accurate data import is the foundation of reliable analysis. The next chapter will focus on **data cleaning and transformation**, preparing imported datasets for statistical analysis.

8 Introduction to the Tidyverse

8.1 Introduction

Modern data analysis in R is increasingly shaped by a collection of packages known as the **tidyverse**. The tidyverse is not a single package, but a **coherent ecosystem of R packages** designed for data science. These packages share a common philosophy, consistent syntax, and are built to work seamlessly together.

This chapter introduces the tidyverse conceptually and practically. It prepares the foundation for the next chapter on **data manipulation with `dplyr`**, ensuring that learners understand the broader framework before diving into individual tools.

8.2 What Is the Tidyverse?

The tidyverse is a set of R packages developed primarily by Hadley Wickham and the RStudio (Posit) team. These packages are designed around a shared approach to data analysis that emphasizes:

- Clear and readable code
- Consistent function naming and behavior
- A unified data structure (`tibble`)
- A grammar-based approach to data manipulation and visualization

Rather than learning many unrelated functions, the tidyverse encourages learning a **small number of core ideas** that apply across packages.

8.3 The Tidy Data Philosophy

At the heart of the tidyverse is the concept of **tidy data**. A dataset is considered tidy when:

- Each variable forms a column
- Each observation forms a row
- Each type of observational unit forms a table

This structure makes data easier to manipulate, visualize, and model.

Example of untidy data:

```
data.frame(
  Name = c("A", "B"),
  Math = c(80, 90),
  Science = c(85, 88)
)
```

In many analyses, data may need to be reshaped into a tidy format before further processing.

8.4 Core Packages of the Tidyverse

The tidyverse includes several core packages, each addressing a specific stage of the data analysis workflow:

- **readr** – importing rectangular data (CSV, TXT)
- **tibble** – modern reimagining of data frames
- **dplyr** – data manipulation
- **tidyr** – reshaping and tidying data
- **ggplot2** – data visualization
- **purrr** – functional programming and iteration
- **stringr** – working with strings
- **forcats** – working with categorical (factor) data

Together, these packages support the entire lifecycle of a data analysis project.

8.5 Installing and Loading the Tidyverse

The tidyverse can be installed as a single package.

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

Loading `tidyverse` automatically loads its core packages and displays a message indicating which packages are attached.

8.6 Tibbles: Modern Data Frames

A **tibble** is a modern version of a data frame with improved behavior.

```
library(tibble)

df <- tibble(
  Name = c("A", "B", "C"),
  Score = c(78, 85, 92)
)
```

Key advantages of tibbles:

- Better printing (only first rows and columns shown)
- Does not automatically convert strings to factors
- Stricter subsetting rules that reduce errors

Tibbles are the default data structure used by tidyverse functions.

8.7 The Pipe Operator %>%

One of the most recognizable features of the tidyverse is the **pipe operator %>%**. It allows code to be written as a sequence of transformations.

```
df %>%
  head()
```

The pipe passes the result of the left-hand expression as the first argument of the function on the right-hand side. This approach improves readability and mirrors logical thinking.

8.8 A Typical Tidyverse Workflow

A tidyverse-based analysis usually follows these steps:

1. Import data using `readr` or `readxl`
2. Convert data into a tidy structure
3. Manipulate and transform data using `dplyr`
4. Visualize patterns using `ggplot2`
5. Summarize and report results

This structured workflow reduces complexity and promotes reproducibility.

8.9 Base R vs Tidyverse: A Conceptual Comparison

Base R and the tidyverse can both perform the same tasks. However, tidyverse code is often:

- More readable
- Easier to maintain
- Closer to natural language descriptions of analysis steps

Example comparison:

```
# Base R
mean(df$Score)

# Tidyverse
summarise(df, AverageScore = mean(Score))
```

8.10 When to Use the Tidyverse

The tidyverse is especially useful when:

- Working with rectangular datasets
- Performing multi-step data transformations
- Teaching data analysis concepts
- Writing reproducible and readable code

However, understanding base R remains important, as both approaches complement each other.

8.11 Best Practices When Using the Tidyverse

- Load only the packages you need
- Use pipes for clarity, not complexity
- Keep data in tidy format as long as possible
- Combine tidyverse functions thoughtfully
- Comment pipelines that involve complex logic

8.12 Summary

This chapter introduced the tidyverse as a unified framework for modern data analysis in R. You learned about:

- The philosophy behind the tidyverse
- Tidy data principles
- Core tidyverse packages
- Tibbles and pipes
- Typical tidyverse workflows

With this foundation, you are now ready to explore **data manipulation using `dplyr`**, which builds directly on the concepts introduced here.

9 Data Manipulation with dplyr

9.1 Introduction

Once data has been imported into R, the next crucial step is **data manipulation**. Real-world datasets are rarely analysis-ready. They often contain unnecessary variables, missing values, inconsistent coding, or require transformation before statistical analysis can be performed.

The `dplyr` package provides a powerful, consistent, and readable set of tools for data manipulation. It is part of the **tidyverse**, a collection of R packages designed to make data science tasks intuitive and efficient. This chapter focuses on understanding the philosophy of `dplyr`, its core verbs, and their practical application through extensive examples.

9.2 Why Use dplyr?

Traditional base R functions are powerful but can become complex and difficult to read when performing multiple operations. `dplyr` improves clarity by:

- Using **simple, consistent verbs**
- Allowing operations to be expressed as a sequence of steps
- Producing readable and maintainable code
- Integrating seamlessly with data frames and tibbles

The guiding idea behind `dplyr` is:

> *Each function performs one clear data manipulation task.*

9.3 Installing and Loading dplyr

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

Most modern R installations already include `dplyr` as part of the tidyverse.

9.4 The Pipe Operator %>%

One of the defining features of `dplyr` is the **pipe operator %>%**. It allows the output of one function to be passed directly as the input to the next.

9.4.1 Without Pipe

```
summary(head(data))
```

9.4.2 With Pipe

```
data %>%
  head() %>%
  summary()
```

The pipe makes code easier to read by expressing operations **from left to right**, similar to how we describe analysis steps in words.

9.5 Selecting Variables with `select()`

The `select()` function is used to choose specific columns from a data frame.

```
data %>%
  select(Name, Age, Salary)
```

9.5.1 Selecting by Position

```
data %>%
  select(1:3)
```

9.5.2 Excluding Variables

```
data %>%
  select(-Salary)
```

9.5.3 Helper Functions

```
data %>%
  select(starts_with("Age"))
```

Common helpers include `starts_with()`, `ends_with()`, `contains()`, and `matches()`.

9.6 Filtering Observations with `filter()`

The `filter()` function extracts rows that satisfy logical conditions.

```
data %>%
  filter(Age > 30)
```

9.6.1 Multiple Conditions

```
data %>%
  filter(Age > 30 & Gender == "Male")
```

9.6.2 Using Logical OR

```
data %>%
  filter(Department == "HR" | Department == "Finance")
```

9.6.3 Handling Missing Values

```
data %>%
  filter(!is.na(Salary))
```

9.7 Creating and Transforming Variables with `mutate()`

The `mutate()` function creates new variables or modifies existing ones.

```
data %>%
  mutate(Bonus = Salary * 0.10)
```

9.7.1 Multiple Transformations

```
data %>%
  mutate(
    Bonus = Salary * 0.10,
    AnnualSalary = Salary * 12
  )
```

9.7.2 Conditional Variables with `ifelse()`

```
data %>%
  mutate(Performance = ifelse(Salary > 50000, "High", "Average"))
```

9.8 Arranging Rows with `arrange()`

The `arrange()` function sorts data.

```
data %>%
  arrange(Salary)
```

9.8.1 Descending Order

```
data %>%
  arrange(desc(Salary))
```

9.9 Summarising Data with `summarise()`

The `summarise()` function reduces multiple values into summary statistics.

```
data %>%
  summarise(
    AverageSalary = mean(Salary, na.rm = TRUE),
    MaxSalary = max(Salary, na.rm = TRUE)
  )
```

9.10 Grouped Operations with group_by()

The true power of dplyr lies in grouped analysis.

```
data %>%
  group_by(Department) %>%
  summarise(
    AvgSalary = mean(Salary, na.rm = TRUE),
    Count = n()
  )
```

Grouped operations are fundamental in statistical reporting and exploratory analysis.

9.11 Counting Observations

```
data %>%
  count(Gender)
```

Equivalent to grouping and counting in one step.

9.12 Renaming Variables with rename()

```
data %>%
  rename(
    MonthlySalary = Salary,
    EmployeeAge = Age
  )
```

9.13 Removing Duplicate Rows with `distinct()`

```
data %>%
  distinct(EmployeeID, .keep_all = TRUE)
```

9.14 Combining Multiple dplyr Verbs

A typical real-world workflow:

```
data %>%
  filter(!is.na(Salary)) %>%
  mutate(AnnualSalary = Salary * 12) %>%
  group_by(Department) %>%
  summarise(
    AvgAnnualSalary = mean(AnnualSalary),
    Employees = n()
  ) %>%
  arrange(desc(AvgAnnualSalary))
```

This example demonstrates how complex analysis can be expressed in a clear, step-by-step pipeline.

9.15 Working with Missing Values

```
data %>%
  summarise(MissingSalary = sum(is.na(Salary)))
```

Handling missing data explicitly is a critical step before any statistical modeling.

9.16 Converting to Tibble

```
data_tbl <- as_tibble(data)
```

Tibbles provide improved printing and stricter behavior compared to base data frames.

9.17 Best Practices for Data Manipulation

- Always inspect data before and after manipulation
- Avoid overwriting original datasets
- Use clear variable names
- Break complex pipelines into readable steps
- Comment your code when logic is not obvious

9.18 Summary

This chapter introduced the principles and practical usage of `dplyr` for data manipulation. You learned how to:

- Select and filter data
- Create and transform variables
- Summarise and group data
- Combine multiple operations using pipes

Mastering `dplyr` is essential for efficient data analysis in R. The next chapter will focus on **data visualization using `ggplot2`**, where manipulated data is transformed into meaningful graphical representations.

10 Tidyr and Tidy Data

10.1 Introduction

In practical data analysis, datasets are often **not in a form that is immediately suitable for analysis or visualization**. Even after importing data and understanding tidyverse principles, analysts frequently encounter data that is spread across multiple columns, nested in complex structures, or stored in inconvenient formats.

The `tidyr` package is designed to address this problem. It provides a set of tools to **reshape, reorganize, and tidy data**, making it easier to analyze using `dplyr`, visualize using `ggplot2`, and model using statistical methods. This chapter focuses on the concept of tidy data and demonstrates how `tidyr` helps transform untidy datasets into a clean and consistent structure.

10.2 Understanding Tidy Data

The idea of tidy data is central to modern data analysis in R. A dataset is considered tidy when:

- Each variable is stored in its own column
- Each observation is stored in its own row
- Each type of observational unit forms a separate table

These principles may appear simple, but many real-world datasets violate them in subtle ways. Untidy data makes analysis more error-prone and code more complex.

10.3 Common Causes of Untidy Data

Data becomes untidy for several reasons:

- Data is designed for human readability rather than analysis
- Multiple variables are encoded in column names
- Values are spread across multiple columns
- Missing values are represented implicitly

Recognizing these patterns is the first step toward tidying data effectively.

10.4 Installing and Loading tidyr

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

The `tidyr` package is part of the tidyverse and is often loaded automatically when `library(tidyverse)` is used.

10.5 Wide and Long Data Formats

Two common data layouts encountered in practice are **wide format** and **long format**.

- **Wide format**: Values are spread across multiple columns
- **Long format**: Values are stacked in a single column, with another column indicating the variable

Long format is generally preferred for analysis and visualization in R.

Example of wide data:

```
wide_data <- data.frame(
  ID = c(1, 2, 3),
  Math = c(80, 75, 90),
  Science = c(85, 70, 88)
)
```

10.6 Converting Wide Data to Long Data with `pivot_longer()`

The `pivot_longer()` function transforms data from wide to long format.

```
long_data <- wide_data %>%
  pivot_longer(
    cols = c(Math, Science),
    names_to = "Subject",
    values_to = "Score"
  )
```

In this transformation:

- Column names become values in a new variable
- Observations are stacked into a single column

This structure is ideal for grouped summaries and visualizations.

10.7 Converting Long Data to Wide Data with `pivot_wider()`

The `pivot_wider()` function performs the reverse operation.

```
wide_again <- long_data %>%
  pivot_wider(
    names_from = Subject,
    values_from = Score
  )
```

This is useful when preparing summary tables or reports.

10.8 Handling Multiple Variables in Column Names

Some datasets encode multiple variables in column names.

```
messy_data <- data.frame(
  ID = c(1, 2),
  Height_cm = c(170, 165),
  Weight_kg = c(65, 58)
)
```

Such datasets can be reshaped by separating column names into meaningful components.

10.9 Separating Columns with `separate()`

```
separated_data <- messy_data %>%
  pivot_longer(-ID, names_to = "Measure", values_to = "Value") %>%
  separate(Measure, into = c("Variable", "Unit"), sep = "_")
```

This creates distinct variables for the measurement type and unit.

10.10 Combining Columns with `unite()`

The `unite()` function merges multiple columns into one.

```
combined_data <- separated_data %>%
  unite("Measurement", Variable, Unit, sep = "_")
```

This is useful when preparing labels or compact identifiers.

10.11 Dealing with Missing Values

In tidy data, missing values should be **explicitly represented as NA**.

```
data_with_na <- data.frame(
  ID = c(1, 2, 3),
  Score = c(80, NA, 90)
)
```

10.11.1 Dropping Missing Values

```
data_with_na %>%
  drop_na()
```

10.11.2 Filling Missing Values

```
data_with_na %>%
  fill(Score, .direction = "down")
```

10.12 Expanding and Completing Data

Sometimes combinations of variables are missing entirely.

```
incomplete_data <- data.frame(
  ID = c(1, 1, 2),
  Year = c(2022, 2023, 2022),
  Score = c(80, 85, 78)
)
```

```
complete_data <- incomplete_data %>%
  complete(ID, Year)
```

The `complete()` function ensures that all combinations are represented.

10.13 Nesting and Unnesting Data

Tidyr also supports working with nested data structures.

```
nested_data <- long_data %>%
  group_by(Subject) %>%
  nest()
```

```
unnested_data <- nested_data %>%
  unnest(cols = data)
```

Nesting is particularly useful for advanced analysis and modeling workflows.

10.14 Tidyr in the Data Analysis Workflow

`tidyr` is typically used **before `dplyr` summarization and `ggplot2` visualization**. A common workflow is:

1. Import data
2. Tidy and reshape using `tidyr`
3. Manipulate and summarize using `dplyr`
4. Visualize using `ggplot2`

Keeping data tidy simplifies every subsequent step.

10.15 Best Practices for Tidying Data

- Always inspect raw data before reshaping
- Make missing values explicit
- Keep variable names meaningful
- Avoid unnecessary reshaping
- Tidy data once, then reuse it

10.16 Summary

This chapter introduced the principles of tidy data and the tools provided by the `tidyr` package. You learned how to:

- Recognize untidy data structures
- Convert between wide and long formats
- Separate and unite variables
- Handle missing and incomplete data
- Work with nested data

Mastering `tidyr` ensures that your data is in a clean, consistent form, making analysis, visualization, and modeling both easier and more reliable.

11 Basic Plots in R (Base Graphics)

11.1 Introduction

Before the development of advanced visualization systems such as `ggplot2`, R provided a powerful built-in plotting system known as **base graphics**. Even today, base R plots remain widely used due to their **simplicity, speed, and flexibility**, especially for quick exploratory analysis and teaching fundamental graphical concepts.

This chapter introduces the **basic plotting functions in R**, explains how they work, and demonstrates how to customize plots using base graphics. Understanding base plots is important for building strong visualization fundamentals and for interpreting legacy R code commonly found in academic and research settings.

11.2 The Base Graphics System in R

Base graphics in R follow a **pen-and-paper model**:

- A plot is created first
- Additional elements are added layer by layer

Once a plot is drawn, it cannot be easily modified; instead, it must be redrawn. This contrasts with `ggplot2`, which builds plots declaratively.

11.3 The `plot()` Function

The `plot()` function is the most fundamental plotting function in R. Its behavior depends on the type of data provided.

11.3.1 Scatter Plot

```
x <- c(1, 2, 3, 4, 5)
y <- c(2, 4, 6, 8, 10)
plot(x, y)
```

This produces a scatter plot showing the relationship between `x` and `y`.

11.3.2 Plotting a Single Vector

```
plot(y)
```

When a single vector is provided, R plots the values against their index.

11.4 Customizing Plots

Base R plots can be customized using graphical parameters.

```
plot(x, y,
     main = "Simple Scatter Plot",
     xlab = "X values",
     ylab = "Y values",
     col = "blue",
     pch = 19)
```

Common parameters include:

- `main` – title of the plot
- `xlab`, `ylab` – axis labels
- `col` – color
- `pch` – plotting symbol

11.5 Line Plots

Line plots are useful for showing trends and time-based data.

```
plot(x, y, type = "l")
```

To add both points and lines:

```
plot(x, y, type = "b")
```

11.6 Adding Elements to an Existing Plot

Additional elements can be added using separate functions.

11.6.1 Adding Points

```
plot(x, y, type = "l")
points(x, y, pch = 16)
```

11.6.2 Adding Lines

```
plot(x, y)
lines(x, y, col = "red")
```

11.6.3 Adding Text

```
text(x, y, labels = y)
```

11.7 Histograms

Histograms display the distribution of a numeric variable.

```
hist(mtcars$mpg)
```

Customized histogram:

```
hist(mtcars$mpg,
     breaks = 10,
     col = "lightgray",
     main = "Histogram of MPG",
     xlab = "Miles per Gallon")
```

11.8 Boxplots

Boxplots summarize the distribution of data using quartiles.

```
boxplot(mtcars$mpg)
```

Grouped boxplot:

```
boxplot(mpg ~ cyl, data = mtcars,
        main = "MPG by Number of Cylinders",
        xlab = "Cylinders",
        ylab = "Miles per Gallon")
```

11.9 Bar Plots

Bar plots are used for categorical data or summarized values.

```
counts <- table(mtcars$cyl)
barplot(counts)
```

Customized bar plot:

```
barplot(counts,
        col = "steelblue",
        main = "Car Counts by Cylinders",
        xlab = "Cylinders",
        ylab = "Frequency")
```

11.10 Pie Charts

Pie charts show proportions of categories.

```
pie(counts)
```

Although easy to create, pie charts are generally discouraged in statistical analysis due to difficulty in comparing values accurately.

11.11 Multiple Plots in One Window

The `par()` function allows multiple plots to be displayed together.

```
par(mfrow = c(2, 2))
plot(mtcars$mpg)
hist(mtcars$mpg)
boxplot(mtcars$mpg)
barplot(counts)
```

Reset layout after plotting:

```
par(mfrow = c(1, 1))
```

11.12 Saving Plots to Files

Plots can be saved using graphical devices.

```
png("scatter_plot.png")
plot(x, y)
dev.off()
```

Common devices include `png()`, `pdf()`, and `jpeg()`.

11.13 When to Use Base R Plots

Base plots are especially useful for:

- Quick exploratory analysis
- Teaching fundamental plotting concepts
- Simple visualizations without heavy customization
- Working with legacy R scripts

For complex, publication-quality graphics, `ggplot2` is generally preferred.

11.14 Summary

This chapter introduced **basic plotting in R using base graphics**. You learned how to:

- Create scatter, line, histogram, box, bar, and pie plots
- Customize plot appearance
- Add elements to existing plots
- Arrange multiple plots
- Save plots to external files

Understanding base R graphics provides a strong foundation for advanced visualization techniques and helps in interpreting a wide range of existing R code.

12 Data Visualization with ggplot2

12.1 Introduction

Data visualization is a fundamental component of data analysis. Well-designed graphics help us **explore patterns**, **identify relationships**, **detect anomalies**, and **communicate results effectively**. In statistical analysis, visualizations often reveal insights that numerical summaries alone cannot capture.

The `ggplot2` package is the most widely used visualization system in R. It is part of the tidyverse and is based on the **Grammar of Graphics**, a systematic approach to describing and building plots. This chapter introduces `ggplot2` conceptually and practically, focusing on clear explanations and progressively richer examples.

12.2 Why ggplot2?

Unlike traditional plotting systems that focus on individual plot types, `ggplot2` focuses on **building plots layer by layer**. This approach offers several advantages:

- Consistent syntax across different plot types
- High-quality, publication-ready graphics
- Seamless integration with tidy data and `dplyr`
- Easy customization and extension

The guiding idea is simple:

> *A plot is constructed by mapping data to visual properties and adding geometric layers.*

12.3 Installing and Loading ggplot2

```
install.packages("ggplot2")
library(ggplot2)
```

When the tidyverse is loaded, `ggplot2` is loaded automatically.

12.4 The Grammar of Graphics: Core Concepts

Every `ggplot2` visualization is built from the same core components:

- **Data** – the dataset being visualized
- **Aesthetic mappings (`aes`)** – how variables are mapped to visual properties
- **Geometric objects (`geoms`)** – the type of plot
- **Statistical transformations** – summaries applied to the data
- **Scales** – control how data values map to aesthetics
- **Coordinates** – the coordinate system used
- **Facets** – small multiples for subplots

Understanding these ideas allows you to create almost any statistical graphic.

12.5 Creating Your First ggplot

A `ggplot2` plot begins with the `ggplot()` function.

```
ggplot(data = mtcars, aes(x = wt, y = mpg))
```

At this stage, nothing is displayed because no geometric layer has been added.

12.6 Scatter Plots with `geom_point()`

Scatter plots are useful for examining relationships between two numeric variables.

```
ggplot(mtcars, aes(x = wt, y = mpg)) +
  geom_point()
```

12.6.1 Adding Aesthetics

```
ggplot(mtcars, aes(x = wt, y = mpg, color = factor(cyl))) +
  geom_point(size = 3)
```

Here, the number of cylinders is mapped to color, revealing group-wise patterns.

12.7 Line Plots with geom_line()

Line plots are commonly used for time series or ordered data.

```
ggplot(economics, aes(x = date, y = unemploy)) +
  geom_line()
```

12.8 Bar Charts with geom_bar() and geom_col()

12.8.1 Bar Chart for Counts

```
ggplot(mtcars, aes(x = factor(cyl))) +
  geom_bar()
```

12.8.2 Bar Chart for Precomputed Values

```
avg_mpg <- mtcars %>%
  group_by(cyl) %>%
  summarise(avg_mpg = mean(mpg))

ggplot(avg_mpg, aes(x = factor(cyl), y = avg_mpg)) +
  geom_col()
```

12.9 Histograms and Density Plots

12.9.1 Histogram

```
ggplot(mtcars, aes(x = mpg)) +
  geom_histogram(bins = 10)
```

12.9.2 Density Plot

```
ggplot(mtcars, aes(x = mpg)) +
  geom_density()
```

These plots help understand the distribution of a variable.

12.10 Boxplots and Violin Plots

Boxplots summarize distributions across groups.

```
ggplot(mtcars, aes(x = factor(cyl), y = mpg)) +
  geom_boxplot()
```

Violin plots display the full distribution shape.

```
ggplot(mtcars, aes(x = factor(cyl), y = mpg)) +
  geom_violin()
```

12.11 Faceting: Small Multiples

Faceting allows the same plot to be repeated for different subsets of data.

```
ggplot(mtcars, aes(x = wt, y = mpg)) +
  geom_point() +
  facet_wrap(~ cyl)
```

This technique is powerful for comparative analysis.

12.12 Labels and Titles

```
ggplot(mtcars, aes(x = wt, y = mpg)) +
  geom_point() +
  labs(
    title = "Fuel Efficiency vs Weight",
    x = "Weight",
    y = "Miles per Gallon",
    color = "Cylinders"
  )
```

Clear labeling is essential for effective communication.

12.13 Themes and Appearance

Themes control the non-data components of a plot.

```
ggplot(mtcars, aes(x = wt, y = mpg)) +
  geom_point() +
  theme_minimal()
```

Common themes include `theme_bw()`, `theme_classic()`, and `theme_minimal()`.

12.14 Customizing Scales

```
ggplot(mtcars, aes(x = wt, y = mpg, color = factor(cyl))) +
  geom_point() +
  scale_color_brewer(palette = "Set1")
```

Scales allow fine control over colors, axes, and legends.

12.15 Combining ggplot2 with dplyr

A typical workflow combines data manipulation and visualization.

```r
mtcars %>%
  group_by(cyl) %>%
  summarise(avg_mpg = mean(mpg)) %>%
  ggplot(aes(x = factor(cyl), y = avg_mpg)) +
  geom_col()
```

This integration is one of the major strengths of the tidyverse.

12.16 Best Practices for Statistical Graphics

- Start with simple plots and add complexity gradually
- Always label axes and legends clearly
- Avoid unnecessary visual clutter
- Choose appropriate plot types for the data
- Ensure plots are reproducible and interpretable

12.17 Summary

This chapter introduced **data visualization using `ggplot2`**. You learned:

- The philosophy of the Grammar of Graphics
- How to build plots layer by layer
- Common plot types and their use cases
- Faceting, themes, and customization
- Integration with tidyverse workflows

Effective visualization is essential for statistical analysis and communication. In the next chapter, we will focus on **Exploratory Data Analysis (EDA)**, where numerical summaries and visualizations are combined to understand data comprehensively.

13 String Manipulation with stringr

13.1 Introduction

Text data is an integral part of modern data analysis. Variables such as names, addresses, categories, survey responses, clinical codes, and identifiers are all stored as **character strings**. Cleaning, transforming, and extracting information from such text data is a common and often challenging task.

The `stringr` package provides a consistent and user-friendly set of functions for string manipulation in R. It is part of the tidyverse and is built on top of R's powerful string processing capabilities. This chapter introduces the principles of string manipulation and demonstrates how `stringr` simplifies working with character data.

13.2 Why Use stringr?

Base R includes many functions for working with strings, but they can be difficult to remember due to inconsistent naming conventions. `stringr` improves this by:

- Using a **consistent function naming scheme** (`str_*`)
- Handling missing values (`NA`) safely
- Providing clear and readable syntax
- Integrating seamlessly with tidyverse workflows

The guiding idea is:

> *Every string function begins with `str_` and performs one clear task.*

13.3 Installing and Loading stringr

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

When the tidyverse is loaded using `library(tidyverse)`, `stringr` is loaded automatically.

13.4 Creating and Inspecting Strings

```
text <- c("Statistics", "Data Science", "R Programming")
text
```

To determine the length of each string:

```
str_length(text)
```

Unlike base R's `nchar()`, `str_length()` handles missing values consistently.

13.5 Combining Strings

13.5.1 Concatenation with `str_c()`

```
first_name <- c("Anita", "Rahul")
last_name <- c("K", "S")

full_name <- str_c(first_name, last_name, sep = " ")
full_name
```

To collapse multiple strings into a single string:

```
str_c(text, collapse = ", ")
```

13.6 Subsetting and Extracting Strings

13.6.1 Substrings with `str_sub()`

```
code <- "STAT2024"
str_sub(code, 1, 4)
```

Negative indices count from the end of the string:

```
str_sub(code, -4, -1)
```

13.7 Detecting Patterns in Strings

13.7.1 Pattern Detection with `str_detect()`

```
emails <- c("abc@gmail.com", "test@yahoo.com", "user@college.edu")
str_detect(emails, "gmail")
```

This function returns a logical vector indicating the presence of a pattern.

13.8 Counting and Locating Patterns

13.8.1 Counting Matches

```
sentences <- c("R is powerful", "R is very very powerful")
str_count(sentences, "very")
```

13.8.2 Locating Matches

```
str_locate(sentences, "powerful")
```

13.9 Replacing Text

13.9.1 Replacing First Match

```
text2 <- c("STAT101", "STAT102")
str_replace(text2, "STAT", "DS")
```

13.9.2 Replacing All Matches

```
str_replace_all("2024-01-01", "-", "/")
```

13.10 Removing Whitespace

Whitespace is a common issue in imported text data.

```
raw_text <- c("  Data ", " Science ", " R ")
str_trim(raw_text)
```

To remove whitespace only from the left or right:

```
str_trim(raw_text, side = "left")
str_trim(raw_text, side = "right")
```

13.11 Changing Case of Strings

```
text3 <- c("statistics", "DATA SCIENCE", "R")

str_to_upper(text3)
str_to_lower(text3)
str_to_title(text3)
```

Case normalization is often required before comparisons or grouping.

13.12 Splitting Strings

13.12.1 Splitting into Components

```
course_codes <- c("STAT-101", "STAT-202", "STAT-303")
str_split(course_codes, "-")
```

To simplify the output:

```
str_split_fixed(course_codes, "-", 2)
```

13.13 Working with Regular Expressions (Introductory)

`stringr` uses **regular expressions (regex)** to describe patterns.

```
str_detect("abc123", "[0-9]")
```

Common regex elements:

- `[0-9]` – digits
- `[A-Za-z]` – letters
- `^` – start of string
- `$` – end of string

13.14 Handling Missing Values in Strings

```
text_with_na <- c("A", NA, "B")
str_length(text_with_na)
```

`stringr` functions return `NA` for missing values instead of errors.

13.15 stringr in Data Cleaning Workflows

`stringr` is commonly used together with `dplyr`.

```
data %>%
  mutate(
    Name = str_to_title(Name),
    Code = str_replace_all(Code, " ", "")
  )
```

This combination is especially useful for preparing data for analysis.

13.16 Best Practices for String Manipulation

- Always inspect raw text before cleaning
- Be explicit about patterns you want to match
- Normalize case early in the workflow
- Trim whitespace after importing data
- Use simple regex patterns when possible

13.17 Summary

This chapter introduced **string manipulation using the `stringr` package**. You learned how to:

- Measure and combine strings
- Extract and replace text
- Detect and count patterns
- Handle whitespace and case
- Split strings into components
- Apply string operations within tidyverse workflows

Mastering string manipulation is essential for cleaning real-world datasets and preparing text variables for statistical analysis.

14 Data Transformation in R

14.1 Introduction

Data transformation is the process of **modifying data to make it suitable for analysis, modeling, and interpretation**. After importing, tidying, and manipulating data, analysts often need to transform variables to improve interpretability, meet statistical assumptions, or derive new insights.

In statistics and data science, data transformation is not merely a technical step—it is an **analytical decision**. This chapter introduces the most common data transformation techniques in R, with an emphasis on understanding *why* transformations are applied, not just *how* they are implemented.

14.2 Why Data Transformation Is Necessary

Real-world data frequently exhibits issues such as:

- Skewed distributions
- Outliers
- Inconsistent scales
- Categorical variables encoded numerically
- Variables measured in inconvenient units

Data transformation helps to:

- Improve model performance
- Satisfy statistical assumptions
- Enhance interpretability
- Enable meaningful comparisons
- Prepare data for visualization and reporting

14.3 Creating New Variables

One of the most common transformations is the creation of new variables from existing ones.

```
data <- data.frame(
  Salary = c(25000, 40000, 60000),
  Experience = c(2, 5, 10)
)

data$AnnualSalary <- data$Salary * 12
data
```

Derived variables often carry more analytical meaning than raw measurements.

14.4 Mathematical Transformations

14.4.1 Log Transformation

Logarithmic transformations are used to reduce skewness and stabilize variance.

```
data$LogSalary <- log(data$Salary)
```

Log transformations are common in income, clinical, and biological data.

14.4.2 Square Root Transformation

```
data$SqrtSalary <- sqrt(data$Salary)
```

Square root transformations are useful for count data and moderately skewed variables.

14.5 Standardization and Scaling

Variables measured on different scales often need to be standardized.

14.5.1 Standardization (Z-score)

```
data$Salary_z <- scale(data$Salary)
```

Standardized variables have mean 0 and standard deviation 1.

14.5.2 Min–Max Scaling

```
min_max <- function(x) {
  (x - min(x)) / (max(x) - min(x))
}

data$Salary_scaled <- min_max(data$Salary)
```

Scaling is essential in machine learning and distance-based methods.

14.6 Recoding Variables

Recoding replaces existing values with more meaningful or simplified categories.

```
data$Level <- ifelse(data$Experience < 3, "Junior",
                     ifelse(data$Experience < 8, "Mid", "Senior"))
```

Recoding is especially common in survey and categorical data.

14.7 Transforming Categorical Variables

Categorical variables often require transformation before analysis.

```
data$Level <- factor(data$Level,
                     levels = c("Junior", "Mid", "Senior"))
```

Proper ordering of factor levels is important for modeling and visualization.

14.8 Handling Outliers Through Transformation

Transformations can reduce the influence of extreme values.

```
boxplot(data$Salary)
```

Applying a log transformation:

```
boxplot(log(data$Salary))
```

This often results in a more symmetric distribution.

14.9 Conditional Transformations

Conditional logic allows transformations based on rules.

```
data$Bonus <- ifelse(data$Salary > 50000,
                     data$Salary * 0.10,
                     data$Salary * 0.05)
```

Such transformations are common in policy-based or business-rule-driven analyses.

14.10 Transformations Using dplyr

Most transformations are performed using `mutate()` in tidyverse workflows.

```
library(dplyr)

data <- data %>%
  mutate(
    AnnualSalary = Salary * 12,
    LogSalary = log(Salary),
    Level = case_when(
      Experience < 3 ~ "Junior",
      Experience < 8 ~ "Mid",
      TRUE ~ "Senior"
    )
  )
```

This approach improves readability and reproducibility.

14.11 Transformations for Visualization

Transforming data before plotting can reveal hidden patterns.

```
hist(data$Salary)
hist(log(data$Salary))
```

Choosing appropriate transformations is a key part of exploratory data analysis.

14.12 Common Mistakes in Data Transformation

- Applying transformations without justification
- Forgetting to handle zero or negative values in log transforms
- Overwriting original variables
- Ignoring interpretability after transformation

Maintaining both original and transformed variables is often best practice.

14.13 Best Practices for Data Transformation

- Understand the purpose of each transformation
- Keep transformations simple and interpretable
- Document transformations clearly
- Apply transformations consistently
- Validate results visually and numerically

14.14 Summary

This chapter introduced the concept and practice of **data transformation in R**. You learned how to:

- Create derived variables
- Apply mathematical transformations
- Standardize and scale data
- Recode and transform categorical variables
- Handle outliers through transformation

- Perform transformations using tidyverse workflows

Data transformation is a critical step that connects raw data to meaningful statistical analysis and modeling.

15 Working with Dates and Times in R

15.1 Introduction

Date and time variables play a crucial role in many areas of statistics and data science, including **time series analysis, clinical studies, event history analysis, surveys, and operational analytics**. Unlike numeric variables, dates and times carry inherent structure, such as ordering, intervals, and time zones, which must be handled carefully.

R provides built-in support for dates and times, and the **`lubridate`** package (part of the tidyverse) greatly simplifies working with them. This chapter introduces how dates and times are represented in R, common operations performed on them, and best practices for handling temporal data.

15.2 Date and Time Classes in R

R represents temporal data using specific classes:

- **Date** – calendar dates without time (e.g., 2024-01-15)
- **POSIXct** – date-time stored as seconds since the Unix epoch
- **POSIXlt** – date-time stored as a list of components

Understanding these classes is essential for correct analysis.

15.3 Creating Date Objects

Dates in R typically follow the ISO 8601 format: `YYYY-MM-DD`.

```
date1 <- as.Date("2024-01-01")
date2 <- as.Date("2024-12-31")

date1
date2
```

Different formats can be specified explicitly:

```r
as.Date("01-12-2024", format = "%d-%m-%Y")
```

15.4 Date Arithmetic

Once dates are created, arithmetic operations can be performed.

```r
date2 - date1
```

This returns the number of days between two dates.

Adding or subtracting days:

```r
date1 + 30
date2 - 15
```

15.5 Creating Date-Time Objects

Date-time objects include both date and time components.

```r
dt1 <- as.POSIXct("2024-01-01 10:30:00")
dt2 <- as.POSIXct("2024-01-01 18:45:00")
```

The default time zone is taken from the system unless specified.

15.6 Understanding POSIXct and POSIXlt

```r
class(dt1)
unclass(dt1)
```

- **POSIXct** is efficient and preferred for storage
- **POSIXlt** is useful for extracting components

```r
as.POSIXlt(dt1)
```

15.7 Extracting Components from Dates

Base R provides functions to extract components:

```
format(date1, "%Y")   # Year
format(date1, "%m")   # Month
format(date1, "%d")   # Day
```

However, these return character values, which may require conversion.

15.8 Introduction to lubridate

The **lubridate** package simplifies working with dates and times by providing intuitive functions.

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

15.9 Parsing Dates with lubridate

Lubridate automatically recognizes common date formats.

```
d1 <- ymd("2024-01-15")
d2 <- dmy("15-01-2024")
d3 <- mdy("01/15/2024")
```

These functions return Date objects without requiring format strings.

15.10 Parsing Date-Time Values

```
dt <- ymd_hms("2024-01-01 10:30:00")
```

Lubridate functions include `ymd_hm()`, `dmy_hms()`, and others.

15.11 Extracting Date Components with lubridate

```
year(d1)
month(d1)
day(d1)
```

Additional components:

```
wday(d1)
week(d1)
quarter(d1)
```

Lubridate returns numeric values by default, which are easy to use in analysis.

15.12 Working with Time Zones

Time zones are critical in global and clinical data.

```
dt_tz <- ymd_hms("2024-01-01 10:30:00", tz = "UTC")
```

Converting time zones:

```
with_tz(dt_tz, "Asia/Kolkata")
```

15.13 Time Differences and Durations

Lubridate distinguishes between durations, periods, and intervals.

15.13.1 Durations

```
duration <- dhours(5)
```

15.13.2 Periods

```
period <- days(10)
```

15.13.3 Intervals

```
interval <- interval(date1, date2)
```

Each serves a different analytical purpose.

15.14 Adding and Subtracting Time Periods

```
date1 + days(7)
date1 + months(1)
date1 - years(2)
```

Lubridate handles calendar irregularities automatically.

15.15 Rounding and Flooring Dates

```
floor_date(dt, unit = "hour")
ceiling_date(dt, unit = "day")
round_date(dt, unit = "month")
```

These operations are useful for aggregation and reporting.

15.16 Dates and Times in Data Frames

Dates often appear as character variables when imported.

```
data <- data.frame(
  ID = 1:3,
  VisitDate = c("2024-01-01", "2024-02-15", "2024-03-10")
)

data$VisitDate <- ymd(data$VisitDate)
```

Converting early in the workflow is considered best practice.

15.17 Dates in Analysis and Visualization

Dates are commonly used in grouping and plotting.

```
data %>%
  mutate(Month = month(VisitDate)) %>%
  group_by(Month) %>%
  summarise(Count = n())
```

Temporal aggregation is a key step in time-based analysis.

15.18 Common Issues with Dates and Times

- Incorrect date formats
- Implicit time zone conversions
- Treating dates as characters
- Ignoring daylight saving time

Careful inspection and explicit conversion help avoid these issues.

15.19 Best Practices for Working with Dates and Times

- Convert date variables immediately after import
- Use ISO formats whenever possible
- Prefer `lubridate` for clarity and safety
- Always be explicit about time zones
- Validate transformations using summaries and plots

15.20 Summary

This chapter introduced how to **work with dates and times in R**. You learned:

- Date and time classes in R
- Creating and manipulating Date and POSIX objects
- Using `lubridate` for parsing and extraction
- Handling time zones and intervals
- Applying date-time operations in data analysis

Correct handling of dates and times is essential for reliable statistical analysis, particularly in longitudinal and time-based studies.

16 Functional Programming in R

16.1 Introduction

Functional programming is a programming paradigm that treats **functions as first-class objects** and emphasizes writing code by composing functions rather than modifying state. In R, functional programming is especially powerful for writing **concise, reusable, and reliable code**, and it plays a central role in modern data analysis workflows.

This chapter introduces the principles of functional programming in R, starting from base R concepts and extending to tidyverse-friendly tools such as **purrr**. The focus is on *why* functional programming is useful, *when* to use it, and *how* it improves clarity and correctness in data analysis.

16.2 Functions as Objects in R

In R, functions are objects just like vectors or data frames. They can be:

- Assigned to variables
- Passed as arguments to other functions
- Returned as outputs from functions

```
f <- mean
f(c(10, 20, 30))
```

This property allows R users to write highly flexible and expressive code.

16.3 Writing Your Own Functions

Creating functions helps avoid repetition and improves code organization.

```
add_percentage <- function(x, p) {
  x + (x * p / 100)
}

add_percentage(100, 10)
```

A function typically consists of:

- A name
- Input arguments
- A body containing expressions
- A return value (implicit or explicit)

16.4 Pure Functions and Side Effects

A **pure function** always produces the same output for the same input and does not modify external variables.

```
square <- function(x) {
  x^2
}
```

Pure functions are easier to test, debug, and reuse. Avoiding side effects is considered good practice in data analysis code.

16.5 Avoiding Loops with Vectorization

R is designed for vectorized operations, which are faster and more readable than explicit loops.

```
x <- 1:5
x^2
```

Compare with a loop:

```
result <- numeric(length(x))
for (i in seq_along(x)) {
  result[i] <- x[i]^2
}
```

Vectorization is often the first step toward functional programming in R.

16.6 The apply Family of Functions

Base R provides a family of functions for applying operations over data structures.

16.6.1 `lapply()`

```
numbers <- list(1:5, 6:10, 11:15)
lapply(numbers, mean)
```

16.6.2 `sapply()`

```
sapply(numbers, mean)
```

16.6.3 `apply()`

```
mat <- matrix(1:6, nrow = 2)
apply(mat, 1, sum)
```

These functions reduce the need for explicit loops and improve clarity.

16.7 Anonymous Functions

Anonymous (or lambda) functions are functions without names, often used for short operations.

```
lapply(numbers, function(x) x^2)
```

They are useful when a function is needed only once.

16.8 Introduction to purrr

The **purrr** package provides a modern, consistent alternative to the `apply` family. It is part of the tidyverse and is designed to work seamlessly with pipelines.

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

16.9 Mapping Functions with `map()`

The core idea of `purrr` is **mapping a function over a vector or list**.

```
map(numbers, mean)
```

Typed variants ensure predictable output:

```
map_dbl(numbers, mean)
map_int(numbers, length)
```

16.10 Mapping with Anonymous Functions

```
map(numbers, ~ .x * 2)
```

The `.x` syntax provides a concise way to define anonymous functions.

16.11 Mapping Over Multiple Inputs

```
x <- c(1, 2, 3)
y <- c(4, 5, 6)

map2(x, y, ~ .x + .y)
```

This is useful when combining multiple vectors element-wise.

16.12 Iteration with Side Effects: `walk()`

Sometimes iteration is needed for side effects, such as printing or saving files.

```
walk(x, print)
```

Unlike `map()`, `walk()` returns no output.

16.13 Functional Programming in Data Analysis Workflows

Functional programming is often used to apply the same operation to multiple variables or datasets.

```
df <- data.frame(
  A = c(1, 2, 3),
  B = c(4, 5, 6)
)

map(df, mean)
```

This approach avoids repetitive code and scales well as data grows.

16.14 Error Handling with `possibly()` and `safely()`

Functional programming also supports robust error handling.

```
safe_log <- possibly(log, NA)
safe_log(c(1, -1, 10))
```

This prevents entire workflows from failing due to a single error.

16.15 When to Use Functional Programming

Functional programming is particularly useful when:

- Repeating the same operation many times
- Working with lists or nested data
- Writing reusable and modular code
- Building reproducible data analysis pipelines

16.16 Best Practices for Functional Programming

- Prefer vectorization over loops
- Write small, single-purpose functions
- Use `map()` instead of `lapply()` in tidyverse workflows
- Avoid side effects when possible
- Name functions clearly and document them

16.17 Summary

This chapter introduced the principles of **functional programming in R**. You learned how to:

- Treat functions as objects
- Write and use custom functions
- Replace loops with vectorized and functional approaches
- Use the `apply` family and `purrr` mapping functions
- Build robust, reusable data analysis code

Functional programming is a powerful mindset that improves code quality, readability, and scalability in R-based data analysis.

17 Factors and Categorical Data with forcats

17.1 Introduction

Categorical data plays a central role in statistics, representing **qualitative attributes** such as gender, treatment groups, education levels, disease status, or survey responses. In R, categorical variables are handled using a special data type called a **factor**. Correct handling of factors is essential for **statistical modeling, data summarization, and visualization**.

While base R provides basic tools for working with factors, the `forcats` package (part of the tidyverse) offers a more intuitive, consistent, and powerful approach. This chapter focuses on understanding factors conceptually and using `forcats` to manage categorical data effectively in modern R workflows.

Understanding Factors in R

A factor is a data structure used to represent categorical data with a fixed set of possible values called **levels**.

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

Factors store data internally as integers with associated labels. This internal structure makes factors efficient and suitable for statistical modeling.

Why Factors Matter in Statistics

Factors are not just labels; they influence how R performs:

- Group-wise summaries
- Statistical tests
- Regression and ANOVA models
- Plot ordering and legends

Incorrect factor handling can lead to misleading results, especially in modeling and interpretation.

Creating Factors

From Character Vectors

```
status <- c("Low", "Medium", "High", "Medium")
status_factor <- factor(status)
status_factor
```

17.1.1 Specifying Levels Explicitly

```
status_factor <- factor(status, levels = c("Low", "Medium", "High"))
```

Explicit levels ensure correct ordering and interpretation.

Ordered vs Unordered Factors

Some categorical variables have a natural order.

```
rating <- factor(c("Poor", "Good", "Excellent"),
                 levels = c("Poor", "Good", "Excellent"),
                 ordered = TRUE)
```

Ordered factors are especially important in ordinal data analysis.

Common Problems with Factors

- Automatic conversion from character to factor
- Unwanted level ordering
- Levels with no observations
- Inconsistent labels (e.g., spelling or case)

The `forcats` package is designed to address these issues cleanly.

Installing and Loading forcats

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

When using `library(tidyverse)`, `forcats` is loaded automatically.

Changing the Order of Factor Levels

Reordering by Frequency

```
responses <- factor(c("Yes", "No", "Yes", "Yes", "No"))
fct_infreq(responses)
```

This is particularly useful for bar plots.

Reordering by Another Variable

```
data <- data.frame(
group = factor(c("A", "A", "B", "B")),
score = c(10, 15, 20, 18)
)

data$group <- fct_reorder(data$group, data$score, .fun = mean)
```

This ensures meaningful ordering in summaries and plots.

Renaming Factor Levels

```
levels(gender)

gender <- fct_recode(gender,
                     M = "Male",
                     F = "Female"
)
```

Renaming improves clarity without altering the underlying data.

Combining and Collapsing Levels

Collapsing Levels

```
region <- factor(c("North", "South", "East", "West"))

region2 <- fct_collapse(region,
                        Urban = c("North", "East"),
                        Rural = c("South", "West")
)
```

17.1.2 Lump Infrequent Levels

```
categories <- factor(c("A", "B", "C", "A", "D", "E"))
fct_lump(categories, n = 3)
```

This is useful when dealing with high-cardinality categorical variables.

Dropping Unused Levels

```
f <- factor(c("A", "B"), levels = c("A", "B", "C"))
fct_drop(f)
```

Unused levels can cause confusion in analysis and plots.

Handling Missing Values in Factors

```
f <- factor(c("Yes", NA, "No"))
fct_na_value_to_level(f, level = "Missing")
```

Explicitly labeling missing categories improves transparency.

Factors in Data Analysis Workflows

Factors are often manipulated within `dplyr` pipelines.

```
library(dplyr)

data %>%
  mutate(
    group = fct_relevel(group, "B", "A"),
    group = fct_recode(group, Control = "A", Treatment = "B")
  )
```

This approach ensures consistency across analysis steps.

Factors and Visualization

Factor levels directly affect plot ordering.

```
library(ggplot2)

ggplot(data, aes(x = group, y = score)) +
  geom_boxplot()
```

Reordering factors before plotting leads to clearer and more interpretable graphics.

Best Practices for Working with Factors

- Convert character variables to factors intentionally
- Always inspect levels using `levels()`
- Explicitly set reference levels in modeling
- Use `forcats` functions instead of manual recoding
- Drop unused levels regularly

Summary

This chapter introduced **factors and categorical data handling using `forcats`**. You learned how to:

- Understand factors and their role in statistics
- Create ordered and unordered factors
- Reorder, rename, and collapse levels
- Handle missing and infrequent categories
- Integrate factor handling into tidyverse workflows

Mastering factors is essential for accurate statistical analysis, effective visualization, and meaningful interpretation of categorical data.

18 Introduction to Statistics

18.1 Introduction

Statistics is the science of **collecting, organizing, analyzing, interpreting, and presenting data**. In modern data-driven disciplines, statistics provides the theoretical foundation that guides how data is summarized, modeled, and used for decision-making. In this course, statistics and R are taught together so that **statistical thinking is reinforced through computation and real data examples**.

This chapter introduces the **core ideas of statistics** that are essential before studying probability theory and statistical inference. The focus here is on understanding what statistics is, why it is needed, and how data is classified and summarized. Detailed probability concepts will be covered in the next chapter, followed by statistical inference.

18.2 What Is Statistics?

Statistics can be broadly defined as the discipline concerned with:

- Designing data collection methods
- Organizing and summarizing data
- Analyzing patterns and relationships
- Drawing conclusions under uncertainty

Rather than dealing with certainty, statistics helps us **reason from data**, especially when variability and randomness are present.

18.3 Role of Statistics in Data Analysis

In data analysis workflows, statistics plays several critical roles:

- Understanding the structure and nature of data
- Summarizing large datasets into meaningful quantities
- Comparing groups and identifying patterns
- Supporting evidence-based decisions

In R-based analysis, statistical concepts guide the choice of functions, models, and visualizations.

18.4 Branches of Statistics

Statistics is traditionally divided into two main branches:

- **Descriptive Statistics** – concerned with summarizing and describing data
- **Inferential Statistics** – concerned with drawing conclusions about populations based on samples

This chapter focuses primarily on descriptive statistics. Inferential statistics will be discussed in detail in a later chapter.

18.5 Data and Its Types

Before applying statistical methods, it is essential to understand the nature of the data.

18.5.1 Qualitative and Quantitative Data

- **Qualitative (Categorical) Data**: Describes attributes or categories
  - Examples: Gender, blood group, treatment group
- **Quantitative (Numerical) Data**: Represents numerical measurements
  - Examples: Height, weight, income, age

18.6 Levels of Measurement

Data can be classified based on levels of measurement:

- **Nominal**: Categories without order (e.g., gender, blood group)
- **Ordinal**: Categories with order but no fixed scale (e.g., satisfaction level)
- **Interval**: Numeric scale without a true zero (e.g., temperature in Celsius)
- **Ratio**: Numeric scale with a true zero (e.g., weight, income)

Understanding measurement levels helps determine appropriate statistical methods.

18.7 Population and Sample

- A **population** is the entire set of units of interest
- A **sample** is a subset of the population used for analysis

Because studying entire populations is often impractical, statistical analysis typically relies on samples to make informed conclusions.

18.8 Variables and Observations

- A **variable** is a characteristic that can take different values
- An **observation** is a single recorded instance of all variables for one unit

In R, variables are usually represented as columns and observations as rows in a data frame.

18.9 Descriptive Statistics

Descriptive statistics summarize and organize data to make it understandable.

18.9.1 Measures of Central Tendency

- **Mean** – arithmetic average
- **Median** – middle value
- **Mode** – most frequent value

```
x <- c(10, 15, 20, 25, 30)
mean(x)
median(x)
```

18.9.2 Measures of Dispersion

Dispersion measures the spread of data.

- **Range** – difference between maximum and minimum
- **Variance** – average squared deviation from the mean
- **Standard Deviation** – square root of variance

```
range(x)
var(x)
sd(x)
```

18.10 Shape of a Distribution

The distribution of data describes how values are spread.

Common shapes include:

- Symmetric distribution
- Positively skewed distribution
- Negatively skewed distribution

Understanding distribution shape is important for selecting appropriate statistical methods.

18.11 Graphical Representation of Data

Graphs are a powerful way to summarize data visually.

Common graphical tools include:

- Bar charts for categorical data
- Histograms for numerical data
- Boxplots for comparing distributions

```
hist(x)
boxplot(x)
```

Visualization complements numerical summaries and often reveals patterns not immediately obvious.

18.12 Data Summaries Using R

R provides several functions for quick descriptive summaries.

```
summary(x)
```

For data frames:

```
data <- data.frame(Age = c(20, 22, 25, 30))
summary(data)
```

18.13 Importance of Exploratory Analysis

Before applying probability models or inferential methods, it is essential to explore the data thoroughly. Exploratory analysis helps:

- Detect errors or anomalies
- Understand variability
- Identify potential relationships

This exploratory mindset is central to good statistical practice.

18.14 Statistics, Probability, and Inference

Statistics, probability, and inference are closely connected:

- **Statistics** describes and explores data
- **Probability** models randomness
- **Inference** draws conclusions using probability

The next chapter will introduce **probability theory**, which provides the mathematical framework for statistical inference.

18.15 Summary

This chapter introduced the foundations of statistics. You learned about:

- The meaning and role of statistics
- Types and levels of data
- Population and sample concepts
- Variables and observations
- Descriptive statistics and data summaries
- The importance of exploratory analysis

These concepts form the groundwork for the study of **probability** and **statistical inference**, which will be covered in the following chapters.

19 Probability

19.1 Introduction

Probability provides the **mathematical framework for quantifying uncertainty**. While statistics focuses on describing and analyzing observed data, probability deals with the underlying randomness that generates the data. Together, probability and statistics form the foundation of modern data analysis, inference, and decision-making.

This chapter introduces the **core concepts of probability theory** required for statistical analysis. The emphasis is on clear definitions, intuitive understanding, and simple illustrations, with light use of R to reinforce concepts. Statistical inference based on probability will be covered in the next chapter.

19.2 Random Experiments

A **random experiment** is a process that produces an outcome which cannot be predicted with certainty, even if the experiment is repeated under identical conditions.

Examples include:

- Tossing a coin
- Rolling a die
- Selecting a patient randomly from a population
- Measuring response time in an experiment

The defining feature of a random experiment is **uncertainty in outcomes**.

19.3 Sample Space

The **sample space**, denoted by S, is the set of all possible outcomes of a random experiment.

Examples:

- Coin toss:
  $S = \{H, T\}$

- Die roll:
  $S = \{1, 2, 3, 4, 5, 6\}$

In R, a sample space can be represented using vectors:

```
sample_space <- c("H", "T")
sample_space
```

19.4 Events

An **event** is any subset of the sample space. Events may consist of:

- A single outcome (simple event)
- Multiple outcomes (compound event)

Example (die roll):

- Event A = rolling an even number
  $A = \{2, 4, 6\}$

```
A <- c(2, 4, 6)
```

19.5 Types of Events

19.5.1 Mutually Exclusive Events

Two events are **mutually exclusive** if they cannot occur at the same time.

Example:

- Rolling a 2 and rolling a 5 on a single die

19.5.2 Exhaustive Events

A set of events is **exhaustive** if at least one of them must occur.

Example:

- Even number or odd number in a die roll

19.6 Classical Definition of Probability

If a random experiment has a finite number of equally likely outcomes, the probability of an event A is defined as:

$$P(A) = \frac{\text{Number of favorable outcomes}}{\text{Total number of outcomes}}$$

Example:

Probability of rolling an even number on a fair die:

$$P(A) = \frac{3}{6} = 0.5$$

19.7 Axiomatic Definition of Probability

Modern probability theory is based on three axioms proposed by **Kolmogorov**:

1. For any event A, $P(A) \geq 0$
2. $P(S) = 1$
3. For mutually exclusive events $A_1, A_2, ...$:
   $P(A_1 \cup A_2 \cup ...) = \sum P(A_i)$

These axioms provide a rigorous mathematical foundation for probability.

19.8 Probability of Complementary Events

The **complement** of an event A, denoted by A^c, consists of all outcomes not in A.

$$P(A^c) = 1 - P(A)$$

Example:

If $P(A) = 0.7$, then:

$$P(A^c) = 0.3$$

19.9 Addition Law of Probability

For any two events A and B:

$$P(A \cup B) = P(A) + P(B) - P(A \cap B)$$

If A and B are mutually exclusive:

$$P(A \cup B) = P(A) + P(B)$$

19.10 Conditional Probability

The **conditional probability** of event A given that event B has occurred is defined as:

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)}, \quad P(B) > 0$$

Conditional probability allows us to update probabilities based on additional information.

19.11 Independence of Events

Two events A and B are said to be **independent** if the occurrence of one does not affect the probability of the other.

$$P(A \cap B) = P(A)P(B)$$

Independence is a strong assumption and must be justified carefully in practice.

19.12 Bayes' Theorem

Bayes' theorem provides a way to reverse conditional probabilities.

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)}$$

Bayes' theorem is fundamental in medical testing, machine learning, and decision theory.

19.13 Random Variables

A **random variable** is a function that assigns a numerical value to each outcome in the sample space.

- **Discrete random variables** take countable values
- **Continuous random variables** take values over an interval

Examples include:

- Number of heads in coin tosses
- Height or weight of individuals

19.14 Probability Distributions

A **probability distribution** describes how probabilities are assigned to values of a random variable.

Examples:

- Discrete: Bernoulli, Binomial, Poisson
- Continuous: Uniform, Normal, Exponential

Detailed study of distributions will appear in later chapters.

19.15 Expectation and Variance

For a random variable X:

- **Expected value** measures the long-run average
- **Variance** measures variability around the mean

These quantities play a central role in statistical modeling and inference.

19.16 Law of Large Numbers (Conceptual)

The **Law of Large Numbers** states that as the number of trials increases, the sample mean converges to the expected value.

This principle explains why probabilities manifest as stable long-run frequencies.

19.17 Probability and Simulation in R

R can be used to simulate random experiments and illustrate probability concepts.

```
set.seed(123)
coin_tosses <- sample(c("H", "T"), size = 1000, replace = TRUE)
mean(coin_tosses == "H")
```

Simulation provides intuition when analytical solutions are complex.

19.18 Common Misconceptions in Probability

- Confusing independence with mutual exclusivity
- Misinterpreting conditional probabilities
- Assuming short-term frequencies must match probabilities
- Ignoring base rates in real-world problems

Awareness of these pitfalls is essential for sound reasoning.

19.19 Summary

This chapter introduced the fundamental ideas of **probability theory**. You learned about:

- Random experiments, sample spaces, and events
- Definitions and axioms of probability
- Conditional probability and independence
- Bayes' theorem
- Random variables and probability distributions
- The role of simulation in understanding probability

These concepts form the mathematical backbone for **statistical inference**, which will be developed in the next chapter.

20 Statistical Inference

20.1 Introduction

Statistical inference is the process of **drawing conclusions about a population based on information obtained from a sample**. Because populations are often too large or impractical to study in full, inference provides a principled way to make decisions under uncertainty using probability theory.

Building on the foundations of statistics and probability introduced in the previous chapters, this chapter presents the **core framework of statistical inference**. The emphasis is on understanding concepts, assumptions, and interpretation, with supporting examples in R. The goal is to develop sound inferential reasoning rather than mechanical application of tests.

20.2 Population, Sample, and Parameters

- A **population** is the complete set of units of interest
- A **sample** is a subset selected from the population
- A **parameter** is a numerical characteristic of the population (e.g., mean, variance)
- A **statistic** is a numerical summary computed from the sample

Statistical inference uses sample statistics to learn about unknown population parameters.

20.3 Sampling Distributions

A **sampling distribution** describes the distribution of a statistic over repeated samples drawn from the same population.

For example, if many samples are taken and the sample mean is computed each time, the distribution of these means forms the sampling distribution of the sample mean.

Sampling distributions are fundamental because they quantify **sampling variability**, which underlies all inferential procedures.

20.4 The Central Limit Theorem

The **Central Limit Theorem (CLT)** states that, for a sufficiently large sample size, the sampling distribution of the sample mean is approximately normal, regardless of the population distribution, provided the variance is finite.

Key implications:

- Normal-based inference is widely applicable
- Sample size plays a critical role in inference
- Variability decreases as sample size increases

20.5 Point Estimation

A **point estimator** provides a single numerical estimate of a population parameter.

Common examples include:

- Sample mean as an estimator of population mean
- Sample proportion as an estimator of population proportion

```
x <- c(10, 12, 15, 14, 13)
mean(x)
```

Good estimators are typically unbiased and have low variability.

20.6 Interval Estimation and Confidence Intervals

A **confidence interval** provides a range of plausible values for a population parameter.

A 95% confidence interval means that, in repeated sampling, approximately 95% of such intervals will contain the true parameter.

```
x <- c(10, 12, 15, 14, 13)
t.test(x)$conf.int
```

Confidence intervals convey both estimation and uncertainty.

20.7 Hypothesis Testing Framework

Hypothesis testing provides a structured method for evaluating claims about population parameters.

The framework involves:

- **Null hypothesis (H_0)**: a statement of no effect or no difference
- **Alternative hypothesis (H_1)**: a statement contradicting H_0
- A test statistic
- A decision rule based on probability

20.8 Test Statistics

A **test statistic** summarizes the evidence in the data against the null hypothesis. Its distribution under the null hypothesis is known or approximated.

Examples include:

- z-statistic
- t-statistic
- chi-square statistic
- F-statistic

20.9 p-values

The **p-value** is the probability, under the null hypothesis, of observing a result at least as extreme as the one obtained.

Small p-values indicate strong evidence against the null hypothesis.

```
t.test(x)$p.value
```

Correct interpretation of p-values is essential for valid conclusions.

20.10 Significance Level

The **significance level**, denoted by , is the probability of rejecting a true null hypothesis.

Common choices include:

- $= 0.05$
- $= 0.01$

The significance level should be chosen before examining the data.

20.11 Types of Errors

Two types of errors may occur in hypothesis testing:

- **Type I Error**: Rejecting a true null hypothesis
- **Type II Error**: Failing to reject a false null hypothesis

There is a trade-off between these errors, influenced by sample size and test design.

20.12 Power of a Test

The **power** of a statistical test is the probability of correctly rejecting a false null hypothesis.

Higher power is desirable and can be increased by:

- Increasing sample size
- Reducing variability
- Choosing an appropriate test

20.13 Common Statistical Tests

Some widely used inferential procedures include:

- One-sample and two-sample t-tests
- Paired t-test
- Chi-square tests for independence
- Analysis of variance (ANOVA)

Detailed treatment of these tests will be provided in subsequent chapters.

20.14 Assumptions in Statistical Inference

Inferential methods rely on assumptions such as:

- Random sampling
- Independence of observations
- Normality of distributions (in some tests)
- Homogeneity of variance

Violating assumptions can lead to invalid conclusions.

20.15 Statistical Inference Using R

R provides built-in functions for inference.

```
t.test(x)
```

While software performs calculations automatically, understanding the underlying assumptions and interpretation remains essential.

20.16 Misinterpretations and Common Pitfalls

- Confusing statistical significance with practical importance
- Treating p-values as probabilities of hypotheses
- Ignoring assumptions
- Overreliance on automated output

Critical thinking is essential when applying inferential methods.

20.17 Inference in the Data Analysis Workflow

Statistical inference is typically applied after:

1. Data collection
2. Data cleaning and transformation
3. Exploratory data analysis

Inference should never be performed blindly without understanding the data.

20.18 Summary

This chapter introduced the principles of **statistical inference**. You learned about:

- Sampling distributions and the Central Limit Theorem
- Point and interval estimation
- Hypothesis testing framework
- p-values, significance levels, and errors
- Power and assumptions

Statistical inference provides the tools needed to make evidence-based conclusions from data, forming the core of statistical reasoning and applied data analysis.

21 Regression Analysis

21.1 Introduction

Regression analysis is one of the most important tools in statistics and data science. It is used to **study the relationship between a response (dependent) variable and one or more explanatory (independent) variables**. Regression allows us to explain variation, make predictions, and assess the effect of one variable on another.

In this chapter, we introduce the **fundamental ideas of regression analysis**, with an emphasis on interpretation, assumptions, and practical implementation using R. The focus is on understanding regression as a statistical model rather than merely a computational technique.

21.2 Relationship Between Variables

In many real-world problems, variables are related to each other. For example:

- Income may depend on education and experience
- Blood pressure may depend on age and lifestyle
- Sales may depend on price and advertising

Regression provides a formal framework to quantify such relationships.

21.3 Simple Linear Regression

Simple linear regression models the relationship between two variables:

- One response variable Y
- One explanatory variable X

The model is written as:

$$Y = \beta_0 + \beta_1 X + \varepsilon$$

where: - β_0 is the intercept - β_1 is the slope - ε is the random error term

---

21.4 Interpretation of Regression Coefficients

- **Intercept** (β_0): Expected value of Y when $X = 0$
- **Slope** (β_1): Expected change in Y for a one-unit increase in X

Interpretation should always be done in the context of the problem.

21.5 Fitting a Simple Linear Regression Model in R

```
model <- lm(mpg ~ wt, data = mtcars)
summary(model)
```

The `lm()` function fits linear regression models using the method of least squares.

21.6 Least Squares Principle

The method of **least squares** estimates regression coefficients by minimizing the sum of squared residuals:

$$\sum (Y_i - \hat{Y}_i)^2$$

This ensures the best linear fit in terms of squared error.

21.7 Fitted Values and Residuals

- **Fitted values** are the predicted values $\hat{Y}$
- **Residuals** are the differences $Y - \hat{Y}$

```
fitted(model)
residuals(model)
```

Residuals play a crucial role in diagnosing model adequacy.

21.8 Assumptions of Linear Regression

Classical linear regression relies on the following assumptions:

- Linearity of relationship
- Independence of errors
- Constant variance (homoscedasticity)
- Normality of errors (for inference)

Violations of these assumptions can affect the validity of conclusions.

21.9 Graphical Diagnostics

Diagnostic plots help assess model assumptions.

```
plot(model)
```

Common diagnostic plots include:

- Residuals vs fitted values
- Normal Q–Q plot
- Scale–location plot
- Residuals vs leverage

21.10 Coefficient of Determination (R^2)

The **coefficient of determination**, R^2, measures the proportion of variability in the response variable explained by the model.

$$R^2 = 1 - \frac{\text{Residual Sum of Squares}}{\text{Total Sum of Squares}}$$

Higher values of R^2 indicate better explanatory power, but should not be used alone to judge model quality.

21.11 Hypothesis Testing in Regression

Regression models allow hypothesis testing for coefficients.

Typical hypotheses:

- $H_0 : \beta_1 = 0$
- $H_1 : \beta_1 \neq 0$

The test assesses whether the explanatory variable has a statistically significant effect on the response.

21.12 Confidence Intervals for Regression Coefficients

```
confint(model)
```

Confidence intervals provide a range of plausible values for regression parameters.

21.13 Multiple Linear Regression

When more than one explanatory variable is included, the model becomes:

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p + \varepsilon$$

Multiple regression allows the effect of each variable to be studied **while controlling for others**.

21.14 Fitting Multiple Regression in R

```
model2 <- lm(mpg ~ wt + hp, data = mtcars)
summary(model2)
```

Interpretation of coefficients must consider the presence of other predictors.

21.15 Model Comparison

Models can be compared using:

- Adjusted R^2
- Analysis of variance (ANOVA)

```
anova(model, model2)
```

21.16 Prediction Using Regression Models

Regression models can be used for prediction.

```
new_data <- data.frame(wt = 3)
predict(model, new_data)
```

Prediction intervals account for uncertainty in future observations.

21.17 Limitations of Regression

- Correlation does not imply causation
- Extrapolation outside data range can be misleading
- Omitted variables can bias estimates
- Multicollinearity affects interpretation

Careful model building and domain knowledge are essential.

21.18 Regression in the Data Analysis Workflow

Regression analysis is typically performed after:

1. Exploratory data analysis
2. Data transformation and cleaning
3. Checking assumptions

Regression results should always be interpreted in context.

21.19 Summary

This chapter introduced **regression analysis** as a core statistical modeling tool. You learned about:

- Simple and multiple linear regression
- Interpretation of regression coefficients
- Least squares estimation
- Model assumptions and diagnostics
- Hypothesis testing and confidence intervals
- Prediction and model comparison

Regression forms the foundation for many advanced statistical and machine learning methods and is a cornerstone of applied data analysis.

22 Time Series Analysis

22.1 Introduction

Time series analysis deals with data observed **sequentially over time**. Such data arise naturally in economics, finance, climatology, public health, industrial processes, and many other domains. Unlike cross-sectional data, time series observations are often **dependent**, meaning that values at one time point are related to past values.

This chapter introduces the **foundations of time series analysis**, focusing on concepts, components, and basic modeling approaches. Emphasis is placed on understanding temporal structure, preparing data for analysis, and applying core methods using R. Advanced models and forecasting techniques can be built upon this foundation.

22.2 What Is a Time Series?

A **time series** is a sequence of observations recorded at successive points in time, usually at equally spaced intervals.

Examples include:

- Monthly rainfall totals
- Daily stock prices
- Hourly temperature readings
- Annual disease incidence counts

Time series analysis seeks to understand the underlying structure and generate reliable forecasts.

22.3 Types of Time Series

Time series can be classified based on frequency and structure:

- **Discrete-time series**: observed at fixed intervals (daily, monthly, yearly)
- **Continuous-time series**: observed continuously (often approximated discretely)
- **Univariate time series**: single variable over time
- **Multivariate time series**: multiple variables observed simultaneously

---

22.4 Time Series Objects in R

R provides dedicated classes for time series data.

22.4.1 The `ts` Class

```
sales <- c(120, 135, 150, 160, 170, 180)
sales_ts <- ts(sales, start = c(2020, 1), frequency = 12)
sales_ts
```

The `frequency` argument specifies the number of observations per unit time (e.g., 12 for monthly data).

22.5 Plotting Time Series

Visual inspection is the first step in time series analysis.

```
plot(sales_ts, main = "Monthly Sales Time Series",
     ylab = "Sales", xlab = "Time")
```

Plots help identify trends, seasonality, and irregular behavior.

22.6 Components of a Time Series

A time series is often decomposed into the following components:

- **Trend (T)**: long-term movement
- **Seasonal (S)**: repeating patterns at fixed intervals
- **Cyclical (C)**: long-term oscillations
- **Irregular (I)**: random variation

Understanding these components is essential for modeling and forecasting.

22.7 Additive and Multiplicative Models

Two common decomposition models are:

- **Additive model**:
  $Y_t = T_t + S_t + I_t $
- **Multiplicative model**:
  $Y_t = T_t \times S_t \times I_t $

The choice depends on whether seasonal variation is constant or proportional to the level of the series.

22.8 Time Series Decomposition

R provides tools for decomposing time series.

```
decomp <- decompose(sales_ts)
plot(decomp)
```

Decomposition separates the observed series into its components, aiding interpretation.

22.9 Stationarity

A key concept in time series analysis is **stationarity**.

A time series is said to be stationary if its statistical properties (mean, variance, autocorrelation) do not change over time.

Many time series models assume stationarity.

22.10 Achieving Stationarity

Non-stationary series can often be transformed into stationary ones using:

- Differencing
- Logarithmic transformation
- Removing trend and seasonality

```
diff_sales <- diff(sales_ts)
plot(diff_sales)
```

22.11 Autocorrelation and Partial Autocorrelation

22.11.1 Autocorrelation Function (ACF)

The ACF measures correlation between observations separated by different time lags.

```
acf(sales_ts)
```

22.11.2 Partial Autocorrelation Function (PACF)

The PACF measures correlation between observations after removing intermediate effects.

```
pacf(sales_ts)
```

ACF and PACF plots are essential tools for model identification.

22.12 Basic Time Series Models

22.12.1 Moving Average Models

Moving average models express the current value as a function of past random shocks.

22.12.2 Autoregressive Models

Autoregressive (AR) models express the current value as a function of its past values.

22.12.3 ARMA and ARIMA Models

- **ARMA**: combines AR and MA (stationary series)
- **ARIMA**: includes differencing to handle non-stationarity

Detailed treatment of these models can be developed in advanced chapters.

22.13 Fitting an ARIMA Model in R

```
model_arima <- arima(sales_ts, order = c(1, 1, 1))
model_arima
```

Model selection is guided by diagnostics and information criteria.

22.14 Forecasting

Time series models are commonly used for forecasting future values.

```
library(forecast)
forecast_values <- forecast(model_arima, h = 6)
plot(forecast_values)
```

Forecasts should always be accompanied by measures of uncertainty.

22.15 Model Diagnostics

After fitting a model, residuals should be examined.

```
acf(residuals(model_arima))
```

Residuals should resemble white noise if the model is adequate.

22.16 Time Series in the Data Analysis Workflow

Time series analysis typically involves:

1. Visualization and exploration
2. Decomposition
3. Checking stationarity
4. Model identification
5. Model estimation
6. Diagnostic checking
7. Forecasting

This systematic approach improves reliability and interpretability.

22.17 Limitations and Challenges

- Structural breaks
- Missing observations
- Nonlinear patterns
- External shocks

Time series models should be used with caution and domain knowledge.

22.18 Summary

This chapter introduced the foundations of **time series analysis**. You learned about:

- Time series concepts and types
- Time series objects and plotting in R
- Components and decomposition
- Stationarity and differencing
- Autocorrelation and partial autocorrelation
- Basic ARIMA modeling and forecasting

Time series analysis is a powerful tool for understanding temporal data and forms the basis for advanced forecasting and dynamic modeling techniques.

23 Classification

23.1 Introduction

Classification is a fundamental task in statistics, machine learning, and data science, where the objective is to **assign observations to predefined categories or classes** based on their features. Unlike regression, which predicts continuous outcomes, classification deals with **categorical response variables**.

Classification problems arise across many domains, including medicine (disease diagnosis), finance (credit risk), marketing (customer segmentation), and text analytics (spam detection). This chapter introduces the **conceptual foundations of classification**, common types of classification problems, and the general workflow used in classification analysis. Detailed algorithms and implementations will be covered in subsequent chapters.

23.2 What Is Classification?

Classification is a supervised learning task in which:

- The response variable (target) is categorical
- Each observation belongs to one of a finite number of classes
- A model is trained using labeled data

Examples include:

- Classifying emails as spam or not spam
- Diagnosing patients as diseased or healthy
- Predicting customer churn (yes/no)

23.3 Types of Classification Problems

23.3.1 Binary Classification

Binary classification involves two possible classes.

Examples:

- Pass / Fail
- Positive / Negative
- Yes / No

Binary classification is the most common and forms the basis for many advanced methods.

23.3.2 Multiclass Classification

Multiclass classification involves more than two classes.

Examples:

- Blood group classification
- Handwritten digit recognition
- Product category prediction

23.3.3 Multilabel Classification

In multilabel classification, an observation can belong to **multiple classes simultaneously**.

Example:

- Tagging news articles with multiple topics

23.4 Features and Class Labels

- **Features (predictors)** are variables used to make predictions
- **Class labels** are the categorical outcomes

In R, features are usually stored as columns in a data frame, while the class label is represented as a factor.

```r
data <- data.frame(
  Age = c(25, 40, 30, 50),
  Income = c(30000, 60000, 45000, 80000),
  Purchase = factor(c("No", "Yes", "No", "Yes"))
)
```

23.5 Classification vs Regression

Aspect	Classification	Regression
Target variable	Categorical	Continuous
Output	Class or probability	Numeric value
Examples	Disease status	Blood pressure

Understanding this distinction is crucial for selecting appropriate models.

23.6 Decision Boundary Concept

A classification model learns a **decision boundary** that separates different classes in the feature space.

- Linear boundaries are simple and interpretable
- Nonlinear boundaries can capture complex patterns

The choice of boundary affects accuracy and generalization.

23.7 Common Classification Algorithms (Overview)

Some widely used classification methods include:

- Logistic Regression
- k-Nearest Neighbors (k-NN)
- Decision Trees
- Naive Bayes
- Support Vector Machines (SVM)

Each method has different assumptions, strengths, and limitations.

23.8 Logistic Regression as a Classification Model

Although called regression, **logistic regression** is a classification method used for binary outcomes.

- Models the probability of class membership
- Uses a logistic (sigmoid) function
- Outputs probabilities between 0 and 1

```
model <- glm(Purchase ~ Age + Income,
             data = data,
             family = binomial)
summary(model)
```

Logistic regression is widely used due to its interpretability.

23.9 Probabilistic Interpretation

Most classification models estimate **class probabilities**.

```
predict(model, type = "response")
```

A decision threshold (commonly 0.5) is then applied to assign class labels.

23.10 Training and Testing Data

To evaluate classification models fairly, data is typically split into:

- Training set – used to build the model
- Test set – used to evaluate performance

This helps assess how well the model generalizes to unseen data.

23.11 Model Evaluation in Classification

Classification performance is evaluated using several metrics:

- Accuracy
- Sensitivity (Recall)
- Specificity
- Precision
- F1-score

The choice of metric depends on the problem context.

23.12 Confusion Matrix

A **confusion matrix** summarizes prediction results.

```
table(Actual = data$Purchase,
      Predicted = ifelse(predict(model, type = "response") > 0.5,
                         "Yes", "No"))
```

It provides insight into different types of classification errors.

23.13 Imbalanced Classes

In many real-world problems, one class may be much more frequent than others.

Examples:

- Fraud detection
- Rare disease diagnosis

Accuracy alone can be misleading in such cases, and alternative evaluation metrics are needed.

23.14 Assumptions and Challenges in Classification

Common challenges include:

- Overfitting
- Multicollinearity among predictors
- Noisy or irrelevant features
- Class imbalance

Understanding data characteristics is essential for effective classification.

23.15 Classification Workflow

A typical classification analysis follows these steps:

1. Data collection and cleaning
2. Feature selection and transformation
3. Model selection
4. Model training
5. Model evaluation
6. Interpretation and deployment

23.16 Applications of Classification

Classification methods are widely used in:

- Medical diagnosis and prognosis
- Credit scoring and risk assessment
- Marketing response prediction
- Text and document classification
- Image and signal recognition

23.17 Summary

This chapter introduced the fundamental concepts of **classification**. You learned about:

- The nature of classification problems
- Types of classification tasks
- Features and class labels
- Decision boundaries
- Common classification algorithms
- Model evaluation and challenges

Classification forms a core component of predictive analytics. Subsequent chapters will explore **specific classification algorithms and their implementation in R** in greater detail.

24 Clustering

24.1 Introduction

Clustering is an unsupervised learning technique used to group observations such that objects within the same group (cluster) are more similar to each other than to those in other groups. Unlike classification, clustering does not rely on predefined class labels. Instead, it seeks to **discover hidden structure and patterns** in data.

Clustering plays a key role in exploratory data analysis, market segmentation, bioinformatics, image analysis, and social science research. This chapter introduces the **conceptual foundations of clustering**, common clustering paradigms, and practical considerations for applying clustering methods in R.

24.2 What Is Clustering?

In clustering:

- The response variable is unknown or absent
- The algorithm groups data based on similarity or distance
- The goal is pattern discovery rather than prediction

Typical questions answered by clustering include:

- Are there natural groupings in the data?
- How many distinct patterns exist?
- Which observations are similar to each other?

Aspect	Clustering	Classification

24.3 Clustering vs Classification

Aspect	Clustering	Classification
Learning type	Unsupervised	Supervised
Class labels	Unknown	Known
Objective	Discover structure	Predict class
Evaluation	Subjective / internal	Objective / external

Clustering is often used **before classification** to understand data structure.

24.4 Similarity and Distance Measures

Clustering methods rely on measures of similarity or dissimilarity.

Common distance measures include:

- **Euclidean distance** – continuous variables
- **Manhattan distance** – absolute differences
- **Cosine similarity** – text and high-dimensional data

The choice of distance metric strongly influences clustering results.

24.5 Importance of Feature Scaling

Clustering algorithms are sensitive to scale.

```
scaled_data <- scale(data)
```

Without scaling, variables with larger ranges may dominate distance calculations.

24.6 Types of Clustering Methods

Clustering methods can be broadly classified into:

- Partition-based clustering
- Hierarchical clustering
- Density-based clustering

Each approach has different assumptions and use cases.

24.7 Partition-Based Clustering: k-Means

k-means clustering partitions data into (k) clusters by minimizing within-cluster variation.

Key characteristics:

- Requires pre-specifying the number of clusters
- Assumes roughly spherical clusters
- Efficient for large datasets

```
set.seed(123)
kmeans_model <- kmeans(scaled_data, centers = 3)
kmeans_model$cluster
```

24.8 Choosing the Number of Clusters

Selecting an appropriate number of clusters is a critical step.

Common approaches include:

- Elbow method
- Silhouette analysis
- Domain knowledge

```
# Elbow method illustration
wss <- sapply(1:10, function(k) kmeans(scaled_data, k)$tot.withinss)
plot(1:10, wss, type = "b")
```

24.9 Hierarchical Clustering

Hierarchical clustering builds a tree-like structure of clusters.

Two main approaches:

- **Agglomerative** – bottom-up approach
- **Divisive** – top-down approach

```
dist_mat <- dist(scaled_data)
hc <- hclust(dist_mat)
plot(hc)
```

24.10 Linkage Methods

Hierarchical clustering uses linkage criteria to define cluster distance:

- Single linkage
- Complete linkage
- Average linkage
- Ward's method

Different linkage methods can lead to very different clustering structures.

24.11 Cutting the Dendrogram

```
clusters <- cutree(hc, k = 3)
clusters
```

This assigns each observation to a cluster.

24.12 Density-Based Clustering (Overview)

Density-based methods identify clusters as regions of high density.

Key ideas:

- Clusters can have arbitrary shapes
- Noise and outliers are explicitly identified

Examples include DBSCAN and OPTICS (covered in advanced chapters).

24.13 Evaluating Clustering Results

Clustering evaluation is challenging due to the absence of true labels.

Common internal measures include:

- Within-cluster sum of squares
- Silhouette width
- Cluster stability

Visualization also plays a key role in assessment.

24.14 Visualization of Clusters

```
plot(scaled_data, col = kmeans_model$cluster)
```

Dimensionality reduction techniques such as PCA are often used for visualization.

24.15 Practical Challenges in Clustering

- Choosing the number of clusters
- Sensitivity to initialization
- High-dimensional data
- Interpretability of clusters

Clustering results should always be interpreted cautiously and in context.

24.16 Applications of Clustering

Clustering is widely applied in:

- Customer and market segmentation
- Gene expression analysis
- Image and pattern recognition
- Social network analysis
- Exploratory survey analysis

24.17 Best Practices for Clustering

- Scale variables before clustering
- Try multiple clustering methods
- Validate results using multiple criteria
- Combine clustering with visualization
- Use domain knowledge for interpretation

24.18 Summary

This chapter introduced the foundations of **clustering analysis**. You learned about:

- The concept and purpose of clustering
- Differences between clustering and classification
- Distance measures and scaling
- k-means and hierarchical clustering
- Choosing the number of clusters
- Evaluating and visualizing clusters

Clustering is a powerful exploratory tool that often precedes predictive modeling and deeper statistical analysis.

25 k-Nearest Neighbors (k-NN)

25.1 Introduction

The **k-Nearest Neighbors (k-NN)** algorithm is one of the simplest and most intuitive classification methods. It is a **non-parametric, instance-based learning algorithm**, meaning that it does not assume an explicit statistical model for the data and does not involve a training phase in the traditional sense.

Instead, k-NN makes predictions by comparing a new observation to previously observed data points and assigning the most common class among its nearest neighbors. This chapter focuses on the conceptual understanding of k-NN, its assumptions, and its practical use in classification problems.

25.2 Basic Idea of k-NN

The central idea of k-NN is straightforward:

1. Choose a value for (k), the number of nearest neighbors
2. Measure the distance between a new observation and all training observations
3. Identify the (k) closest observations
4. Assign the class that occurs most frequently among these neighbors

The simplicity of this approach makes k-NN an excellent starting point for understanding classification algorithms.

25.3 Distance Measures

The notion of "nearness" depends on the distance metric used. Common distance measures include:

- **Euclidean distance** – most commonly used for continuous variables
- **Manhattan distance** – based on absolute differences
- **Minkowski distance** – a generalization of distance measures

For numeric features, Euclidean distance is typically used.

25.4 Choosing the Value of k

The choice of (k) has a significant impact on model performance:

- Small (k) values can lead to overfitting
- Large (k) values can lead to oversmoothing and underfitting

Selecting an appropriate value of (k) is often done using cross-validation.

25.5 Feature Scaling in k-NN

Because k-NN relies on distance calculations, **feature scaling is essential**.

```
scale()
```

Variables measured on larger scales can dominate distance computations if scaling is ignored.

25.6 k-NN Classification in R

The `class` package provides a simple implementation of k-NN.

```
library(class)

knn(train = train_data,
    test = test_data,
    cl = train_labels,
    k = 5)
```

Here:

- `train` contains predictor variables for training
- `test` contains predictor variables for testing
- `cl` contains class labels

25.7 Advantages of k-NN

- Simple and intuitive
- No explicit training phase
- Flexible decision boundaries

25.8 Limitations of k-NN

- Computationally expensive for large datasets
- Sensitive to noise and irrelevant features
- Requires careful choice of distance metric and (k)

25.9 Applications of k-NN

k-NN is commonly used in:

- Pattern recognition
- Recommendation systems
- Medical diagnosis (small datasets)
- Image classification

25.10 Summary

This chapter introduced the **k-Nearest Neighbors algorithm** as a simple yet powerful classification technique. You learned about:

- The intuition behind k-NN
- Distance measures and feature scaling
- Choosing the value of (k)
- Advantages and limitations

k-NN provides a foundation for understanding more complex classification algorithms.

26 Decision Trees

26.1 Introduction

Decision Trees are a popular and intuitive classification and regression method that model decision-making as a sequence of rules. They represent decisions using a tree-like structure, where internal nodes correspond to decision rules, branches represent outcomes of these rules, and terminal nodes (leaves) represent predicted classes.

Decision trees are widely used due to their **interpretability and flexibility**, making them particularly valuable in applied statistics, healthcare, and business analytics.

26.2 Structure of a Decision Tree

A decision tree consists of:

- **Root node** – the initial split
- **Internal nodes** – decision rules based on features
- **Branches** – outcomes of the rules
- **Leaf nodes** – final class labels

Each path from root to leaf represents a classification rule.

26.3 Splitting Criteria

Decision trees use impurity measures to decide how to split data:

- **Gini Index**
- **Entropy** and **Information Gain**

The goal is to create child nodes that are as homogeneous as possible.

26.4 Building Decision Trees in R

The `rpart` package is commonly used to build decision trees.

```
library(rpart)

model <- rpart(Class ~ ., data = data)
```

The formula interface specifies the response variable and predictors.

26.5 Tree Visualization

Decision trees can be visualized for interpretation.

```
plot(model)
text(model)
```

Visualization helps in understanding decision rules and variable importance.

26.6 Overfitting and Pruning

Decision trees can grow very large and overfit the data.

Pruning reduces tree complexity by removing branches that provide little predictive power.

```
printcp(model)
```

Pruning improves generalization performance.

26.7 Advantages of Decision Trees

- Easy to interpret and explain
- Handles both numeric and categorical variables
- Requires little data preparation

26.8 Limitations of Decision Trees

- Prone to overfitting
- Unstable with small data changes
- Often less accurate than ensemble methods

26.9 Applications of Decision Trees

Decision trees are widely used in:

- Medical decision-making
- Credit scoring
- Customer segmentation
- Risk analysis

26.10 Summary

This chapter introduced **decision trees** as an interpretable classification method. You learned about:

- Tree structure and splitting criteria
- Building and visualizing trees in R
- Overfitting and pruning
- Advantages and limitations

Decision trees form the basis for advanced ensemble methods such as random forests and boosting.

27 Model Evaluation and ROC Curves

27.1 Introduction

Evaluating the performance of a classification model is as important as building the model itself. A model that performs well on training data may fail to generalize to new, unseen data. **Model evaluation** provides quantitative tools to assess how well a classifier performs and to compare competing models.

This chapter introduces key evaluation metrics for classification models, with particular emphasis on **Receiver Operating Characteristic (ROC) curves** and **Area Under the Curve (AUC)**.

27.2 Confusion Matrix

A **confusion matrix** summarizes the relationship between actual and predicted class labels.

```
table(Actual, Predicted)
```

It forms the basis for many evaluation metrics.

27.3 Classification Metrics

Common performance measures include:

- **Accuracy** – proportion of correct predictions
- **Sensitivity (Recall)** – ability to detect positive cases
- **Specificity** – ability to detect negative cases
- **Precision** – proportion of predicted positives that are correct
- **F1-score** – harmonic mean of precision and recall

Each metric highlights different aspects of model performance.

27.4 Limitations of Accuracy

Accuracy can be misleading, especially when class distributions are imbalanced. In such cases, a classifier may achieve high accuracy by predicting the majority class while performing poorly on the minority class.

27.5 ROC Curve Concept

A **ROC curve** plots:

- True Positive Rate (Sensitivity)
- False Positive Rate ($1 -$ Specificity)

across different classification thresholds.

ROC curves illustrate the trade-off between sensitivity and specificity.

27.6 Area Under the Curve (AUC)

The **AUC** summarizes the ROC curve into a single value between 0 and 1.

- AUC = 0.5 indicates no discriminative power
- AUC = 1 indicates perfect classification

Higher AUC values indicate better overall model performance.

27.7 ROC Curves in R

The `pROC` package is commonly used for ROC analysis.

```
library(pROC)

roc_obj <- roc(actual, predicted_probabilities)
plot(roc_obj)
auc(roc_obj)
```

Predicted probabilities, not class labels, are required to construct ROC curves.

27.8 Comparing Models Using ROC Curves

ROC curves can be used to compare multiple classifiers on the same dataset.

A model with a consistently higher ROC curve and larger AUC is generally preferred.

27.9 Threshold Selection

Classification thresholds affect sensitivity and specificity.

ROC analysis helps identify thresholds that balance false positives and false negatives according to application needs.

27.10 Cross-Validation and Evaluation

Reliable model evaluation often involves **cross-validation**.

Cross-validation provides a more accurate estimate of generalization performance by repeatedly splitting data into training and testing sets.

27.11 Practical Considerations

- Choose evaluation metrics aligned with problem goals
- Use ROC and AUC for probabilistic classifiers
- Avoid evaluating models only on training data
- Interpret metrics in the context of the application

27.12 Summary

This chapter introduced methods for **evaluating classification models**, focusing on ROC curves and AUC. You learned about:

- Confusion matrices and classification metrics
- Limitations of accuracy
- ROC curves and their interpretation
- AUC as a summary measure
- Model comparison and threshold selection

Robust evaluation is essential for building reliable and trustworthy classification models.

28 Course Summary

28.1 Overview of the Course

This course, **R for Data Analysis: From Fundamentals to Advanced Applications**, was designed to provide a **comprehensive and integrated learning experience** that combines statistical theory, data analysis concepts, and practical implementation using R. Beginning from basic programming and data handling, the course gradually progresses to advanced statistical thinking and modern data science methods.

The emphasis throughout the course has been on developing **statistical reasoning**, **computational skills**, and **reproducible analysis practices**, enabling learners to confidently work with real-world data across academic, industrial, and research settings.

28.2 Learning Journey

The course follows a carefully structured progression, ensuring that each topic builds naturally on the previous ones.

28.2.1 Foundations in R Programming

The early part of the course focuses on building a strong foundation in R:

- Understanding the R and RStudio environment
- Core R syntax and programming principles
- Data types and data structures
- Writing clean, readable, and reproducible R code

These fundamentals ensure that learners are comfortable with R as a statistical programming language before moving to data analysis tasks.

28.2.2 Data Handling and Preparation

A major portion of the course is dedicated to **data preparation**, which is often the most time-consuming part of any data analysis project.

Key topics include:

- Importing data from multiple sources (CSV, Excel, SPSS, SAS)
- Tidy data principles
- Data reshaping and cleaning using `tidyr`
- Data manipulation using `dplyr`
- String handling with `stringr`
- Working with dates and times using `lubridate`
- Data transformation and feature engineering

These skills equip learners to transform raw data into analysis-ready datasets.

28.2.3 Visualization and Exploratory Analysis

Visualization is emphasized as a critical tool for understanding data.

The course covers:

- Base R plotting techniques
- Grammar of graphics using `ggplot2`
- Choosing appropriate visualizations for different data types
- Using plots for exploratory data analysis and communication

Learners develop the ability to uncover patterns, detect anomalies, and communicate insights effectively.

28.2.4 Statistical Foundations

Strong statistical foundations form the backbone of this course. The statistical core is organized into three coherent chapters:

- **Statistics** – concepts of data, variables, descriptive measures, and exploratory thinking
- **Probability** – randomness, events, random variables, and probability distributions
- **Statistical Inference** – estimation, confidence intervals, hypothesis testing, and decision-making

This structure ensures conceptual clarity and prepares learners for both classical statistical analysis and modern modeling techniques.

28.2.5 Unsupervised and Supervised Learning

The later part of the course introduces learners to **data-driven modeling techniques** used in data science.

Topics include:

- Clustering as an unsupervised learning approach
- Classification concepts and workflows
- k-Nearest Neighbors algorithm
- Decision Trees and rule-based models
- Model evaluation using confusion matrices, ROC curves, and AUC

The focus is on understanding **when and why** to use each method, along with their assumptions and limitations.

28.3 Skills Gained

By the end of this course, learners are expected to have developed the ability to:

- Write efficient and readable R code
- Manage, clean, and transform real-world datasets
- Perform exploratory data analysis using visual and numerical methods
- Apply statistical reasoning to data-driven problems
- Understand and implement basic machine learning techniques

- Evaluate and compare predictive models critically
- Communicate results clearly and responsibly

28.4 Applications and Relevance

The skills acquired in this course are directly applicable to:

- Academic research in statistics and related fields
- Clinical and healthcare data analysis
- Government and policy-oriented data analysis
- Industry roles in data science and analytics
- Further study in advanced statistics and machine learning

The integration of theory with R-based implementation ensures that learners are prepared for both **conceptual understanding and practical application**.

28.5 Concluding Remarks

This course aims to bridge the gap between **statistical theory and real-world data analysis**. By combining foundational concepts, modern data manipulation tools, and applied modeling techniques, it prepares learners to approach data problems with confidence, rigor, and ethical responsibility.

Learners completing this course are well-positioned to advance into specialized areas such as advanced statistical modeling, machine learning, time series analysis, and domain-specific data science applications.

References