LAB MANUAL

MAT1011: APPLIED STATISTICS

DEPARTMENT OF MATHEMATICS
SCHOOL OF ADVANCED SCIENCES
VIT-AP UNIVERSITY

VIT-AP UNIVERSITY

Contents

Preface vii

1 Introduction 1

2 Introduction to R Programming 3
- 2.1 Introduction to R . . . 3
- 2.2 Installing R and RStudio . . . 5
- 2.3 Installing R . . . 5
  - 2.3.1 Windows . . . 5
  - 2.3.2 Mac OS X . . . 6
  - 2.3.3 Linux . . . 6
- 2.4 Installing RStudio . . . 6
  - 2.4.1 Windows and Mac . . . 7
  - 2.4.2 Linux . . . 7
- 2.5 Overview of R Packages . . . 7
- 2.6 Installing R Packages . . . 7
  - 2.6.1 Example of Installing a Package . . . 7
  - 2.6.2 Loading an Installed Package . . . 8
- 2.7 Commonly Used Packages . . . 8
- 2.8 Basic Configuration Tips for RStudio . . . 8
- 2.9 Code Execution . . . 8
- 2.10 Packages . . . 8
- 2.11 Basic R Syntax . . . 9
- 2.12 Variables in R . . . 9
- 2.13 Data Types in R . . . 9
  - 2.13.1 Numeric . . . 9
  - 2.13.2 Character . . . 9
  - 2.13.3 Factor . . . 10
  - 2.13.4 Logical . . . 10
- 2.14 Introduction . . . 10
- 2.15 Arithmetic Operators . . . 10

2.15.1 Examples . . . 11
2.16 Relational Operators . . . 11
2.16.1 Examples . . . 12
2.17 Logical Operators . . . 12
2.17.1 Examples . . . 12
2.18 Vectors in R . . . 12
2.18.1 Example of a Numeric Vector . . . 13
2.18.2 Example of a Character Vector . . . 13
2.18.3 Integer . . . 13
2.18.4 Character . . . 13
2.18.5 Creating a Vector . . . 14
2.18.6 Class of Vector . . . 15
2.18.7 Adding Vectors of Different Types . . . 15
2.18.8 Accessing Elements of Vectors . . . 16
2.18.9 Creating Vector Using In-Built Functions . . . 16
2.18.10 Let's Try to Repeat Vectors . . . 16
2.18.11 Arithmetic Operators in Vectors in R . . . 17
2.19 Introduction to Data Frames in R . . . 18
2.19.1 Characteristics of Data Frames . . . 18
2.20 Steps For Creating Data Frames in R . . . 18
2.20.1 Step 1: Create a Data Frame of a Class in a School . . . 18
2.20.2 Step 2: Add the following line to our code . . . 19
2.20.3 Step 3: Use the `summary()` Function . . . 19
2.21 Inspecting Data Frames . . . 26
2.22 Extracting Specific Data from the Data Frame . . . 28
2.23 Conclusion . . . 30
2.24 Matrices in R . . . 30
2.24.1 Extracting Row/Column from Matrices . . . 32
2.24.2 Operations in Matrices . . . 32
2.24.3 Transposing a Matrix . . . 35
2.24.4 Common Matrix Operations in R . . . 36
2.24.5 Naming Matrix Rows and Columns . . . 36
2.25 Creating a Function . . . 37
2.26 Function Arguments . . . 37
2.26.1 Example with Default and Named Arguments . . . 38
2.27 Return Values . . . 38
2.28 Default Arguments . . . 38
2.29 Lazy Evaluation . . . 38
2.30 Anonymous Functions . . . 39
2.31 Scoping Rules . . . 39
2.32 Complete Example of a Function . . . 39

2.33 Conclusion . . . 40
2.34 Data Import and Export in R . . . 40
2.34.1 1. CSV Files . . . 40
2.34.2 2. Excel Files . . . 41
2.34.3 3. Text Files . . . 41
2.34.4 4. R Data Files . . . 41
2.34.5 5. Databases . . . 42
2.34.6 6. SPSS, SAS, and Stata Files . . . 42
2.35 Data Export in R . . . 42
2.35.1 1. CSV Files . . . 42
2.35.2 2. Excel Files . . . 43
2.35.3 3. Text Files . . . 43
2.35.4 4. R Data Files . . . 43
2.36 Importing Data from the Internet . . . 43
2.36.1 1. Reading Data from URLs . . . 43
2.36.2 2. JSON and APIs . . . 44
2.36.3 3. Web Scraping . . . 44
2.36.4 4. Google Sheets . . . 44
2.36.5 5. XML Data . . . 44
2.37 Conclusion . . . 45
2.38 Data Visualization . . . 45
2.39 For Loop in R . . . 72
2.39.1 Syntax of the For Loop . . . 72
2.39.2 Example 1: Basic For Loop . . . 72
2.39.3 Example 2: Calculating the Sum of a Vector . . . 73
2.39.4 Example 3: Iterating Over a List . . . 73
2.39.5 Example 4: Creating a Multiplication Table . . . 73
2.39.6 Example 5: Nested For Loops . . . 74
2.39.7 Example 6: Using a For Loop with Conditional Statements . . . 74
2.39.8 Summary . . . 75
2.40 While Loop in R . . . 75
2.40.1 Syntax of `while` Loop . . . 75
2.40.2 Characteristics of `while` Loop . . . 76
2.40.3 Examples of `while` Loop . . . 76
2.40.4 Important Notes . . . 78
2.40.5 Conclusion . . . 78
2.41 If-Else Statements in R . . . 78
2.41.1 Basic Structure . . . 78
2.41.2 Explanation of Components . . . 79
2.41.3 Extended If-Else Statement . . . 79

2.41.4 Examples . . . 79
2.41.5 Summary . . . 81
2.42 Working with Packages . . . 85
2.43 R Markdown . . . 85
2.43.1 Exercise . . . 87

3 Module 1: Descriptive Statistics and Probability 88
3.1 Measures of Central Tendency . . . 88
3.1.1 Individual/Ungrouped Data . . . 88
3.2 Grouped - Continuous data . . . 91
3.2.1 Exercises . . . 95
3.3 Measures of dispersion . . . 95
3.3.1 Discrete Data . . . 95
3.3.2 Continuous Data . . . 96
3.3.3 Exercises . . . 97

4 Module 2: Modelling with Probability distributions 98
4.1 Random Variables – Discrete & Continuous . . . 98
4.2 Probability Mass and Density Function . . . 99
4.3 Expected Value and Variance of Random Variable . . . 99
4.3.1 Properties of Expected Values . . . 100
4.4 Probability Models . . . 100
4.4.1 The Binomial Model . . . 100
4.4.2 The Poisson Model . . . 101
4.4.3 The Normal Model . . . 101

5 Module 3: Correlation and Regression Analysis 103
5.1 Correlation . . . 103
5.2 Objectives . . . 103
5.3 Results . . . 104
5.3.1 Correlation Coefficient . . . 104
5.3.2 Scatter Plot . . . 104
5.4 Manual calculation of Pearson Correlation . . . 104
5.5 Discussion . . . 106
5.6 Regression . . . 106
5.7 Objectives . . . 106
5.8 Materials and Methods . . . 106
5.8.1 Data Collection . . . 106
5.8.2 Linear Regression Analysis . . . 107
5.9 Results . . . 107
5.9.1 Descriptive Statistics . . . 107
5.9.2 Regression Coefficients . . . 107

5.9.3 Model Summary 108
5.10 Discussion 108
5.10.1 Interpretation of Coefficients 108
5.10.2 Model Fit 108
5.11 Manual Calculation of Regression Analysis 108

6 Module 4: Inference for Decision Making – I **111**
6.1 Hypothesis 111
6.1.1 Test for Proportion—Single Proportion 111
6.1.2 Difference of Proportions 111
6.1.3 Testing Mean – Single Sample (Z-Test and t-Test) . . . 111
6.1.4 Two-Sample Tests – Comparing Two Means 112
6.1.5 Test for Equality of Variance – F-Test 112
6.1.6 ANOVA: One-Way Analysis of Variance 112

7 Inference for Decision Making –II **114**
7.1 Non-parametric Tests 114
7.1.1 Chi-Square Test for Goodness of Fit 114
7.1.2 Chi-Square Test for Goodness of Fit Using Probability Distributions 114
7.1.3 Chi-Square Test of Independence of Attributes 115
7.1.4 Wilcoxon Signed Rank Test 115
7.1.5 Wald-Wolfowitz Run Test 116
7.1.6 Mann-Whitney U Test 116
7.1.7 Test for Normality 116

Contributors Name

Dr. S. Srinivas, Dean, School of Advanced Sciences

Dr. Vemula Ramakrishna Reddy, Head, Dept. of Mathematics

Dr. Komandla Mahipal Reddy

Dr. M. Phani Kumar

Dr. Santanu Mandal

Dr. M. Sudhakar

Dr. Sinuvasan

Dr. G. Siva

Preface

In today's data-driven world, the ability to analyze and interpret data is more critical than ever. Statistics provides the foundation for understanding data, while programming tools like R offer the computational power to perform complex analyses efficiently. This manual combines essential statistical concepts with hands-on R programming, offering students a comprehensive guide to both understanding and applying statistical techniques.

This manual is designed to introduce students to fundamental statistical methods and their implementation in R, enabling them to solve real-world problems using a practical, computational approach.

The first section focuses on descriptive statistics, covering measures of central tendency such as mean, median, and mode. These tools help summarize and interpret data, providing initial insights into datasets. R functions will be introduced to calculate these measures, allowing students to immediately see how statistical concepts translate into code.

We then explore probability theory, including key concepts like random events, conditional probability, and Bayes' theorem. With R, students will simulate probabilistic scenarios and learn how to model uncertainty using code. Bayes' theorem, in particular, will be implemented to demonstrate how new information updates probabilities in real time.

Next, the manual covers random variables and probability distributions, both discrete and continuous. In this section, students will learn how to work with common distributions (e.g., binomial, normal, Poisson) and use R to visualize and analyze these distributions. Simulations in R will help students gain a deeper understanding of how random variables behave and how distributions are used in practice.

Understanding relationships between variables is critical in data analysis, so we explore correlation and regression analysis. Using R, students will compute correlation coefficients and build regression models to investigate the strength and direction of relationships between variables. R's graphical capabilities will also be employed to create scatter plots, residual plots, and other visualizations, making the results more interpretable.

In the final part of the manual, we focus on decision-making through hypothesis testing. Various tests, including the z-test, t-test, F-test, and chi-square test, will be covered. Each test will be demonstrated using R, guiding students through hypothesis formulation, execution of the tests, and interpretation of results. This hands-on approach ensures that students learn not only the theory but also the practical skills necessary for conducting statistical tests using real data.

Throughout this manual, R programming is integrated seamlessly with the statistical topics covered. By the end of the course, students will have a solid understanding of both statistical theory and how to use R to implement these techniques effectively. The focus on R not only equips students with a valuable skill for the job market but also enables them to work on data-driven projects with confidence.

Each chapter contains code snippets, exercises, and case studies to reinforce learning, ensuring that students can apply their knowledge to solve real-world problems. We hope this manual serves as a valuable resource, enabling students to harness the power of statistics and R programming in their academic and professional endeavors.

This version of the preface highlights how R programming is used throughout the manual to teach statistical concepts. It emphasizes the practical, hands-on approach that students will gain by combining theory with code, preparing them for real-world data analysis tasks.

December 2024

1. Introduction

In an age dominated by data, the ability to analyze and interpret information has become a critical skill across various fields, including business, healthcare, social sciences, and engineering. Statistics provides the essential framework for transforming raw data into meaningful insights, enabling informed decision-making. As we navigate through complex datasets, the integration of statistical theory with programming skills has proven invaluable. Among the tools available, **R** stands out as a powerful programming language specifically designed for statistical computing and data analysis.

R is an open-source programming language that has gained immense popularity among statisticians and data scientists for its robust capabilities and extensive ecosystem. It is renowned for its ability to handle complex statistical analyses, create compelling visualizations, and manage data effectively. With a vast array of packages available—such as `dplyr` for data manipulation, `ggplot2` for data visualization, and `tidyverse` for streamlined data science workflows—R empowers users to perform sophisticated statistical analyses with relative ease. Its flexibility and adaptability make it an essential tool for anyone working with data.

This manual is structured to provide a comprehensive overview of fundamental statistical concepts while employing R programming throughout. We will explore a variety of topics that are crucial for statistical analysis, including:

- **Descriptive Statistics**: We begin by examining measures of central tendency—mean, median, and mode. These statistics summarize data sets, allowing us to draw initial insights about distributions. Through R, students will learn how to calculate and visualize these measures, gaining a deeper understanding of their significance.

- **Probability Theory**: Understanding probability is essential for making predictions based on data. This section introduces key concepts, including basic probability rules, conditional probability, and Bayes' theorem. By using R to simulate probabilistic scenarios, students will learn

to model uncertainty and apply Bayes' theorem to real-world situations, enhancing their analytical skills.

- **Random Variables and Distributions**: We will delve into random variables and their associated probability distributions, exploring both discrete and continuous cases. This section will cover important distributions such as the binomial, normal, and Poisson distributions. R will be utilized to visualize these distributions and understand their applications in statistical analysis.

- **Correlation and Regression**: Understanding relationships between variables is crucial in data analysis. We will explore correlation coefficients and regression analysis, teaching students how to quantify relationships between variables. Using R, students will create regression models and visualizations, allowing them to make predictions based on their findings.

- **Hypothesis Testing and Decision Making**: The final section addresses hypothesis testing—a fundamental aspect of statistical analysis. Students will learn how to formulate and test hypotheses using z-tests, t-tests, F-tests, and chi-square tests. R will serve as a powerful tool for conducting these tests and interpreting the results, providing students with the skills to make informed decisions based on data.

By integrating R programming throughout the manual, we ensure that students not only grasp statistical theories but also develop practical coding skills essential for data analysis. Each chapter includes coding examples, exercises, and case studies that encourage students to apply their knowledge in real-world contexts.

The use of R in this manual is supported by a wealth of literature that showcases its capabilities and applications in statistics. For instance, *R for Data Science* by Hadley Wickham and Garrett Grolemund provides an excellent introduction to data science concepts using R, guiding readers through practical data analysis techniques. Furthermore, the R community continuously contributes to a growing repository of packages and resources, ensuring that users have access to the latest developments in statistical methodologies.

By the end of this manual, students will have a solid foundation in both statistical concepts and R programming, equipping them with the skills necessary to tackle diverse data-driven problems. We hope this resource inspires a deeper appreciation for the power of statistics and the role of R in unlocking insights from data.

2. Introduction to R Programming

2.1 Introduction to R

R is a programming language and environment primarily used for statistical computing and data analysis. It was developed by statisticians Ross Ihaka and Robert Gentleman at the University of Auckland, New Zealand, and has gained popularity among data analysts and researchers in various fields, including social sciences, bioinformatics, and finance.

R provides a wide range of statistical techniques and is highly extensible, allowing users to create custom functions and packages. As an open-source language, R has a large and active community that continuously contributes to its development and improvement. This community-driven approach has resulted in a vast ecosystem of packages available through the Comprehensive R Archive Network (CRAN), which enhances R's capabilities for various data analysis tasks.

The language is particularly known for its powerful data visualization tools, which enable users to create high-quality graphs and plots. R's syntax and data structures are designed to facilitate data manipulation, making it a preferred choice for exploratory data analysis.

R is also widely used for reproducible research, enabling analysts to produce dynamic reports that combine code, output, and narrative text, thus making the research process transparent and verifiable.

Overall, R has become an essential tool for data scientists and statisticians, offering robust solutions for data analysis and visualization.

- History and development of R.: R was developed in the early 1990s by statisticians Ross Ihaka and Robert Gentleman at the University of Auckland, New Zealand. It was designed as a free, open-source programming language for statistical computing and data analysis, inspired by the S programming language created by John Chambers and his colleagues at Bell Laboratories.

  The first version of R was released in 1995, and it quickly gained traction within the statistical community. Its development was fueled by

the growing need for powerful statistical tools that were accessible to researchers and analysts. In 2000, the R Project was formally established, and R became widely recognized as a leading language for statistical analysis and data visualization.

Over the years, R has evolved significantly, with contributions from a vibrant global community of developers and users. The Comprehensive R Archive Network (CRAN) was established as a repository for R packages, allowing users to share their work and access a vast array of additional functionalities.

Today, R is supported by a comprehensive ecosystem, including numerous packages for various statistical techniques, data manipulation, and visualization. Its adoption continues to grow in academia, industry, and government, making it one of the most popular tools for data analysis and research.

- Advantages of using R for statistics and data analysis.: R offers several advantages for statisticians and data analysts:
  - **Open Source:** R is free to use and distribute, which makes it accessible to a wide range of users and encourages community contributions.
  - **Comprehensive Statistical Packages:** R has a vast collection of packages available through CRAN, providing tools for a wide array of statistical analyses, from basic to advanced.
  - **Data Visualization:** R excels in data visualization, with packages like `ggplot2` that allow users to create high-quality, customizable graphics.
  - **Active Community:** A large and active community contributes to R's development, providing support, resources, and continuously updated packages.
  - **Reproducible Research:** R supports reproducible research through R Markdown, allowing users to combine code, analysis, and documentation in a single document.
  - **Integration with Other Tools:** R can easily integrate with other programming languages (e.g., Python, C++), databases, and tools, enhancing its capabilities for data analysis.
  - **User-Friendly Syntax:** The syntax of R is designed for data analysis, making it intuitive for statisticians and researchers, even those with limited programming experience.

- Overview of R's ecosystem, including RStudio and CRAN.: R's ecosystem is rich and diverse, facilitating a wide range of statistical computing and data analysis tasks. Key components include:
  - **R:** The core programming language, designed for statistical analysis and data manipulation, with a wide array of built-in functions and libraries.
  - **CRAN (Comprehensive R Archive Network):** A repository of R packages that extends R's capabilities. It hosts thousands of packages for various statistical methods, data visualization, and machine learning.
  - **RStudio:** An integrated development environment (IDE) that enhances the R programming experience. It offers features like syntax highlighting, debugging tools, and project management, making it easier for users to write and organize their R code.
  - **R Markdown:** A tool for creating dynamic reports that combine R code with narrative text, enabling the presentation of data analysis results in an easily shareable format.
  - **Bioconductor:** A project that provides tools for the analysis of genomic data, offering specialized packages for bioinformatics.
  - **Shiny:** A web application framework for R that allows users to create interactive web applications directly from R, making data analysis accessible to non-programmers.

  This ecosystem collectively empowers users to perform complex data analyses, create reproducible research, and visualize results effectively.

2.2 Installing R and RStudio

To get started with R, you first need to install R and RStudio.

- Step-by-step instructions for installing R and RStudio.
- Overview of R packages and how to install them using `install.packages()`.
- Basic configuration tips for RStudio.

2.3 Installing R

2.3.1 Windows

To install R on Windows:

- Visit the CRAN Windows page.
- Click on the download link for the latest version of R.
- Run the installer and follow the on-screen instructions, leaving most options at their default values unless you have specific needs.

2.3.2 Mac OS X

For Mac users:

- Go to the CRAN Mac OS page.
- Download the most recent version of the `.pkg` file (e.g., `R-4.3.0.pkg`).
- Open the file and follow the installer instructions to complete the installation.

2.3.3 Linux

On Linux-based systems, R can be installed using the terminal:

- Open a terminal window.
- For Debian-based distributions (e.g., Ubuntu), use the command:

```
sudo apt-get install r-base
```

- For Fedora-based distributions, use:

```
sudo dnf install R
```

- Ensure that any dependencies required by R are installed.

2.4 Installing RStudio

Once R is installed, you can install RStudio:

- Visit the RStudio download page.
- Choose your operating system (Windows, Mac, or Linux) and download the corresponding installer.
- Run the installer and follow the on-screen instructions to set up RStudio.

2.4.1 Windows and Mac

For both Windows and Mac, the installation is straightforward: download the executable file, open it, and proceed with the default options unless specific configurations are needed.

2.4.2 Linux

On Linux, after downloading the `.deb` or `.rpm` file, use the terminal to install RStudio:

- For Debian-based systems, run:

```
sudo dpkg -i rstudio-x.yy.zzz-amd64.deb
```

- For Fedora-based systems, run:

```
sudo dnf install rstudio-x.yy.zzz-x86_64.rpm
```

- Replace `x.yy.zzz` with the actual version number of RStudio you downloaded.

2.5 Overview of R Packages

R packages are collections of R functions, data, and compiled code that are stored in a well-defined format. They extend R's functionality by providing tools for specific tasks, such as data manipulation, visualization, or statistical analysis. Many packages are available through CRAN (Comprehensive R Archive Network), but others are hosted on platforms like GitHub.

2.6 Installing R Packages

To install an R package, use the `install.packages()` function. This function downloads and installs the specified package from CRAN or other repositories.

2.6.1 Example of Installing a Package

For example, to install the `ggplot2` package for data visualization, run the following command in your R console:

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

This command will download and install the package, including any dependencies, automatically.

2.6.2 Loading an Installed Package

Once a package is installed, load it into your R session using the `library()` function:

```
library(ggplot2)
```

2.7 Commonly Used Packages

Some widely-used packages in R include:

- `dplyr` – for data manipulation
- `ggplot2` – for data visualization
- `tidyverse` – a collection of data science tools
- `shiny` – for building interactive web applications

2.8 Basic Configuration Tips for RStudio

- **Appearance:** Go to `Tools > Global Options > Appearance` to change the editor theme and font size for better visibility.
- **Pane Layout:** Customize the layout of RStudio panes (Source, Console, Environment) under `Tools > Global Options > Pane Layout`.

2.9 Code Execution

- Enable line numbers, auto-complete, and code folding under `Tools > Global Options > Code`.
- Set keyboard shortcuts for running code chunks (e.g., `Ctrl + Enter`).

2.10 Packages

- Manage installed packages via the `Packages` tab or use `install.packages()` to add new packages.

2.11 Basic R Syntax

Understanding the basic syntax of R is crucial for efficient programming.

- Variables, data types (numeric, character, factor, logical).
- Basic operators (arithmetic, relational, logical).
- Introduction to vectors and data frames.

2.12 Variables in R

Variables in R are used to store data that can be manipulated during the execution of a program. A variable can be assigned a value using the assignment operator `<-` or `=`. For example:

```
x <- 10
name <- "John"
```

In R, variables do not need explicit declaration, and their type is determined dynamically when a value is assigned.

2.13 Data Types in R

R provides several basic data types that are essential for handling different kinds of data. These include numeric, character, factor, and logical types.

2.13.1 Numeric

The numeric data type is used to store numbers, both integers and real numbers (decimals). For example:

```
x <- 42
y <- 3.14
```

R treats all numbers as double precision by default, but integers can also be explicitly declared using the `as.integer()` function.

2.13.2 Character

Character data represents text or strings. Characters are enclosed in double or single quotes. For example:

```
name <- "Alice"
greeting <- 'Hello'
```

2.13.3 Factor

Factors are used to handle categorical data in R. They store both the values and the possible categories. Factors are particularly useful for statistical modeling:

```
colors <- factor(c("red", "blue", "green", "red"))
```

Factors can be ordered or unordered and are often used in data frames for analysis.

2.13.4 Logical

Logical data types store boolean values: `TRUE` or `FALSE`. They are often the result of comparisons or logical operations:

```
is_sunny <- TRUE
result <- 5 > 3 # result will be TRUE
```

Understanding variables and data types is crucial for working effectively with R. The dynamic nature of R allows for flexible handling of different types of data, making it easier to perform statistical analysis and data manipulation.

2.14 Introduction

In R, operators allow you to perform various computations on data. These include arithmetic operations for numerical calculations, relational operators for comparisons, and logical operators for evaluating boolean expressions. Understanding these operators is fundamental for programming in R.

2.15 Arithmetic Operators

Arithmetic operators are used for basic mathematical computations. The main arithmetic operators in R are:

- `+` : Addition
- `-` : Subtraction

- `*` : Multiplication
- `/` : Division
- `^` : Exponentiation
- `%/%` : Integer division (quotient)
- `%%` : Modulo (remainder)

2.15.1 Examples

```
x <- 10
y <- 3

sum <- x + y      # sum = 13
diff <- x - y     # diff = 7
prod <- x * y     # prod = 30
quot <- x / y     # quot = 3.33
exp <- x ^ y      # exp = 1000
int_div <- x %/% y # int_div = 3
mod <- x %% y     # mod = 1
```

2.16 Relational Operators

Relational operators compare values and return `TRUE` or `FALSE`. The main relational operators are:

- `==` : Equal to
- `!=` : Not equal to
- `>` : Greater than
- `<` : Less than
- `>=` : Greater than or equal to
- `<=` : Less than or equal to

2.16.1 Examples

```
x <- 5
y <- 10

result1 <- x == y  # result1 = FALSE
result2 <- x != y  # result2 = TRUE
result3 <- x > y   # result3 = FALSE
result4 <- x <= y  # result4 = TRUE
```

Relational operators are often used in conditions for `if` statements, loops, and other control structures.

2.17 Logical Operators

Logical operators evaluate logical expressions and are primarily used to combine relational operations. The main logical operators are:

- `&` : Logical AND
- `|` : Logical OR
- `!` : Logical NOT

2.17.1 Examples

```
a <- TRUE
b <- FALSE

result1 <- a & b   # result1 = FALSE (both must be TRUE)
result2 <- a | b   # result2 = TRUE (one must be TRUE)
result3 <- !a      # result3 = FALSE (logical negation)
```

Logical operators are especially useful when you need to check multiple conditions.

Arithmetic, relational, and logical operators are crucial in R for data manipulation, decision-making, and computations. They provide a foundation for writing effective R scripts and performing complex analyses.

2.18 Vectors in R

A vector is one of the most basic data structures in R. It is a sequence of data elements of the same type, such as numeric, character, or logical. Vectors can

be created using the `c()` function, which combines values into a single vector.

2.18.1 Example of a Numeric Vector

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

Vectors in R are 1-dimensional and are fundamental to most data manipulation operations.

2.18.2 Example of a Character Vector

```
names <- c("Alice", "Bob", "Charlie")
```

In this chapter, we will cover vectors. A vector is a data structure in R. A data structure can be defined as a collection of data values, the relationships among them, and the functions or operations that can be applied to the data. A vector can either be an atomic vector or a list. An atomic vector contains only one data type, whereas a list may contain multiple data types.

An atomic vector could be Character, Logical, Double, Integer, or Complex.

2.18.3 Integer

```
4
## [1] 4
```

2.18.4 Character

```
"Pune"
## [1] "Pune"
```

You can use the `typeof` function to check the type of each of these:

```
typeof("Pune")
## [1] "character"
typeof(4)
## [1] "double"
typeof(4.3)
## [1] "double"
typeof(2+1i)
## [1] "complex"
```

2.18.5 Creating a Vector

Let's create a vector containing three numeric values. This will be an example of a Numeric Vector:

```
c(1, 4, 7)
## [1] 1 4 7
```

To check the length of the vector, we can use the length function:

```
length(c(1, 4, 7))
## [1] 3
```

Let's assign this to a variable for future use:

```
Num_variable <- c(1, 4, 7)
```

Let's create a vector `Names` that contains the names of persons appearing in an Exam. This will be a kind of character vector. Please note that here `Names` is a vector, and `<-` is an operator that assigns the value on the right-hand side of the operator to the variable name on the left-hand side. The character `c` is used to create a vector:

```
Names <- c("Arvind", "Krishna", "Rahul", "Saurabh", "Venkat",
    "Sucharitha")
```

Let's extract the first element of the `Names` vector. To extract the first element, we need to use `[` and the value 1:

```
Names[1]
## [1] "Arvind"
```

Similarly, to extract the third value from the `Names` vector:

```
Names[3]
## [1] "Rahul"
```

Let's create another vector and see if we can combine two vectors:

```
Missed_names <- c("Ajay", "Amit")
```

Here we will be updating the `Names` vector by including `Missed_names` in the `Names` vector:

```
Names <- c(Names, Missed_names)
Names
## [1] "Arvind"   "Krishna"   "Rahul"     "Saurabh"   "Venkat"
```

```
## [6] "Sucharitha" "Ajay"    "Amit"
```

2.18.6 Class of Vector

We can also check the types of vectors using different functions such as:

```
class(Names)
## [1] "character"
class(Num_variable)
## [1] "numeric"
typeof(Num_variable)
## [1] "double"
```

What is the difference between `typeof`, `mode`, and `class`? I will leave it to the reader to explore it on their own. Remember the reader can use the `help` function to check the definitions in detail. For example, just type `?class` in the console to see the details of the class function.

2.18.7 Adding Vectors of Different Types

Let's add a character and numeric vector and check the result:

```
mix_vector <- c(1, 2, "Amish")
class(mix_vector)
## [1] "character"
```

The `mix_vector` we created had multiple data types (Numeric and Character); however, R converts the multiple data types to a single data type through a process called coercion. Logical values are converted to numbers: TRUE is converted to 1 and FALSE to 0.

Values are converted to the simplest type required to represent all information.

The ordering is roughly logical < integer < numeric < complex < character < list.

Objects of type `raw` are not converted to other types.

Objects can also be explicitly coerced using the `as.` function. For example, to coerce the `mix_vector` to numeric use:

```
mix_vector <- as.numeric(mix_vector)
## Warning: NAs introduced by coercion
mix_vector
## [1]  1  2 NA
class(mix_vector)
## [1] "numeric"
```

We can also change the elements of the vector. Let's change the first element of the `Names` vector using the following expression:

```
Names[1] <- "Arun"
Names
## [1] "Arun"      "Krishna"   "Rahul"     "Saurabh"   "Venkat"
## [6] "Sucharitha" "Ajay"     "Amit"
```

2.18.8 Accessing Elements of Vectors

Let's try to access elements of the vector using negative indexing:

```
Names[-1] # All elements except the first element will be printed
## [1] "Krishna"  "Rahul"     "Saurabh"   "Venkat"    "Sucharitha"
## [6] "Ajay"     "Amit"
Names[c(-1, -2)] # All elements except the first and second elements
    will be printed
## [1] "Rahul"    "Saurabh"   "Venkat"    "Sucharitha" "Ajay"
## [6] "Amit"
```

Alternatively, we can also use the following code to remove the first two elements of the `Names` vector:

```
Names[-c(1, 2)] # All elements except the first and second elements
    will be printed
## [1] "Rahul"    "Saurabh"   "Venkat"    "Sucharitha" "Ajay"
## [6] "Amit"
```

2.18.9 Creating Vector Using In-Built Functions

```
seq(1, 10, by = 1) # Create a sequential vector from 1 to 10 which
    increases by 1 in length.
##  [1]  1  2  3  4  5  6  7  8  9 10
seq(1, 10, by = 0.5)
##  [1]  1.0  1.5  2.0  2.5  3.0  3.5  4.0  4.5  5.0  5.5  6.0  6.5  7.0  7.5
## [15] 8.0  8.5  9.0  9.5 10.0
```

2.18.10 Let's Try to Repeat Vectors

```
rep(c(1, 2), times = c(5, 5))
##  [1] 1 1 1 1 1 2 2 2 2 2
```

`typeof(Vector)`: This method tells you the data type of the vector.

`Sort(Vector)`: This method helps us to sort the items in ascending order.

`length(Vector)`: This method counts the number of elements in a vector.

`head(Vector, limit)`: This method returns the top six elements (if you omit the limit). If you specify the limit as 4, then it returns the first 4 elements.

`tail(Vector, limit)`: It returns the last six elements (if you omit the limit). If you specify the limit as 2, then it returns the last two elements.

2.18.11 Arithmetic Operators in Vectors in R

We have performed arithmetic operations in previous chapters. Let's see how we can perform arithmetic operations in vectors.

```
V1 <- c(1, 2, 3, 4) # created a vector V1
V2 <- c(5, 6, 7, 8) # created a vector V2
```

Let's perform addition:

```
V12 <- V1 + V2
V12 # Print result of V12, which is sum of V1 and V2 vector.
## [1]  6  8 10 12
```

As you may have guessed, the addition operation above was performed elementwise. That means that the first element of `V1` was added to the first element of `V2`.

Let's perform multiplication:

```
V12_mult <- V1 * V2
V12_mult
## [1]  5 12 21 32
```

Let's perform division:

```
V12_division <- V2 / V1
V12_division
## [1] 5.000000 3.000000 2.333333 2.000000
typeof(V12_division)
## [1] "double"
```

Let's convert the double to numeric:

```
V12_division <- as.integer(V12_division)
V12_division
## [1] 5 3 2 2
typeof(V12_division)
## [1] "integer"
```

2.19 Introduction to Data Frames in R

Data frames in R language are a type of data structure used to store data in a tabular form, which is two-dimensional. The data frames are a special category of list data structures in which the components are of equal length. R language supports the built-in function `data.frame()` to create data frames and assign data elements. R also supports using the data frame name to modify and retrieve data elements from data frames. Data frames in R are structured with column names as component names, and rows structured by component values. Data frames in R are a widely used data structure when developing machine learning models in data science projects.

2.19.1 Characteristics of Data Frames

- The column name is required.
- Row names should be unique.
- The number of items in each column should be the same.

2.20 Steps For Creating Data Frames in R

Let's start with creating a data frame, which is explained below:

2.20.1 Step 1: Create a Data Frame of a Class in a School

Code:

```
tenthclass = data.frame(roll_number = c(1:5),
Name = c("John","Sam","Casey","Ronald","Mathew"),
Marks = c(77,87,45,68,95), stringsAsFactors = FALSE)
print(tenthclass)
```

When we run this code, we will get a data frame like this:

Output:

Here, in our example, the data frame is very small, but in real life, while dealing with problems, we have lots of data. To understand the structure of the data, we use the `Str()` function.

```
  roll_number   Name Marks
1           1   John    77
2           2    Sam    87
3           3  Casey    45
4           4 Ronald    68
5           5 Mathew    95
```

Figure 2.1: Data frame output example

2.20.2 Step 2: Add the following line to our code

Code:

```
Str(tenthclass)
```

When we run the whole code, we will get the following output:

Output:

```
'data.frame':   5 obs. of  3 variables:
 $ roll_number: int  1 2 3 4 5
 $ Name       : chr  "John" "Sam" "Casey" "Ronald" ...
 $ Marks      : num  77 87 45 68 95
```

Figure 2.2: Data frame output example

The above output means we have 5 observations of 3 variables. It also explains the data type of each variable. For example, the roll number is an integer, the name is a character, and Marks are numeric.

Once we understand the structure of the data, we can pass the following code to understand the data more statistically.

2.20.3 Step 3: Use the `summary()` Function

Code:

```
summary(tenthclass)
```

Output:

The `summary()` function provides a better understanding of our data. It tells us the mean, median, quartiles, maximum, and minimum values, helping us make better decisions.

```
 roll_number     Name              Marks
Min.   :1    Length:5           Min.   :45.0
1st Qu.:2    Class :character   1st Qu.:68.0
Median :3    Mode  :character   Median :77.0
Mean   :3                       Mean   :74.4
3rd Qu.:4                       3rd Qu.:87.0
Max.   :5                       Max.   :95.0
```

Figure 2.3: Data frame output example

Structure

When we want to know the structure of a particular data frame, we can use the following function:

```
Star()

str(Data_frame)
```

Output:

```
Number: num 2 3 4
alpha: Factor w/ 3 levels x,y,z: 1 2 3
Booleans: logi TRUE TRUE FALSE
```

How to Extract Data from Data Frames in R?

Here we will continue the above case. Let's suppose we want to know the name of the student in class tenth, just the name. So how will we extract it?

Our data frame looks like this:

```
roll_number  Name   Marks
1            John   77
2            Sam    87
3            Casey  45
4            Ronald 68
5            Mathew 95
```

To just get the name as an output, we will pass on the following code.

Code:

```
onlyname = tenthclass$Name
```

```
print(onlyname)
```

Output:

```
[1] "John"   "Sam"    "Casey"  "Ronald" "Mathew"
```

Figure 2.4: Data frame output example

Here, if we break the code, we just put the dollar sign in between the name of our data frame and the name of the variable that we want as an output.

Now consider a situation: the teacher wants to know everything about roll number 2, like what his name is and how much he scored.

We need everything about roll number 2, so we will pass on the below-mentioned code.

Code:

```
result_rollnumber2 = tenthclass[c(2), c(1:3)]
print(result_rollnumber2)
```

Output:

```
  roll_number Name Marks
2           2  Sam    87
```

Figure 2.5: Data frame output example

Expand in Data Frames

The data frame can be increased and decreased in size by adding or deleting columns and rows.

1. Add Row

We have two data frames. One data frame belongs to class tenth section A and the other data frame belongs to class tenth section B. Now these different sections are merging into a single class.

Example #1: Class 10 A

Code:

```
tenthclass_sectionA = data.frame(roll_number = c(1:5),
Name = c("John","Sam","Casey","Ronald","Mathew"),
Marks = c(77,87,45,68,95), stringsAsFactors = FALSE)
print(tenthclass_sectionA)
```

Output:

```
  roll_number   Name Marks
1           1   John    77
2           2    Sam    87
3           3  Casey    45
4           4 Ronald    68
5           5 Mathew    95
```

Figure 2.6: Data frame output example

Example #2: Class 10 B

Code:

```
tenthclass_sectionB = data.frame(roll_number = c(6:10),
Name = c("Ria","Justin","Bon","Tim","joe"),
Marks = c(68,98,54,68,42), stringsAsFactors = FALSE)
print(tenthclass_sectionB)
```

Output:

```
  roll_number   Name Marks
1           6    Ria    68
2           7 Justin    98
3           8    Bon    54
4           9    Tim    68
5          10    joe    42
```

Figure 2.7: Data frame output example

Example #3: rbind() function Now we have to merge both classes into a single class. We will use the `rbind()` function here. The only limitation in adding a new row is that we need to bring in the new rows in the same structure as the existing data frame.

Code:

```
new_tenthclass = rbind(tenthclass_sectionA, tenthclass_sectionB)
print(new_tenthclass)
```

Output:

	roll_number	Name	Marks
1	1	John	77
2	2	Sam	87
3	3	Casey	45
4	4	Ronald	68
5	5	Mathew	95
6	6	Ria	68
7	7	Justin	98
8	8	Bon	54
9	9	Tim	68
10	10	joe	42

Figure 2.8: Data frame output example

2. Add Column

Now consider a case where we have to add blood group details for every student in class 10. We will add a new column for it and name it as "Blood_group".

Our data frame looks like this:

Code:

```
tenthclass = data.frame(roll_number = c(1:5),
Name = c("John","Sam","Casey","Ronald","Mathew"),
Marks = c(77,87,45,68,95), stringsAsFactors = FALSE)
print(tenthclass)
```

Output:

```
  roll_number   Name Marks
1           1   John    77
2           2    Sam    87
3           3  Casey    45
4           4 Ronald    68
5           5 Mathew    95
```

Figure 2.9: Data frame output example

Code:

```
tenthclass$Blood_group = c("O","AB","B+","A+","AB")
print(tenthclass)
```

Output:

```
  roll_number   Name Marks Blood_group
1           1   John    77           O
2           2    Sam    87          AB
3           3  Casey    45          B+
4           4 Ronald    68          A+
5           5 Mathew    95          AB
```

Figure 2.10: Data frame output example

3. Delete Column

In this data frame, if we need to delete the blood group variable (the rightmost column), we will pass the following code.

Code:

```
tenthclass$Blood_group = NULL
print(tenthclass)
```

Output:

```
  roll_number   Name Marks
1           1   John    77
2           2    Sam    87
3           3  Casey    45
4           4 Ronald    68
5           5 Mathew    95
```

Figure 2.11: Data frame output example

By passing the `NULL` command, we can directly remove the variable from our data frame.

4. Delete Row

Now consider a situation where we don't need the marks of John, so we have to remove the topmost row.

Code:

```
tenthclass = tenthclass[-1, ]
print(tenthclass)
```

Output:

```
  roll_number   Name Marks
2           2    Sam    87
3           3  Casey    45
4           4 Ronald    68
5           5 Mathew    95
```

Figure 2.12: Data frame output example

5. Update Data in Data Frame

Let's suppose Sam scored 98 marks, but in our data frame, his marks are shown as 87. We can pass the following code to rectify this.

Code:

```
tenthclass$Marks[2] = 98
print(tenthclass)
```

Output:

```
  roll_number   Name Marks
1           1   John    77
2           2    Sam    98
3           3  Casey    45
4           4 Ronald    68
5           5 Mathew    95
```

Figure 2.13: Data frame output example

2.21 Inspecting Data Frames

Below are different ways to inspect a data frame and obtain information about it, similar to the `str()` function.

1. Names: Provides the Names of the Variables in the Data Frame

Syntax:

```
names(data_frame_name)
```

Example:

```
Number <- c(2,3,4)
alpha <- c("x","y","z")
Booleans <- c(TRUE,TRUE,FALSE)
Data_frame <- data.frame(Number, alpha, Booleans)
names(Data_frame)
```

Output:

```
[1] "Number" "alpha" "Booleans"
```

2. Summary: Provides Statistics of the Data Frame

Syntax:

```
summary(data_frame_name)
```

Example:

```
Number <- c(2,3,4)
alpha <- c("x","y","z")
Booleans <- c(TRUE,TRUE,FALSE)
Data_frame <- data.frame(Number, alpha, Booleans)
summary(Data_frame)
```

Output:

```
    Number    alpha  Booleans
 Min.   :2.0  x:1    Mode :logical
 1st Qu.:2.5  y:1    FALSE:1
 Median :3.0  z:1    TRUE :2
 Mean   :3.0  NAs:0
 3rd Qu.:3.5
 Max.   :4.0
```

3. Head: Provides the Data for the First Few Rows

Syntax:

```
head(data_frame_name)
```

Example:

```
Number <- c(2,3,4,5,6,7,8,9,10,11)
alpha <- c("x","y","z","a","b","c","d","f","g","j")
Booleans <-
    c(TRUE,TRUE,FALSE,TRUE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE)
Data_frame <- data.frame(Number, alpha, Booleans)
head(Data_frame)
```

Output:

```
Number alpha Booleans
1      2     x     TRUE
2      3     y     TRUE
3      4     z     FALSE
4      5     a     TRUE
```

```
5     6     b    FALSE
6     7     c    FALSE
```

4. Tail: Prints the Last Few Rows in the Data Frame

Syntax:

```
tail(data_frame_name)
```

Example:

```
Number <- c(2,3,4,5,6,7,8,9,10,11)
alpha <- c("x","y","z","a","b","c","d","f","g","j")
Booleans <-
    c(TRUE,TRUE,FALSE,TRUE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE)
Data_frame <- data.frame(Number, alpha, Booleans)
tail(Data_frame)
```

Output:

```
Number alpha Booleans
5     6     b    FALSE
6     7     c    FALSE
7     8     d    FALSE
8     9     f    FALSE
9    10     g    FALSE
10   11     j    FALSE
```

2.22 Extracting Specific Data from the Data Frame

1. Using the Column Name

We can extract a particular set of data from the data frame. For instance, to extract only the `Number` column:

```
Data_frame <- data.frame(Number)
```

Output:

```
Number
1     2
2     3
3     4
```

2. Using the Rows

To print only the first two rows of the `Number` column:

```
output <- Data_frame[1:2,]
print(output)
```

Output:

```
Number alpha Booleans
1      2     x     TRUE
2      3     y     TRUE
```

3. Printing Specific Rows and Columns

For example, to print rows 1 and 2 of columns 1 and 2:

```
output <- Data_frame[c(1,2),c(1,2)]
print(output)
```

Output:

```
Number alpha
1      2     x
2      3     y
```

4. Adding Another Column to the Data Frame

```
Data_frame$class <- c("A","B","C")
```

Output:

```
Number alpha Booleans class
1      2     x     TRUE     A
2      3     y     TRUE     B
3      4     z    FALSE     C
```

5. Adding a Row to the Data Frame

We use `rbind()` to add a new row:

```
out <- rbind(Data_frame, c(5, "x", FALSE, "D"))
print(out)
```

Output:

```
Number alpha Booleans class
1      2     x     TRUE     A
2      3     y     TRUE     B
3      4     z    FALSE     C
4      5     x    FALSE     D
```

6. Combining Both Data Frames

```
out <- rbind(Data_frame1, Data_frame2)
print(out)
```

Output:

```
Number alpha Booleans
1      2     x     TRUE
2      3     y     TRUE
3      4     z    FALSE
4      4     x     TRUE
5      5     y     TRUE
6      6     z    FALSE
```

2.23 Conclusion

Data frames are commonly used data structures that list variables with unique row IDs. This article covers methods for adding and deleting rows and columns, updating data, and accessing data in a data frame.

2.24 Matrices in R

Matrices are two-dimensional data structures in R and are arranged in a rectangular layout. Matrices can contain only one data type. We can create matrices of any of the six data types we discussed before. A matrix can also be thought of as a vector in two dimensions.

We can use the `matrix` function to create a matrix in R programming. I will suggest using `? matrix` for detailed documentation of the matrix function. Let's create a matrix with 3 rows and 2 columns:

```
matrix(1:6, nrow = 3, ncol = 2)
##      [,1] [,2]
```

```
## [1,]   1   4
## [2,]   2   5
## [3,]   3   6
```

What if we don't specify the number of rows and columns?

```
matrix(1:6, 3, 2)
##      [,1] [,2]
## [1,]   1   4
## [2,]   2   5
## [3,]   3   6
```

As you have guessed from the above output, R will automatically pick the number of rows and columns based on the order of input. Let's specify one of the dimensions (either column or row) and see the output:

```
matrix(1:6, ncol = 3)
##      [,1] [,2] [,3]
## [1,]   1   3   5
## [2,]   2   4   6
```

So far we have not assigned any column names and row names to the matrix. Let's create a matrix M and assign column names and row names:

```
M <- matrix(1:20, ncol = 4)
colnames(M) <- c("A", "B", "C", "D")
rownames(M) <- c("E", "F", "G", "H", "I")
```

Let's check the matrix M:

```
M
##   A  B  C  D
## E 1  6 11 16
## F 2  7 12 17
## G 3  8 13 18
## H 4  9 14 19
## I 5 10 15 20
```

As you can see from the output, we have successfully assigned names to the columns. We can also call specific columns now. For example, `Matrix[rowname, colname]` will call the required row and column by name:

```
M["F", "D"]
## [1] 17
```

We can leave the row name field blank in `Matrix[rowname, colname]` to

call the complete column:

```
M[, "D"]
##  E  F  G  H  I
## 16 17 18 19 20
```

As you can see in the above example, we have kept the row blank. This will cause the above expression to pick all the values from the corresponding columns. We can also access specific elements of vectors by specifying the row number and column number. Let's say we want to pick a specific column and vector from matrix, say the 2nd row and 3rd column:

```
M[2, 3]
## [1] 12
```

2.24.1 Extracting Row/Column from Matrices

We can also extract more than one row or column at a time. For example, below we extracted the first and third column and the second row:

```
M[2, c(1, 3)]
##  A  C
##  2 12
```

So what else can we do with matrices?

2.24.2 Operations in Matrices

Let's perform some operations with matrices. Let's create two matrices M_1 and M_2 and perform some operations:

```
M1 <- matrix(1:15, nrow = 5)
M1
##      [,1] [,2] [,3]
## [1,]    1    6   11
## [2,]    2    7   12
## [3,]    3    8   13
## [4,]    4    9   14
## [5,]    5   10   15

M2 <- matrix(1:15, nrow = 5)
M2
##      [,1] [,2] [,3]
## [1,]    1    6   11
## [2,]    2    7   12
```

```
## [3,]   3    8   13
## [4,]   4    9   14
## [5,]   5   10   15
```

Arithmetic Operations in Matrices

What if we add two matrices? Matrix operations can be performed similarly to how arithmetic operations are performed on numbers. However, we should be careful with the dimensions of the matrices that we are working on.

Let's create three matrices M_1, M_2, M_3:

```
M1 <- matrix(1:15, nrow = 5)
M1
##      [,1] [,2] [,3]
## [1,]   1    6   11
## [2,]   2    7   12
## [3,]   3    8   13
## [4,]   4    9   14
## [5,]   5   10   15

M2 <- matrix(2:16, nrow = 5)
M2
##      [,1] [,2] [,3]
## [1,]   2    7   12
## [2,]   3    8   13
## [3,]   4    9   14
## [4,]   5   10   15
## [5,]   6   11   16

M3 <- matrix(2:10, nrow = 5)
## Warning in matrix(2:10, nrow = 5): data length [9] is not a
    sub-multiple or
## multiple of the number of rows [5]
M3
##      [,1] [,2]
## [1,]   2    7
## [2,]   3    8
## [3,]   4    9
## [4,]   5   10
## [5,]   6    2
```

Arithmetic Operations in Matrices

Addition operation on matrices of the same dimension:

```
M1 + M2
```

```
##      [,1] [,2] [,3]
## [1,]    3   13   23
## [2,]    5   15   25
## [3,]    7   17   27
## [4,]    9   19   29
## [5,]   11   21   31
```

What if we want to add matrices of different dimensions? We can check the dimensions of matrices using the `dim` function:

```
dim(M1)
## [1] 5 3
dim(M3)
## [1] 5 2
```

Multiplying matrices of the same dimension: The following operation using `"*"` is element-wise:

```
M1 * M2
##      [,1] [,2] [,3]
## [1,]    2   42  132
## [2,]    6   56  156
## [3,]   12   72  182
## [4,]   20   90  210
## [5,]   30  110  240
```

What if we want Matrix Multiplication? As an exercise, try `M1%*%M2` in the R console for matrix multiplication and check the result. Once you run the operation, you must have got an error message "Error in M1 %*% M2: non-conformable arguments". Why is it so? As a basic rule for matrix multiplication, the number of rows of Matrix1 should be equal to the number of columns in Matrix2 when multiplying Matrix1 with Matrix2. You can check the following link *mathisfun*.

```
M4 <- matrix(1:9, nrow=3)
M5 <- matrix(10:18, nrow=3)
M4
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

```
M5
```

```
##      [,1] [,2] [,3]
## [1,]   10   13   16
## [2,]   11   14   17
## [3,]   12   15   18
```

Let's see the result of Matrix Multiplication:

```
M4 <- matrix(1:9, nrow=3)
M5 <- matrix(10:18, nrow=3)
M4 \%*\% M5
```

```
##      [,1] [,2] [,3]
## [1,]  138  174  210
## [2,]  171  216  261
## [3,]  204  258  312
```

You can see the elementwise operation result in R:

```
M4 * M5
```

```
##      [,1] [,2] [,3]
## [1,]   10   52  112
## [2,]   22   70  136
## [3,]   36   90  162
```

```
M1 / M2
```

```
##           [,1]     [,2]     [,3]
## [1,] 0.5000000 0.8571429 0.9166667
## [2,] 0.6666667 0.8750000 0.9230769
## [3,] 0.7500000 0.8888889 0.9285714
## [4,] 0.8000000 0.9000000 0.9333333
## [5,] 0.8333333 0.9090909 0.9375000
```

What if one of the matrices' row or column is shorter than the other? Will recycling occur?

As an exercise, create a 3X3 matrix and add it to M1.

2.24.3 Transposing a Matrix

You may also want to transpose your data in R. This is used to interchange rows and columns, i.e., rows become columns and columns become rows in

the new transposed matrix. For example:

```
M3 <- t(M1)
M3
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    4    5
## [2,]    6    7    8    9   10
## [3,]   11   12   13   14   15
```

As you can see from the output, the rows have become columns and the columns have become rows. This can also be used to reshape a DataFrame. We will cover DataFrame in the next chapter.

2.24.4 Common Matrix Operations in R

Sum of rows in Matrix:

```
rowSums(M1)
```

```
## [1] 18 21 24 27 30
```

Sum of columns in Matrix:

```
colSums(M1)
```

```
## [1] 15 40 65
```

Mean of rows in Matrix:

```
rowMeans(M1)
```

```
## [1]  6  7  8  9 10
```

2.24.5 Naming Matrix Rows and Columns

```
rownames(M4) <- c("A","B","C")
colnames(M4) <- c("D","E","F")
M4
```

```
##   D E F
```

```
## A 1 4 7
## B 2 5 8
## C 3 6 9
```

We can also extract specific elements of a Matrix by specifying rows and columns:

```
M4[3,] % # Extracting third row from Matrix M4
```

```
## D E F
## 3 6 9
```

```
M4[,3] % # Extracting third column from Matrix M4
```

```
## A B C
## 7 8 9
```

```
M4[2,3] % # Extracting second row and third column element from
    Matrix M4
```

```
## [1] 8
```

2.25 Creating a Function

In R, functions are created using the `function` keyword. They take arguments as input, perform specified operations, and return a result. The structure is:

```
function_name <- function(arg1, arg2 = default_value, ...) {
    # Function body with operations
    return(result)
}
```

2.26 Function Arguments

Arguments are inputs for functions, and R supports several types:

- **Positional Arguments**: Matched in the order defined.
- **Named Arguments**: Specified by name; order doesn't matter.

- **Default Arguments**: Take predefined values unless overwritten.
- **Flexible Arguments (...)**: Allow a variable number of inputs.

2.26.1 Example with Default and Named Arguments

```
calculate_area <- function(length, width = 5) {
    area <- length * width
    return(area)
}
calculate_area(3)       # Uses default width = 5
calculate_area(3, width = 4) # Overwrites default width with 4
```

2.27 Return Values

Functions can return values explicitly with `return()`, or implicitly by using the last evaluated expression. They can return multiple values via lists.

```
multi_return <- function(x) {
    sum_x <- sum(x)
    mean_x <- mean(x)
    return(list(sum = sum_x, mean = mean_x))
}
result <- multi_return(c(1, 2, 3, 4))
```

2.28 Default Arguments

Default arguments allow certain parameters to be optional.

```
greet <- function(name = "User") {
    paste("Hello,", name)
}
greet()        # Output: "Hello, User"
greet("Alice") # Output: "Hello, Alice"
```

2.29 Lazy Evaluation

R uses lazy evaluation, meaning arguments are only evaluated when needed.

```
lazy_eval <- function(x, y) {
    if (x > 0) {
```

```
        return(x * y)
    } else {
        return("x is not greater than zero")
    }
}
```

2.30 Anonymous Functions

Anonymous functions, or functions without names, are used for short operations, such as with `apply` functions.

```
numbers <- list(1, 2, 3, 4)
squares <- lapply(numbers, function(x) x^2)
```

2.31 Scoping Rules

R uses lexical scoping, which searches for variables starting from the local environment.

```
x <- 10
scope_example <- function(y) {
    x <- 5
    return(x + y)
}
scope_example(3) # Returns 8; local x = 5 is used
```

2.32 Complete Example of a Function

The following example shows various components, such as arguments, control flow, and list returns.

```
generate_report <- function(data, show_summary = TRUE, ...) {
    if (!is.data.frame(data)) stop("Data must be a data frame")

    if (show_summary) {
        summary_data <- summary(data, ...)
        print("Summary Statistics:")
        print(summary_data)
    }

    num_data <- data[sapply(data, is.numeric)]
```

```
    mean_values <- sapply(num_data, mean, ...)

    return(list(summary = summary_data, means = mean_values))
}
generate_report(mtcars)
```

2.33 Conclusion

Functions in R provide a powerful and flexible way to perform calculations, customize analyses, and structure code. They are essential for creating reusable, organized, and efficient code.

2.34 Data Import and Export in R

Data import is a crucial aspect of data analysis, as it enables R to read and manipulate various data formats. Here's a breakdown of common data file types and methods for importing them:

2.34.1 1. CSV Files

CSV files are among the most widely used formats for data storage. They contain plain text data organized in a tabular format, where each line corresponds to a row and each value is separated by a comma.

Functionality: The `read.csv()` function is tailored for reading CSV files. It automatically assumes that the first line contains header names, making it convenient for typical datasets.

Example:

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

Parameters:

- `header`: Logical value indicating whether the first row contains column names.
- `na.strings`: A character vector of strings to interpret as NA values.
- `stringsAsFactors`: If TRUE, converts character vectors to factors.

2.34.2 2. Excel Files

Excel is another common format for data storage, particularly in business and research settings. R provides the `readxl` package to read Excel files (.xls and .xlsx).

Functionality: The `read_excel()` function allows users to specify which sheet to read.

Example:

```
library(readxl)
data <- read_excel("data.xlsx", sheet = "Sheet1")
```

Parameters:

- `col_names`: Set to TRUE or FALSE to indicate whether to treat the first row as column names.
- `range`: Allows users to specify a particular range of cells to import.

2.34.3 3. Text Files

For text files that may not conform to the CSV format, R provides `read.table()` and `read.delim()` for more control.

Functionality: `read.table()` is flexible, allowing you to specify any delimiter.

Example:

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

Parameters:

- `quote`: Specify character(s) to treat as quotes.
- `fill`: If TRUE, will fill rows with fewer columns with NA.

2.34.4 4. R Data Files

R provides the ability to save objects in its own binary format, useful for saving complex objects.

Functionality: The `load()` function loads R objects stored in .RData files.

Example:

```
load("data.RData")
```

2.34.5 5. Databases

R can connect to various database systems (like MySQL, PostgreSQL, and SQLite) through the `DBI` package.

Functionality: You can execute SQL queries to read data directly from databases.

Example:

```
library(DBI)
con <- dbConnect(RSQLite::SQLite(), dbname = "database.db")
data <- dbReadTable(con, "table_name")
```

2.34.6 6. SPSS, SAS, and Stata Files

For users working with data from statistical software, the `haven` package enables importing data from formats like SPSS (.sav), SAS (.sas7bdat), and Stata (.dta).

Functionality: Functions like `read_sav()`, `read_dta()`, and `read_sas()` make it easy to read these files.

Example:

```
library(haven)
data <- read_sav("data.sav") # For SPSS files
```

2.35 Data Export in R

Data export is equally important as it allows users to save processed data into various formats for sharing and analysis.

2.35.1 1. CSV Files

To export data frames to CSV, R provides the `write.csv()` function.

Functionality: This function is straightforward and can be customized to include or exclude row names.

Example:

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

2.35.2 2. Excel Files

The `writexl` package allows exporting data frames to Excel formats (.xls and .xlsx).

Functionality: This package simplifies writing Excel files directly from R.

Example:

```
library(writexl)
write_xlsx(data, "output.xlsx")
```

2.35.3 3. Text Files

You can export data frames to text files using `write.table()`.

Functionality: This function allows for various customizations, including delimiter choices.

Example:

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

2.35.4 4. R Data Files

R objects can be saved using the `save()` function.

Functionality: This method is efficient for preserving the R environment.

Example:

```
save(data, file = "output.RData")
```

2.36 Importing Data from the Internet

R can also handle data import from various online sources, which is particularly useful for accessing real-time data or datasets hosted online.

2.36.1 1. Reading Data from URLs

R can read CSV files hosted on the internet directly using their URL in functions like `read.csv()`.

Example:

```
data <- read.csv("https://example.com/data.csv")
```

2.36.2 2. JSON and APIs

Accessing data from APIs is done through the `httr` and `jsonlite` packages.

Functionality: These packages facilitate making HTTP requests and processing JSON data.

Example:

```
library(httr)
library(jsonlite)

response <- GET("https://api.example.com/data")
data <- fromJSON(content(response, "text"))
```

2.36.3 3. Web Scraping

Using the `rvest` package, users can scrape data from web pages.

Functionality: This allows for the extraction of structured data from HTML documents.

Example:

```
library(rvest)
url <- "https://example.com"
page <- read_html(url)
table <- page %>% html_node("table") %>% html_table()
```

2.36.4 4. Google Sheets

Accessing data from Google Sheets can be done using the `googlesheets4` package.

Functionality: It allows you to read and write data from Google Sheets.

Example:

```
library(googlesheets4)
data <-
    read_sheet("https://docs.google.com/spreadsheets/d/your_sheet_id")
```

2.36.5 5. XML Data

The `xml2` package is useful for reading XML files from online sources.

Example:

```
library(xml2)
data <- read_xml("https://example.com/data.xml")
```

2.37 Conclusion

R's capabilities for data import and export are comprehensive, catering to a wide variety of file formats and sources. By leveraging these functions and packages, users can efficiently manage and analyze data, whether it's stored locally, in databases, or accessible online. For more in-depth details, the R documentation and specific package vignettes are excellent resources that provide guidance and additional examples. You can explore them at the CRAN website and the RStudio documentation.

2.38 Data Visualization

R programming provides powerful tools for data visualization, enabling the creation of a wide range of graphics, from simple charts to complex multi-layered plots. R's graphics capabilities are supported by both its base graphics system and more advanced packages, such as **ggplot2** and **lattice**, each offering distinct methods for visual representation of data.

The **base graphics** system, which is part of R's default functionality, allows users to create basic plots such as histograms, bar charts, line plots, and scatter plots. It provides extensive customization options for colors, labels, legends, and plot dimensions, making it flexible for various types of visualizations. However, creating complex visualizations may require significant manual adjustments.

For more intricate graphics, the **ggplot2** package is widely used. It simplifies the process of creating layered plots and complex graphs. Based on the grammar of graphics, ggplot2 enables users to build visualizations by layering data, scales, coordinates, and aesthetic mappings. This approach allows for modular and scalable chart building, making it ideal for exploratory data analysis and publication-quality graphics.

The **lattice** package is another useful tool, particularly for multi-panel plots, as it allows users to display data across multiple subsets or conditions within a single graphical layout. This is especially useful for comparing patterns across groups and visualizing multi-dimensional data.

Overall, R's graphics system supports data-driven decision-making by transforming raw data into intuitive, interpretable visual formats. It enables both rapid exploration and detailed, professional presentations of findings.

The `plot` function in R is highly customizable, with numerous arguments that allow users to tailor each plot to their specific needs. These arguments cover data input, graphical appearance, axis settings, and labels. Below is an overview of the key arguments that influence the behavior of the `plot` function:

- **x, y**: These arguments define the data to be plotted on the x-axis and y-axis, respectively. The `x` and `y` arguments can be numeric vectors, time series, or other data types compatible with `plot`. If only `x` is specified, `plot` assumes it's a univariate plot.

- **type**: This argument specifies the type of plot to produce. Options include `"p"` for points (default), `"l"` for lines, `"b"` for both points and lines, `"h"` for histogram-like vertical lines, `"o"` for over-plotted points and lines, and `"s"` or `"S"` for step plots. Choosing the appropriate `type` provides different perspectives on the data.

- **main, xlab, ylab**: These arguments control the main title and axis labels of the plot. `main` adds a main title, while `xlab` and `ylab` set labels for the x-axis and y-axis, respectively. This allows users to add context and meaning to the plot, aiding interpretation.

- **xlim, ylim**: The `xlim` and `ylim` arguments define the range of values for the x-axis and y-axis, respectively. These are set as numeric vectors, e.g., `xlim = c(min, max)`, to manually control the scale of the plot. Adjusting these can enhance focus on specific areas of the data.

- **col, pch, lty, lwd**: These arguments control the graphical properties of the plot elements. `col` sets the color of points or lines, `pch` specifies the plotting symbol for points, `lty` defines the line type (e.g., solid, dashed), and `lwd` controls line width. Together, they allow fine-grained customization of the plot's appearance.

- **axes**: This logical argument specifies whether to draw the axes on the plot. By default, `axes = TRUE`, but setting it to `FALSE` allows users to create custom axes using the `axis` function, adding more control over axis settings.

- **ann**: This logical argument determines whether axis labels and titles should be drawn. When `ann = FALSE`, the labels and titles are omitted, which is useful when the user wants to add custom annotations or create minimalist plots.

- **asp**: The aspect ratio (`asp`) sets the relative scaling between x and y axes. For example, `asp = 1` ensures equal scaling, producing a square plot. This is particularly useful for visualizations that need accurate scaling, like maps.

- **...** (Ellipsis): The `...` argument allows users to pass additional arguments to customize the plot further. This can include arguments for

other functions like `text`, `points`, and `lines`, providing a flexible way to enhance the plot with layered elements or annotations.

These arguments make the `plot` function versatile and adaptable, allowing users to create a wide range of visualizations with a high level of detail and customization. The flexibility provided by these parameters is one of the reasons why `plot` is a foundational tool in R's graphical system.

First, we'll produce a very simple graph using the values in the `cars` vector:

```
# Define the cars vector with 5 values
cars <- c(1, 3, 6, 4, 9)

# Graph the cars vector with all defaults
plot(cars)
```

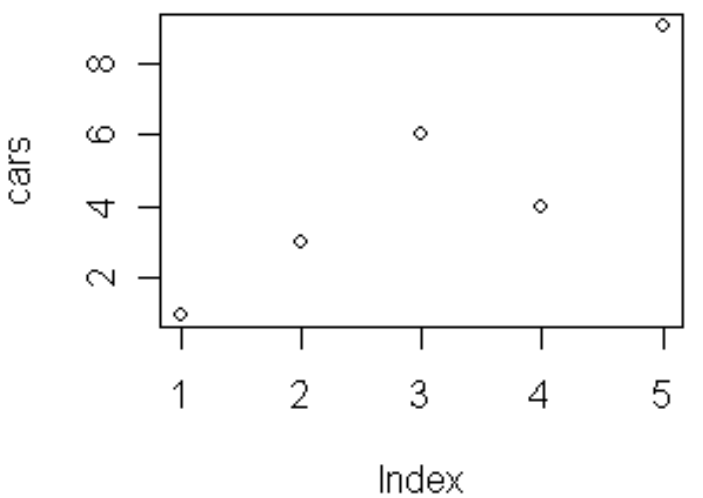


Figure 2.14: Data frame output example

Let's add a title, a line to connect the points, and some color:

```
# Define the cars vector with 5 values
cars <- c(1, 3, 6, 4, 9)

# Graph cars using blue points overlayed by a line
plot(cars, type="o", col="blue")

# Create a title with a red, bold/italic font
title(main="Autos", col.main="red", font.main=4)
```

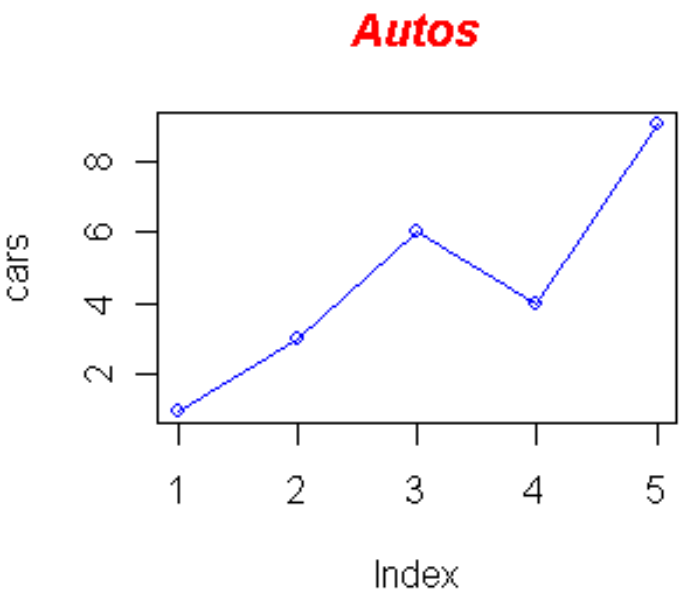


Figure 2.15: Data frame output example

Now let's add a red line for trucks and specify the y-axis range directly so it will be large enough to fit the truck data:

```
# Define 2 vectors
cars <- c(1, 3, 6, 4, 9)
trucks <- c(2, 5, 4, 5, 12)

# Graph cars using a y axis that ranges from 0 to 12
plot(cars, type="o", col="blue", ylim=c(0,12))

# Graph trucks with red dashed line and square points
lines(trucks, type="o", pch=22, lty=2, col="red")

# Create a title with a red, bold/italic font
title(main="Autos", col.main="red", font.main=4)
```

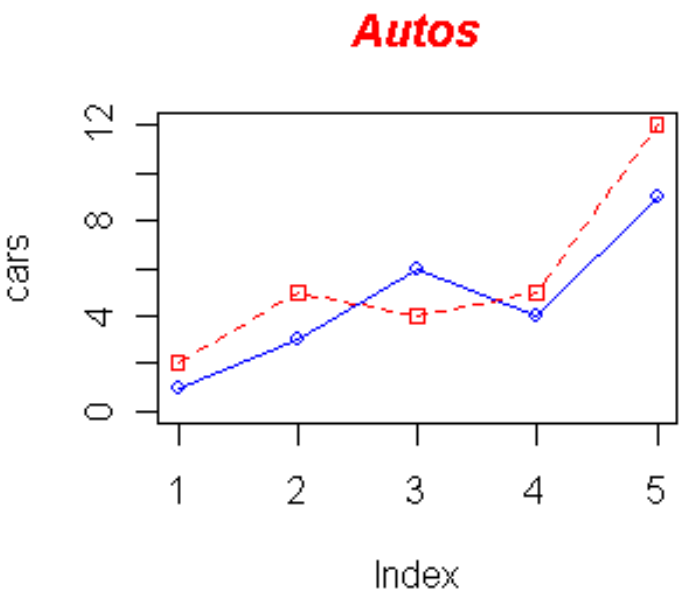


Figure 2.16: Data frame output example

Next, let's change the axes labels to match our data and add a legend. We'll also compute the y-axis values using the `max` function so any changes to our data will be automatically reflected in our graph.

```
# Define 2 vectors
cars <- c(1, 3, 6, 4, 9)
trucks <- c(2, 5, 4, 5, 12)

# Calculate range from 0 to max value of cars and trucks
g_range <- range(0, cars, trucks)

# Graph autos using y axis that ranges from 0 to max
# value in cars or trucks vector. Turn off axes and
# annotations (axis labels) so we can specify them ourselves
plot(cars, type="o", col="blue", ylim=g_range,
   axes=FALSE, ann=FALSE)

# Make x axis using Mon-Fri labels
axis(1, at=1:5, lab=c("Mon","Tue","Wed","Thu","Fri"))

# Make y axis with horizontal labels that display ticks at
# every 4 marks. 4*0:g_range[2] is equivalent to c(0,4,8,12).
axis(2, las=1, at=4*0:g_range[2])
```

```
# Create box around plot
box()

# Graph trucks with red dashed line and square points
lines(trucks, type="o", pch=22, lty=2, col="red")

# Create a title with a red, bold/italic font
title(main="Autos", col.main="red", font.main=4)

# Label the x and y axes with dark green text
title(xlab="Days", col.lab=rgb(0,0.5,0))
title(ylab="Total", col.lab=rgb(0,0.5,0))

# Create a legend at (1, g_range[2]) that is slightly smaller
# (cex) and uses the same line colors and points used by
# the actual plots
legend(1, g_range[2], c("cars","trucks"), cex=0.8,
   col=c("blue","red"), pch=21:22, lty=1:2)
```

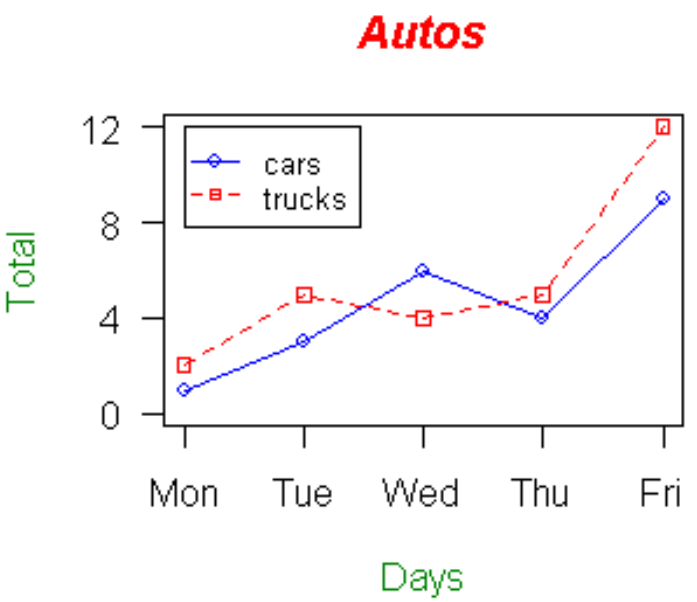


Figure 2.17: Data frame output example

Now let's read the graph data directly from a tab-delimited file. The file contains an additional set of values for SUVs. We'll save the file in the `C:/R` directory (you'll use a different path if not using Windows).

`autos.dat`

```
cars trucks suvs
1    2     4
3    5     4
6    4     6
4    5     6
9    12    16
```

We'll also use a vector for storing the colors to be used in our graph so if we want to change the colors later on, there's only one place in the file that needs to be modified. Finally, we'll send the figure directly to a PNG file.

```
# Read car and truck values from tab-delimited autos.dat
autos_data <- read.table("C:/R/autos.dat", header=T, sep="\t")

# Compute the largest y value used in the data (or we could
# just use range again)
max_y <- max(autos_data)

# Define colors to be used for cars, trucks, suvs
plot_colors <- c("blue","red","forestgreen")

# Start PNG device driver to save output to figure.png
png(filename="C:/R/figure.png", height=295, width=300,
   bg="white")

# Graph autos using y axis that ranges from 0 to max_y.
# Turn off axes and annotations (axis labels) so we can
# specify them ourselves
plot(autos_data$cars, type="o", col=plot_colors[1],
   ylim=c(0,max_y), axes=FALSE, ann=FALSE)

# Make x axis using Mon-Fri labels
axis(1, at=1:5, lab=c("Mon", "Tue", "Wed", "Thu", "Fri"))

# Make y axis with horizontal labels that display ticks at
# every 4 marks. 4*0:max_y is equivalent to c(0,4,8,12).
axis(2, las=1, at=4*0:max_y)

# Create box around plot
box()

# Graph trucks with red dashed line and square points
lines(autos_data$trucks, type="o", pch=22, lty=2,
   col=plot_colors[2])

# Graph suvs with green dotted line and diamond points
```

```
lines(autos_data$suvs, type="o", pch=23, lty=3,
   col=plot_colors[3])

# Create a title with a red, bold/italic font
title(main="Autos", col.main="red", font.main=4)

# Label the x and y axes with dark green text
title(xlab= "Days", col.lab=rgb(0,0.5,0))
title(ylab= "Total", col.lab=rgb(0,0.5,0))

# Create a legend at (1, max_y) that is slightly smaller
# (cex) and uses the same line colors and points used by
# the actual plots
legend(1, max_y, names(autos_data), cex=0.8, col=plot_colors,
   pch=21:23, lty=1:3)

# Turn off device driver (to flush output to png)
dev.off()
```

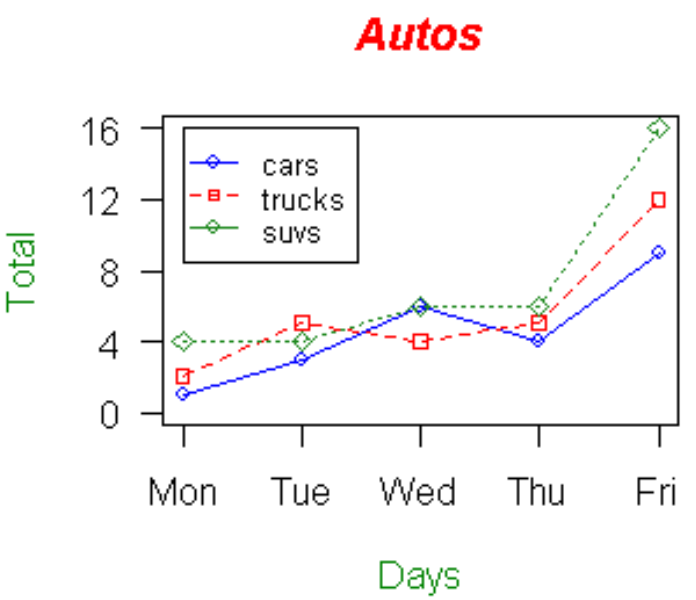


Figure 2.18: Data frame output example

In this next example, we'll save the file to a PDF and chop off extra white space around the graph; this is useful when wanting to use figures in LaTeX. We'll also increase the line widths, shrink the axis font size, and tilt the x-axis labels by 45 degrees.

```
# Read car and truck values from tab-delimited autos.dat
autos_data <- read.table("C:/R/autos.dat", header=T, sep="\t")

# Define colors to be used for cars, trucks, suvs
plot_colors <- c(rgb(r=0.0,g=0.0,b=0.9), "red", "forestgreen")

# Start PDF device driver to save output to figure.pdf
pdf(file="C:/R/figure.pdf", height=3.5, width=5)

# Trim off excess margin space (bottom, left, top, right)
par(mar=c(4.2, 3.8, 0.2, 0.2))

# Graph autos using a y axis that uses the full range of value
# in autos_data. Label axes with smaller font and use larger
# line widths.
plot(autos_data$cars, type="l", col=plot_colors[1],
   ylim=range(autos_data), axes=F, ann=T, xlab="Days",
   ylab="Total", cex.lab=0.8, lwd=2)

# Make x axis tick marks without labels
axis(1, lab=F)

# Plot x axis labels at default tick marks with labels at
# 45 degree angle
text(axTicks(1), par("usr")[3] - 2, srt=45, adj=1,
         labels=c("Mon", "Tue", "Wed", "Thu", "Fri"),
         xpd=T, cex=0.8)

# Plot y axis with smaller horizontal labels
axis(2, las=1, cex.axis=0.8)

# Create box around plot
box()

# Graph trucks with thicker red dashed line
lines(autos_data$trucks, type="l", lty=2, lwd=2,
  col=plot_colors[2])

# Graph suvs with thicker green dotted line
lines(autos_data$suvs, type="l", lty=3, lwd=2,
  col=plot_colors[3])

# Create a legend in the top-left corner that is slightly
# smaller and has no border
legend("topleft", names(autos_data), cex=0.8, col=plot_colors,
   lty=1:3, lwd=2, bty="n")
```

```
# Turn off device driver (to flush output to PDF)
dev.off()

# Restore default margins
par(mar=c(5, 4, 4, 2) + 0.1)
```

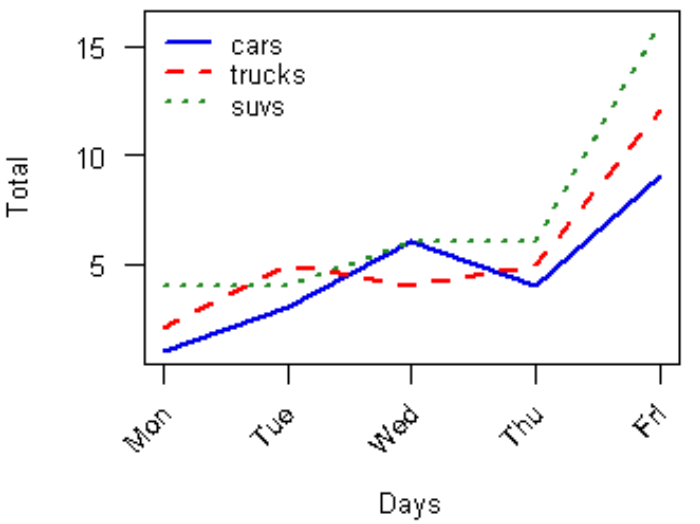


Figure 2.19: Data frame output example

Bar Charts

Let's start with a simple bar chart graphing the `cars` vector:

```
# Define the cars vector with 5 values
cars <- c(1, 3, 6, 4, 9)

# Graph cars
barplot(cars)
```

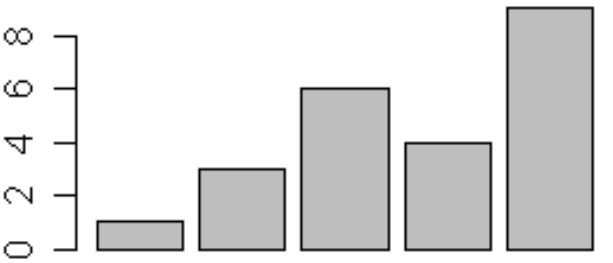


Figure 2.20: Data frame output example

Bar Chart with Gray Bars

Now let's read the auto data from the `autos.dat` data file, add labels, blue borders around the bars, and density lines:

```
# Read values from tab-delimited autos.dat
autos_data <- read.table("C:/R/autos.dat", header=T, sep="\t")

# Graph cars with specified labels for axes. Use blue
# borders and diagonal lines in bars.
barplot(autos_data$cars, main="Cars", xlab="Days",
   ylab="Total", names.arg=c("Mon","Tue","Wed","Thu","Fri"),
   border="blue", density=c(10,20,30,40,50))
```

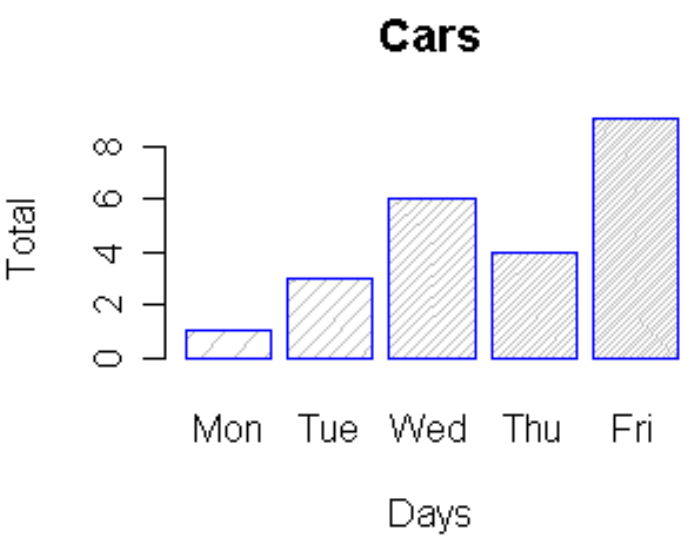


Figure 2.21: Data frame output example

Bar Chart with Lined Bars

Now let's graph the total number of autos per day using some color and show a legend:

```
# Read values from tab-delimited autos.dat
autos_data <- read.table("C:/R/autos.dat", header=T, sep="\t")

# Graph autos with adjacent bars using rainbow colors
barplot(as.matrix(autos_data), main="Autos", ylab= "Total",
   beside=TRUE, col=rainbow(5))

# Place the legend at the top-left corner with no frame
# using rainbow colors
legend("topleft", c("Mon","Tue","Wed","Thu","Fri"), cex=0.6,
   bty="n", fill=rainbow(5))
```

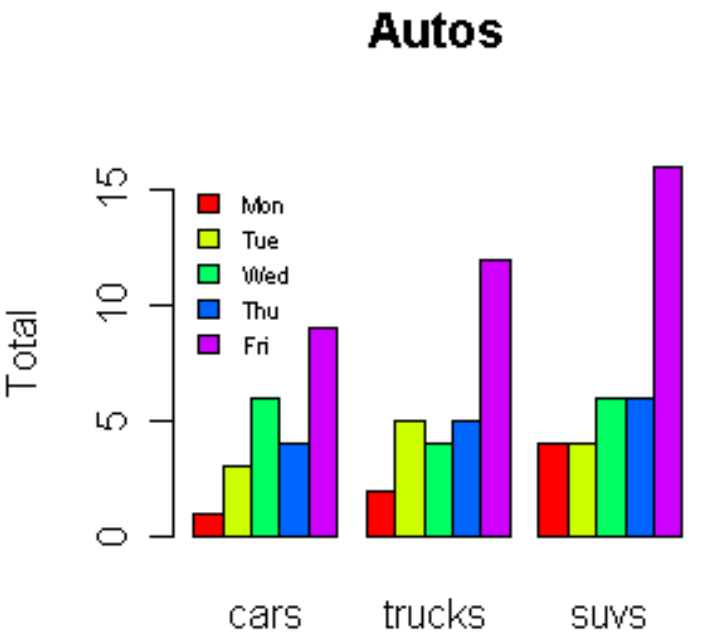


Figure 2.22: Data frame output example

Bar Chart with Colorful Bars

Let's graph the total number of autos per day using a stacked bar chart and place the legend outside of the plot area:

```
# Read values from tab-delimited autos.dat
autos_data <- read.table("C:/R/autos.dat", header=T, sep="\t")

# Expand right side of clipping rect to make room for the legend
par(xpd=T, mar=par()$mar+c(0,0,0,4))

# Graph autos (transposing the matrix) using heat colors,
# put 10% of the space between each bar, and make labels
# smaller with horizontal y-axis labels
barplot(t(autos_data), main="Autos", ylab="Total",
   col=heat.colors(3), space=0.1, cex.axis=0.8, las=1,
   names.arg=c("Mon","Tue","Wed","Thu","Fri"), cex=0.8)

# Place the legend at (6,30) using heat colors
legend(6, 30, names(autos_data), cex=0.8, fill=heat.colors(3))

# Restore default clipping rect
par(mar=c(5, 4, 4, 2) + 0.1)
```

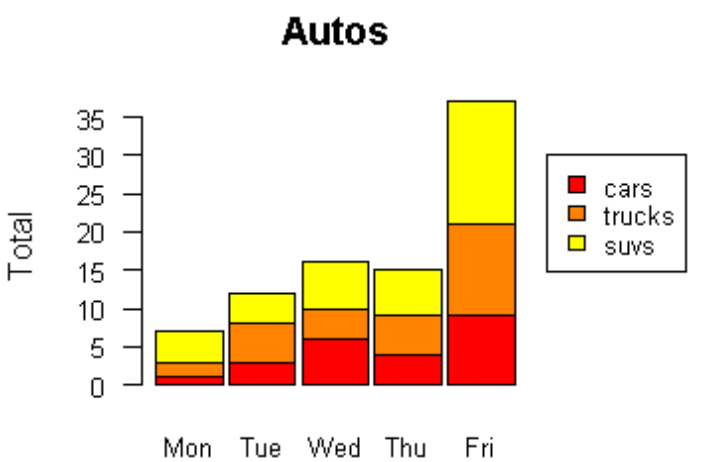


Figure 2.23: Data frame output example

Histograms

Let's start with a simple histogram graphing the distribution of the `suvs` vector:

```
# Define the suvs vector with 5 values
suvs <- c(4,4,6,6,16)
```

```
# Create a histogram for suvs
hist(suvs)
```

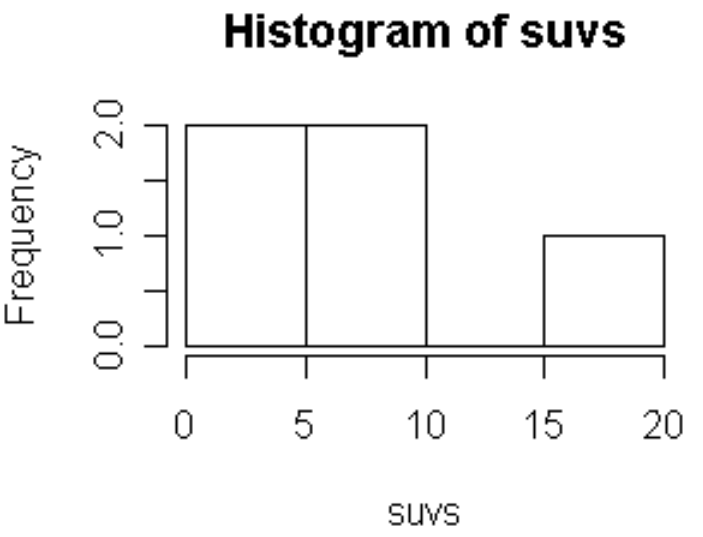


Figure 2.24: Data frame output example

Histogram of SUVs with White Bars

Let's now read the auto data from the `autos.dat` data file and plot a histogram of the combined car, truck, and SUV data in color.

```
# Read values from tab-delimited autos.dat
autos_data <- read.table("C:/R/autos.dat", header=T, sep="\t")

# Concatenate the three vectors
autos <- c(autos_data$cars, autos_data$trucks,
   autos_data$suvs)

# Create a histogram for autos in light blue with the y axis
# ranging from 0-10
hist(autos, col="lightblue", ylim=c(0,10))
```

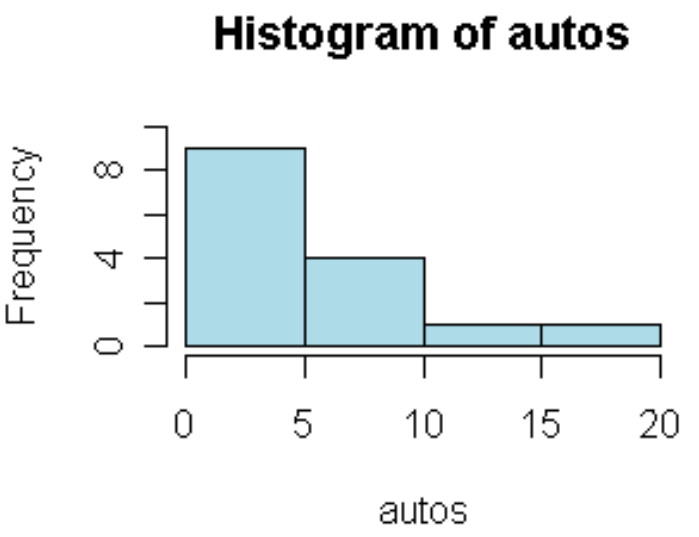


Figure 2.25: Data frame output example

Histogram of Autos with Blue Bars

Now change the breaks so none of the values are grouped together and flip the y-axis labels horizontally.

```
# Read values from tab-delimited autos.dat
autos_data <- read.table("C:/R/autos.dat", header=T, sep="\t")

# Concatenate the three vectors
autos <- c(autos_data$cars, autos_data$trucks,
   autos_data$suvs)

# Compute the largest y value used in the autos
max_num <- max(autos)

# Create a histogram for autos with fire colors, set breaks
# so each number is in its own group, make x axis range from
# 0-max_num, disable right-closing of cell intervals, set
# heading, and make y-axis labels horizontal
hist(autos, col=heat.colors(max_num), breaks=max_num,
   xlim=c(0,max_num), right=F, main="Autos Histogram", las=1)
```

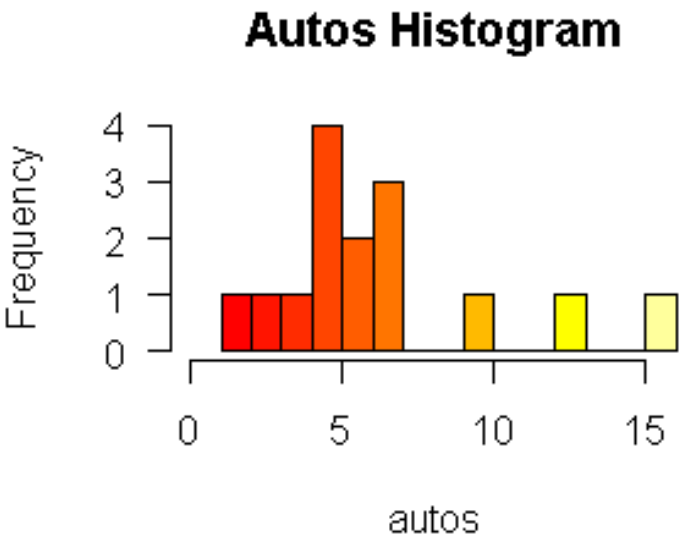


Figure 2.26: Data frame output example

Histogram of Autos with Different Groupings

Now let's create uneven breaks and graph the probability density.

```
# Read values from tab-delimited autos.dat
autos_data <- read.table("C:/R/autos.dat", header=T, sep="\t")

# Concatenate the three vectors
autos <- c(autos_data$cars, autos_data$trucks,
   autos_data$suvs)

# Compute the largest y value used in the autos
max_num <- max(autos)

# Create uneven breaks
brk <- c(0,3,4,5,6,10,16)

# Create a histogram for autos with fire colors, set uneven
# breaks, make x axis range from 0-max_num, disable right-
# closing of cell intervals, set heading, make y-axis labels
# horizontal, make axis labels smaller, make areas of each
# column proportional to the count
hist(autos, col=heat.colors(length(brk)), breaks=brk,
   xlim=c(0,max_num), right=F, main="Probability Density",
   las=1, cex.axis=0.8, freq=F)
```

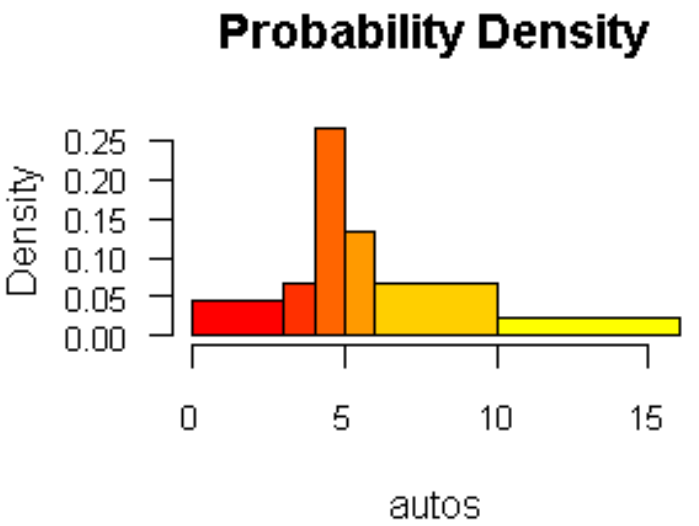


Figure 2.27: Data frame output example

Histogram of Autos with Probability Density

In this example, we'll plot the distribution of 1000 random values that have the log-normal distribution.

```
# Get a random log-normal distribution
r <- rlnorm(1000)

hist(r)
```

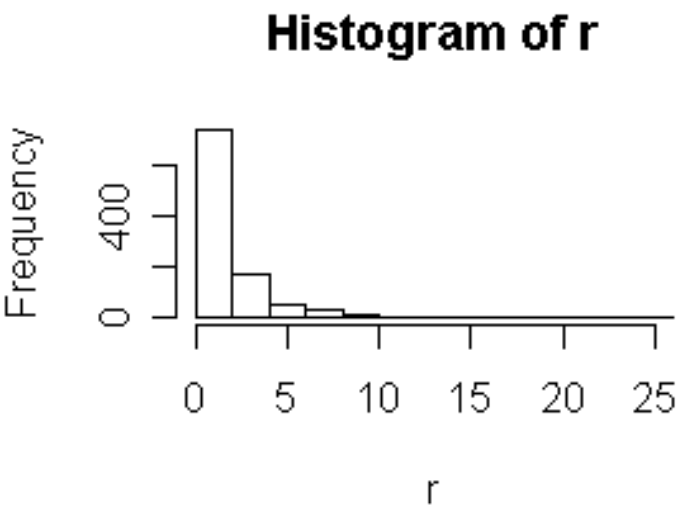


Figure 2.28: Data frame output example

Histogram Showing Log-Normal Distribution

Since log-normal distributions normally look better with log-log axes, let's use the `plot` function with points to show the distribution.

```
# Get a random log-normal distribution
r <- rlnorm(1000)

# Get the distribution without plotting it using tighter breaks
h <- hist(r, plot=F, breaks=c(seq(0,max(r)+1, .1)))

# Plot the distribution using log scale on both axes, and use
# blue points
plot(h$counts, log="xy", pch=20, col="blue",
       main="Log-normal distribution",
       xlab="Value", ylab="Frequency")
```

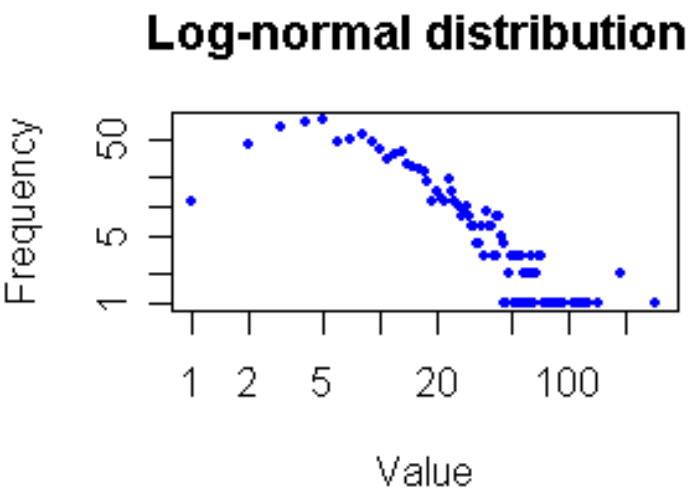


Figure 2.29: Data frame output example

Pie Charts

Let's start with a simple pie chart graphing the `cars` vector:

```
# Define cars vector with 5 values
cars <- c(1, 3, 6, 4, 9)

# Create a pie chart for cars
pie(cars)
```

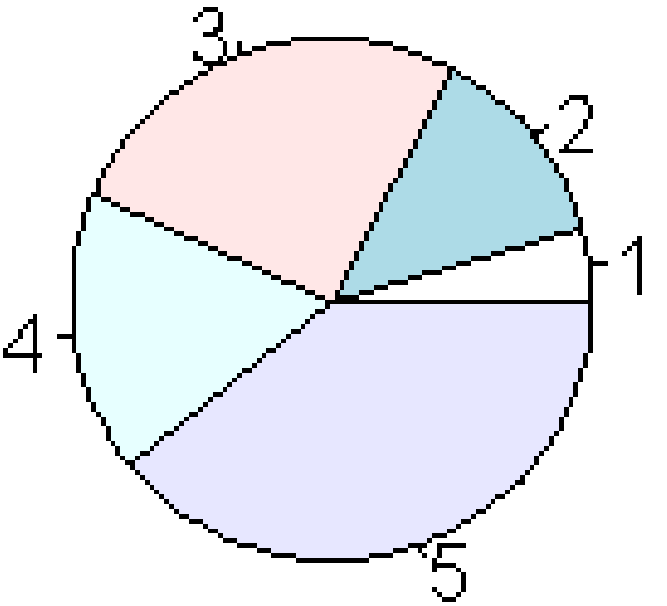


Figure 2.30: Data frame output example

Pie Chart of Cars

Now let's add a heading, change the colors, and define our own labels:

```
# Define cars vector with 5 values
cars <- c(1, 3, 6, 4, 9)

# Create a pie chart with defined heading and
# custom colors and labels
pie(cars, main="Cars", col=rainbow(length(cars)),
   labels=c("Mon","Tue","Wed","Thu","Fri"))
```

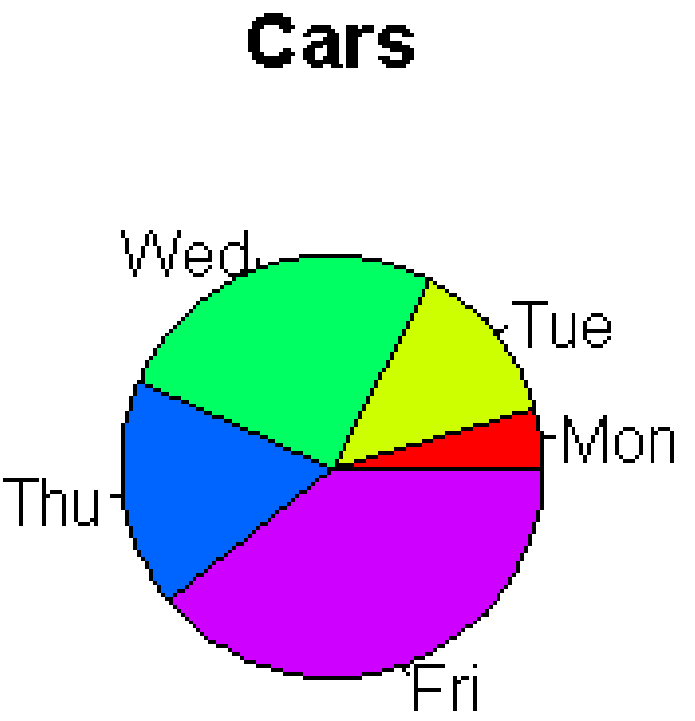


Figure 2.31: Data frame output example

Pie Chart with Rainbow Colors

Now let's change the colors, label using percentages, and create a legend:

```
# Define cars vector with 5 values
cars <- c(1, 3, 6, 4, 9)

# Define some colors ideal for black & white print
colors <- c("white","grey70","grey90","grey50","black")

# Calculate the percentage for each day, rounded to one
# decimal place
car_labels <- round(cars/sum(cars) * 100, 1)

# Concatenate a '%' char after each value
car_labels <- paste(car_labels, "%", sep="")

# Create a pie chart with defined heading and custom colors
# and labels
pie(cars, main="Cars", col=colors, labels=car_labels,
```

```
   cex=0.8)

# Create a legend at the right
legend(1.5, 0.5, c("Mon","Tue","Wed","Thu","Fri"), cex=0.8,
   fill=colors)
```

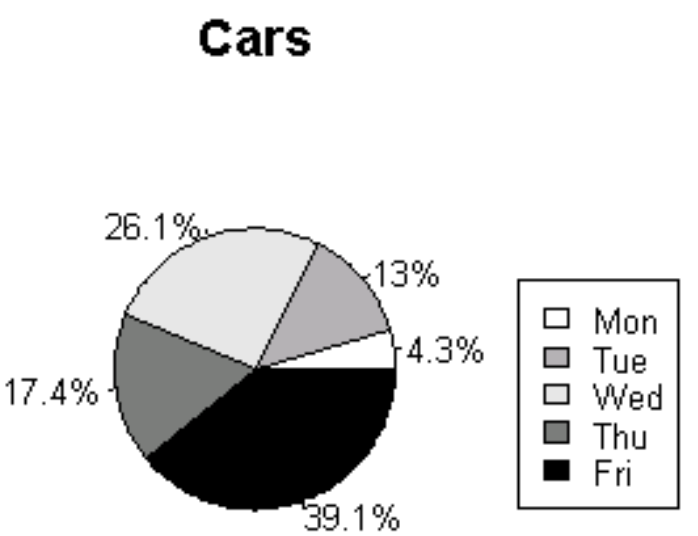


Figure 2.32: Data frame output example

Dotcharts

Let's start with a simple dotchart graphing the `autos` data:

```
# Read values from tab-delimited autos.dat
autos_data <- read.table("C:/R/autos.dat", header=T, sep="\t")

# Create a dotchart for autos
dotchart(t(autos_data))
```

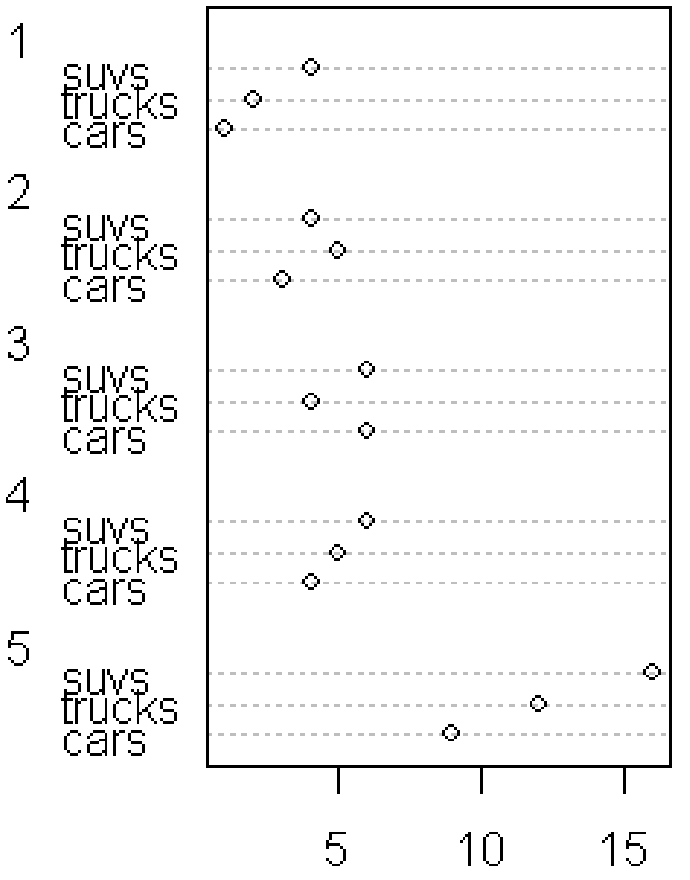


Figure 2.33: Data frame output example

Dotchart of autos.dat

Let's make the dotchart a little more colorful:

```
# Read values from tab-delimited autos.dat
autos_data <- read.table("C:/R/autos.dat", header=T, sep="\t")

# Create a colored dotchart for autos with smaller labels
dotchart(t(autos_data), color=c("red","blue","darkgreen"),
   main="Dotchart for Autos", cex=0.8)
```

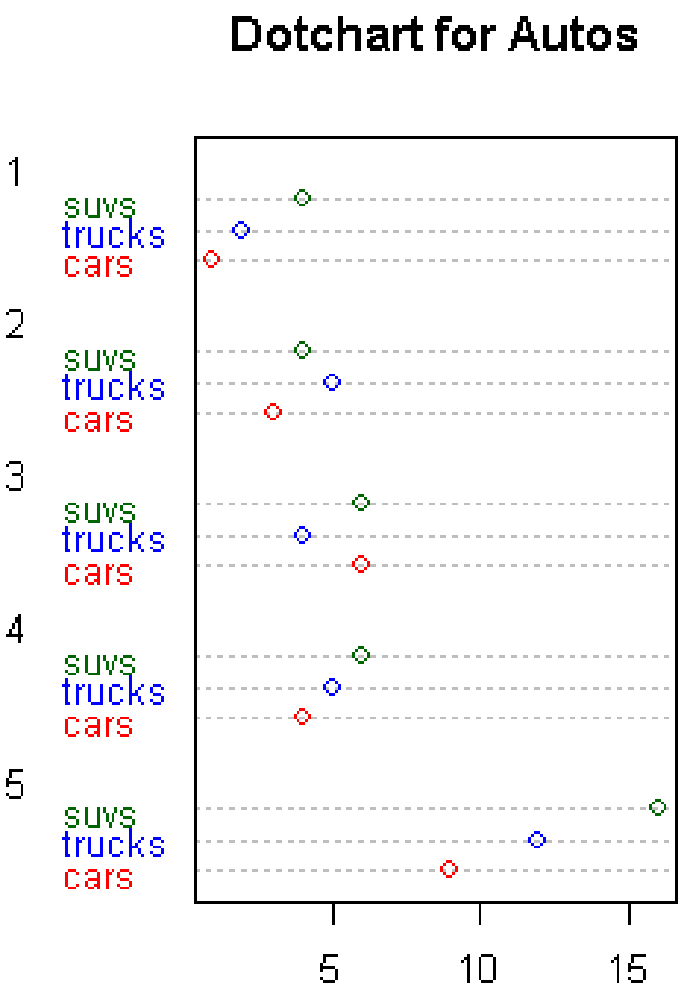


Figure 2.34: Data frame output example

Miscellaneous

This example shows all 25 symbols that you can use to produce points in your graphs:

```
# Make an empty chart
plot(1, 1, xlim=c(1,5.5), ylim=c(0,7), type="n", ann=FALSE)

# Plot digits 0-4 with increasing size and color
text(1:5, rep(6,5), labels=c(0:4), cex=1:5, col=1:5)
```

```
# Plot symbols 0-4 with increasing size and color
points(1:5, rep(5,5), cex=1:5, col=1:5, pch=0:4)
text((1:5)+0.4, rep(5,5), cex=0.6, (0:4))

# Plot symbols 5-9 with labels
points(1:5, rep(4,5), cex=2, pch=(5:9))
text((1:5)+0.4, rep(4,5), cex=0.6, (5:9))

# Plot symbols 10-14 with labels
points(1:5, rep(3,5), cex=2, pch=(10:14))
text((1:5)+0.4, rep(3,5), cex=0.6, (10:14))

# Plot symbols 15-19 with labels
points(1:5, rep(2,5), cex=2, pch=(15:19))
text((1:5)+0.4, rep(2,5), cex=0.6, (15:19))

# Plot symbols 20-25 with labels
points((1:6)*0.8+0.2, rep(1,6), cex=2, pch=(20:25))
text((1:6)*0.8+0.5, rep(1,6), cex=0.6, (20:25))
```

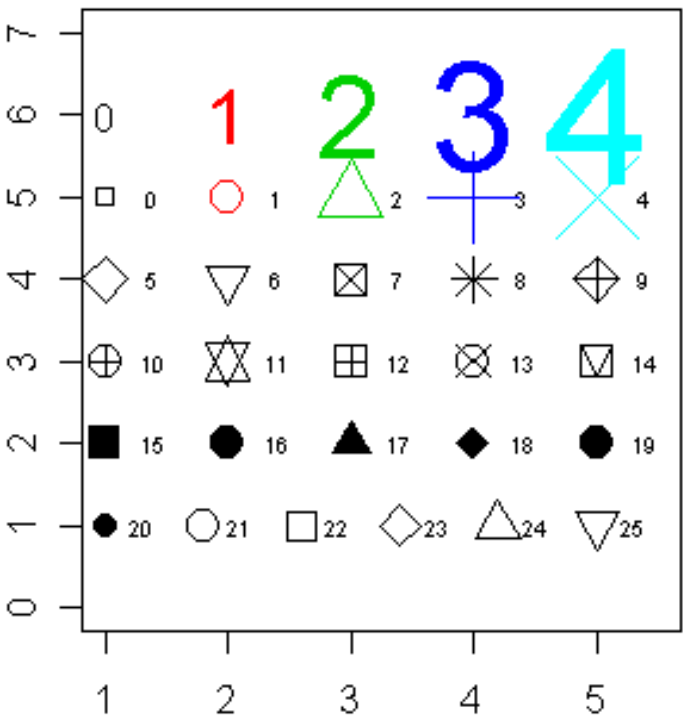


Figure 2.35: Data frame output example

Exercise Problems: R Graphics

1. Scatter Plot Customization

Create a scatter plot with customizations based on the following requirements:

- Generate a dataset of 100 random points where `x` is a sequence of numbers from 1 to 100 and `y` is normally distributed with a mean of 50 and standard deviation of 10.
- Use different colors for points based on whether they are above or below the mean of `y`.
- Add a horizontal line representing the mean of `y`.
- Customize the title, axis labels, and adjust the size of points.

Bonus: Use a legend to distinguish points above and below the mean.

2. Grouped Bar Plot of Product Sales

Using the following data, create a grouped bar plot to compare monthly sales of three products:

- `sales_data` (Data frame): Product A, Product B, and Product C sales for each month from January to June.

Requirements:

- Assign unique colors to each product.
- Label the x-axis with month names and the y-axis with "Sales".
- Add a legend at the top-right corner to indicate each product's color.

Bonus: Use a pattern (e.g., diagonal lines, cross-hatching) to differentiate each product in case of black-and-white printing.

3. Pie Chart for Survey Data

Given the following survey data, create a pie chart representing the distribution of different survey responses:

- Survey Responses: `Satisfied (40%)`, `Neutral (35%)`, and `Dissatisfied (25%)`

Requirements:

- Use distinct colors for each category.
- Label each slice with its category name and percentage.
- Add a title, and place the legend outside the pie chart.

Bonus: Add a custom color palette and adjust label positions to ensure they do not overlap with slices.

4. Histogram of Daily Temperatures

Using the following temperature data, create a histogram showing the distribution of daily temperatures:

- Generate a dataset of 200 random daily temperatures between 60°F and 100°F.

Requirements:

- Choose an appropriate number of bins to represent the data effectively.
- Add color to the histogram bars based on temperature ranges (e.g., `60-70: blue, 71-80: green, 81-90: orange, 91-100: red`).
- Customize the axis labels and title for clarity.

Bonus: Add a density curve overlay on the histogram to visualize the distribution more clearly.

5. Box Plot Comparison by Group

Use the `iris` dataset to create a box plot comparing `Sepal.Length` across the different species:

- Use different colors for each species.
- Customize the y-axis label as "Sepal Length" and the x-axis label as "Species".
- Add a title, and make the notches visible for each box.

Bonus: Add a summary table beneath the plot showing the mean and median `Sepal.Length` for each species.

Control Structures in R: for Loop, while Loop, and if-else Statements

In R programming, control structures play a vital role in managing the flow of execution, enabling the programmer to repeat tasks, perform actions based on conditions, and more. Here, we delve into three of the most commonly used control structures: the `for` loop, `while` loop, and `if-else` statements.

2.39 For Loop in R

The `for` loop in R is a control flow statement that allows you to iterate over a sequence (such as a vector, list, or other collections) and execute a block of code for each element in that sequence. This is particularly useful for performing repetitive tasks without manually writing out each iteration.

2.39.1 Syntax of the For Loop

The basic syntax of a `for` loop in R is as follows:

```
for (variable in sequence) {
    % Code to be executed for each element in the sequence
}
```

- **variable**: A temporary variable that takes on the value of each element in the sequence during each iteration of the loop.
- **sequence**: This can be a vector, list, or any other collection that you want to iterate over.

2.39.2 Example 1: Basic For Loop

Here's a simple example that prints numbers from 1 to 5:

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

Output:

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

```
[1] 5
```

2.39.3 Example 2: Calculating the Sum of a Vector

You can use a `for` loop to perform calculations, such as summing the elements of a numeric vector:

```
numbers <- c(10, 20, 30, 40, 50)
sum_result <- 0

for (num in numbers) {
    sum_result <- sum_result + num
}

print(sum_result)
```

Output:

```
[1] 150
```

2.39.4 Example 3: Iterating Over a List

You can also use a `for` loop to iterate over elements in a list. Here's an example that prints the names and ages from a list:

```
people <- list(name = c("Alice", "Bob", "Charlie"), age = c(25, 30,
    35))

for (i in 1:length(people$name)) {
    cat("Name:", people$name[i], "- Age:", people$age[i], "\n")
}
```

Output:

```
Name: Alice - Age: 25
Name: Bob - Age: 30
Name: Charlie - Age: 35
```

2.39.5 Example 4: Creating a Multiplication Table

You can use a `for` loop to create a multiplication table for a specific number:

```
number <- 7
```

```
cat("Multiplication Table for", number, ":\n")
for (i in 1:10) {
    cat(number, "*", i, "=", number * i, "\n")
}
```

Output:

```
Multiplication Table for 7 :
7 * 1 = 7
7 * 2 = 14
7 * 3 = 21
7 * 4 = 28
7 * 5 = 35
7 * 6 = 42
7 * 7 = 49
7 * 8 = 56
7 * 9 = 63
7 * 10 = 70
```

2.39.6 Example 5: Nested For Loops

You can nest `for` loops to perform operations on multidimensional data. Here's an example of creating a 3x3 multiplication table:

```
for (i in 1:3) {
    for (j in 1:3) {
        cat(i, "*", j, "=", i * j, "\t")
    }
    cat("\n") % New line after each row
}
```

Output:

```
1 * 1 = 1 1 * 2 = 2 1 * 3 = 3
2 * 1 = 2 2 * 2 = 4 2 * 3 = 6
3 * 1 = 3 3 * 2 = 6 3 * 3 = 9
```

2.39.7 Example 6: Using a For Loop with Conditional Statements

You can incorporate `if` statements inside a `for` loop to filter values. For example, to print only even numbers from a vector:

```
numbers <- 1:10

for (num in numbers) {
    if (num %% 2 == 0) {
        cat(num, "is even\n")
    }
}
```

Output:

```
2 is even
4 is even
6 is even
8 is even
10 is even
```

2.39.8 Summary

The `for` loop in R is a powerful tool for automating repetitive tasks, whether you are performing calculations, processing lists, or generating formatted outputs. By understanding how to use `for` loops effectively, you can enhance your data manipulation and analysis capabilities in R.

2.40 While Loop in R

The `while` loop in R is a control flow structure that allows you to repeat a block of code as long as a specified condition evaluates to `TRUE`. It is useful when the number of iterations is not predetermined and depends on dynamic conditions evaluated at runtime.

2.40.1 Syntax of `while` Loop

The basic syntax for a `while` loop in R is as follows:

```
while (condition) {
  # Code to be executed repeatedly
}
```

- **condition**: A logical expression that is evaluated before each iteration. If the condition is `TRUE`, the loop continues; if it is `FALSE`, the loop stops.
- The code block within the braces {} will be executed repeatedly as long as the condition remains `TRUE`.

2.40.2 Characteristics of `while` Loop

- The condition is checked **before** executing the loop body, which means if the condition is `FALSE` from the beginning, the loop body will not execute at all.
- It is crucial to ensure that the condition will eventually evaluate to `FALSE` to avoid creating an infinite loop.

2.40.3 Examples of `while` Loop

1. Basic Countdown

This example demonstrates a simple countdown from 10 to 1.

```
countdown <- 10
while (countdown > 0) {
  print(countdown)
  countdown <- countdown - 1
}
print("Liftoff!")
```

Explanation:

- The loop starts with `countdown` set to 10.
- It prints the value of `countdown` and then decrements it by 1 until it reaches 0.
- Once `countdown` is 0, it exits the loop and prints "Liftoff!"

2. User Input Until Condition Met

In this example, we will prompt the user to enter numbers until they enter a negative number.

```
sum <- 0
number <- 0

while (number >= 0) {
  number <- as.numeric(readline(prompt = "Enter a number (negative to
      stop): "))
  sum <- sum + number
}
cat("Total sum:", sum, "\n")
```

Explanation:

- The loop continues as long as the user enters a non-negative number.
- The numbers entered by the user are summed up, and when a negative number is entered, the loop terminates, and the total sum is printed.

3. Collatz Conjecture

This example implements the Collatz Conjecture using a `while` loop.

```
collatz_steps <- function(n) {
  steps <- 0
  while (n != 1) {
    if (n %% 2 == 0) {
      n <- n / 2 # If n is even
    } else {
      n <- 3 * n + 1 # If n is odd
    }
    steps <- steps + 1
  }
  return(steps)
}

number <- 7
cat("Number of steps for", number, "to reach 1:",
    collatz_steps(number), "\n")
```

Explanation:

- The function `collatz_steps` takes a positive integer `n` and applies the Collatz rules until `n` becomes 1.
- The number of steps taken is counted and returned.

4. Infinite Loop with Exit Condition

Here's an example of creating a loop that runs indefinitely until a certain condition is met.

```
repeat {
  response <- readline(prompt = "Type 'exit' to stop: ")
  if (response == "exit") {
    cat("Exiting the loop.\n")
    break # Exit the loop
  }
  cat("You typed:", response, "\n")
}
```

Explanation:

- This loop will keep running until the user types "exit".
- The `break` statement is used to exit the loop when the condition is met.

2.40.4 Important Notes

- Be cautious when using `while` loops to avoid infinite loops. Ensure that there is a clear exit condition and that the condition will eventually be met.
- Use `break` to exit the loop prematurely based on a specific condition if necessary.
- You can combine `while` loops with other control flow statements (like `if` statements) to create complex logic as required by your application.

2.40.5 Conclusion

The `while` loop is a powerful and flexible tool in R that allows for repetitive execution of code based on dynamic conditions. Understanding how to use it effectively is essential for programming in R, especially for tasks that require iterative processes based on user input or calculated conditions.

2.41 If-Else Statements in R

The **if-else statement** in R is a fundamental control structure that allows you to execute different blocks of code based on specific conditions. This is particularly useful for decision-making within your code, enabling you to respond differently to varying inputs or situations.

2.41.1 Basic Structure

The basic syntax of an if-else statement in R is as follows:

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

2.41.2 Explanation of Components

- **condition**: This is a logical expression that evaluates to either `TRUE` or `FALSE`. If the condition is `TRUE`, the code within the first block is executed. If it's `FALSE`, the code within the `else` block is executed.
- **Code Blocks**: The code inside the {} braces is the block of code that gets executed based on the evaluation of the condition. You can have multiple lines of code in each block.

2.41.3 Extended If-Else Statement

You can also extend the if-else structure using `else if` to handle multiple conditions:

```
if (condition1) {
  # Code to execute if condition1 is TRUE
} else if (condition2) {
  # Code to execute if condition2 is TRUE
} else {
  # Code to execute if all conditions are FALSE
}
```

2.41.4 Examples

Example 1: Basic If-Else

```
x <- 10

if (x > 5) {
  print("x is greater than 5")
} else {
  print("x is less than or equal to 5")
}
```

Output:

```
[1] "x is greater than 5"
```

In this example, since `x` is 10, the condition `x > 5` evaluates to `TRUE`, and the corresponding message is printed.

Example 2: If-Else with Multiple Conditions

```
x <- 15

if (x < 10) {
  print("x is less than 10")
} else if (x >= 10 && x < 20) {
  print("x is between 10 and 20")
} else {
  print("x is 20 or more")
}
```

Output:

```
[1] "x is between 10 and 20"
```

In this example, `x` is 15, which satisfies the second condition (`x >= 10 && x < 20`), so the corresponding message is printed.

Example 3: Checking for Even or Odd

```
num <- 7

if (num %% 2 == 0) {
  print("The number is even.")
} else {
  print("The number is odd.")
}
```

Output:

```
[1] "The number is odd."
```

Here, the modulus operator (`%%`) checks if `num` is even or odd. Since 7 is not divisible by 2, the message indicates it is odd.

Example 4: Grading System

```
score <- 85

if (score >= 90) {
  grade <- "A"
} else if (score >= 80) {
  grade <- "B"
} else if (score >= 70) {
  grade <- "C"
```

```
} else if (score >= 60) {
  grade <- "D"
} else {
  grade <- "F"
}

print(paste("Your grade is:", grade))
```

Output:

```
[1] "Your grade is: B"
```

In this grading example, the score is evaluated against several thresholds to determine the corresponding letter grade.

Example 5: Using Logical Operators

```
age <- 25
has_license <- TRUE

if (age >= 18 && has_license) {
  print("You can drive.")
} else {
  print("You cannot drive.")
}
```

Output:

```
[1] "You can drive."
```

In this example, both the age and license status are checked. Since the conditions are met, the message indicates that the user can drive.

2.41.5 Summary

If-else statements are a powerful tool in R programming, allowing for conditional execution of code based on logical conditions. They can be used in various scenarios, from simple checks to complex decision-making processes. By using `if`, `else if`, and `else`, you can create more sophisticated logic flows in your R scripts.

Exercise Problems

1. For Loop Exercises

1. **Sum of Squares**
   - Write a `for` loop that calculates the sum of the squares of the integers from 1 to 20 and prints the result.
2. **Fibonacci Sequence**
   - Generate the first 10 Fibonacci numbers using a `for` loop and print them in a single line.
3. **Factorial Calculation**
   - Create a `for` loop that calculates the factorial of a given number n (e.g., n 5) and print the result.
4. **Multiplication Table**
   - Using a `for` loop, create a multiplication table for the number 7 (from 1 to 10) and print it in a formatted way.
5. **Character Count**
   - Write a `for` loop that counts the number of vowels in a given string (e.g., "Hello, World!") and prints the count.

2. While Loop Exercises

1. **Countdown**
   - Write a `while` loop that counts down from 10 to 1 and prints each number. Print "Liftoff!" after reaching 0.
2. **Sum of Numbers**
   - Create a `while` loop that continually asks the user to input numbers until they enter a negative number. Calculate and print the sum of the entered numbers (excluding the negative number).
3. **Collatz Conjecture**
   - Implement the Collatz Conjecture using a `while` loop: Starting from a positive integer n, keep applying the following rules until n becomes 1:

  - If n is even, divide it by 2.
  - If n is odd, multiply it by 3 and add 1.

  Print the number of steps taken to reach 1.

4. **Guess the Number**

   - Write a simple number guessing game using a `while` loop. Randomly generate a number between 1 and 100, and prompt the user to guess the number. Provide feedback on whether their guess is too high or too low, and continue until they guess correctly.

5. **Infinite Loop Control**

   - Write a `while` loop that runs indefinitely and prints "Running..." until the user types "stop". Use `readline()` to capture user input to control the loop.

3. If-Else Statement Exercises

1. **Even or Odd**

   - Write a program that checks if a number input by the user is even or odd, and prints the result.

2. **Grade Calculator**

   - Create a program that takes a numerical score (0 to 100) from the user and prints the corresponding letter grade based on the following scale:
     - A: 90-100
     - B: 80-89
     - C: 70-79
     - D: 60-69
     - F: below 60

3. **Age Checker**

   - Write an `if-else` statement that checks a person's age and prints whether they are a child (0-12), teenager (13-19), adult (20-64), or senior (65+).

4. **Discount Eligibility**

- Create a program that checks if a customer is eligible for a discount based on their membership status (member or non-member) and the total purchase amount (e.g., discounts for members over $50).

5. **Password Strength Checker**

   - Write an `if-else` statement to check the strength of a password based on the following criteria:
     - Length: at least 8 characters
     - Contains at least one uppercase letter
     - Contains at least one lowercase letter
     - Contains at least one digit

     Print whether the password is strong, moderate, or weak based on these criteria.

Additional Challenge Problems

1. **Combining Control Structures**

   - Write a program that takes a list of numbers (e.g., from 1 to 50) and uses a `for` loop and an `if` statement to print only the prime numbers from that list.

2. **Pattern Printing**

   - Create a nested `for` loop that prints the following pattern:

```
*
**
***
****
*****
```

3. **Bank Account Simulation**

   - Implement a simple bank account simulation where a user can deposit or withdraw money. Use a `while` loop to continue prompting the user until they choose to exit. Use `if-else` statements to handle invalid transactions (e.g., overdrafts).

This code uses a simple enumerated list without specifying labels, which should resolve the issue you encountered. You can now copy and paste this into your LaTeX document. Let me know if you need any further adjustments!

2.42 Working with Packages

Packages enhance R's capabilities, providing additional functions.

- How to find and use R packages.
- Overview of essential packages for statistics (e.g., `MASS`, `caret`, `tidyverse`).
- How to check package documentation and help files.

2.43 R Markdown

R Markdown is a powerful tool for creating dynamic reports.

- Creating dynamic reports that combine code and output.
- Exporting reports to different formats (HTML, PDF, Word).

Calculating Weighted Mean for Grouped Data

Suppose you have a dataset representing a set of observations, and you want to analyze the distribution of values within specific class intervals. The data points are divided into intervals, and you want to calculate the weighted mean using the midpoints of these intervals.

1. **Given Data:**
   - A set of data points: $\{5, 15, 25, 35, 45, 55, 65\}$.
   - Class intervals defined as $0, 10, 20, 30, 40, 50, 60, 70$ with a step size of 10.
2. **Objective:**
   - Utilize the provided R code to create class intervals, calculate midpoints, and determine the weighted mean for the grouped data.
3. **Procedure:**
   - Use the `cut` function in R to categorize the data into intervals based on the specified class intervals.
   - Calculate the midpoints of these intervals.
   - Compute the weighted mean using the formula $\frac{\sum(f \cdot x)}{\sum f}$, where f is the frequency and x is the midpoint of the interval.

4. **Results:**
    - Display the resulting data frame, showing the original data points along with their corresponding class intervals.
    - Print the calculated weighted mean using the midpoints.
5. **Conclusion:**
    - Interpret the results and draw conclusions about the distribution of the data within the defined intervals.

R Code

Listing 2.1: R Code for Calculating Weighted Mean

```
# Create some example data
data <- c(5, 15, 25, 35, 45, 55, 65)

# Define the class intervals
class_intervals <- seq(0, 70, by = 10)

# Use cut to create intervals
interval_labels <- cut(data, breaks = class_intervals,

 include.lowest = TRUE, right = FALSE)

# Calculate midpoints
midpoints <- (class_intervals[-1] +

class_intervals[-length(class_intervals)]) / 2

# Calculate the weighted mean
weighted_mean <- sum(midpoints * table(interval_labels)) /
    length(data)

# Display the result
data_frame <- data.frame(Data = data, Class_Interval =
    interval_labels)
print(data_frame)

# Print the mean using midpoints
cat("Weighted Mean using Midpoints:", weighted_mean, "\n")
```

2.43.1 Exercise

Write an R program to find the median and mode of continuously grouped data.

3. Module 1: Descriptive Statistics and Probability

3.1 Measures of Central Tendency

Measures of central tendency describe the central point of a data set. Common measures include:

Mean: The average of the data.
Median: The middle value of the data.
Mode: The most frequently occurring value in the data.

We will compute these for both discrete and continuous data sets using R.

3.1.1 Individual/Ungrouped Data

Example Data Set: Let's consider a simple discrete data set representing the number of books read by a group of students:

```
# Discrete data: Number of books read by students
books_read <- c(2, 4, 5, 3, 4, 2, 6, 5, 3, 4)
```

Mean Calculation

The mean (average) is calculated as the sum of all values divided by the number of observations.

$$\text{The mean is calculated as: Mean } \frac{\text{Sum of all values}}{\text{Number of observations}}$$

```
# Calculate the mean
mean_books <- mean(books_read)
print(paste("Mean:", mean_books))
```

```
Output:
  [1] "Mean: 3.8"
```

Median Calculation

The median is the middle value when the data is sorted. If the number of observations is even, it is the average of the two middle values.

```
# Calculate the median
median_books <- median(books_read)
print(paste("Median:", median_books))
```

```
Output:
[1] "Median: 4"
```

Mode Calculation

The mode is the most frequently occurring value in the data. R does not have a built-in function for mode, so we use a custom function.

```
## Function to calculate mode for ungrouped/raw data
mode_raw_data <- function(data) {
  freq_table <- table(data) # Create a frequency table
  mode_values <- as.numeric(names(freq_table[freq_table ==
      max(freq_table)]))
  if (length(mode_values) > 1) {
    cat("The mode(s) for the raw data:", mode_values, "\n")
  } else if (length(mode_values) == 1) {
    cat("The mode for the raw data is:", mode_values, "\n")
  } else {
    cat("No mode exists for the raw data.\n")
  }
}
# Example Inputs and Usage
# Ungrouped/raw data
raw_data <- c(90, 70, 50, 30, 40, 86, 65, 73, 68, 90, 90, 10, 73, 25,
    35, 88, 67, 80, 74, 46)
mode_raw_data(raw_data)
```

```
Output:
The mode for the raw data is: 90
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
\section{Grouped - Discrete Data}
\subsection*{Mean Calculations}
\begin{lstlisting}[language=R]
```

```
# Given Data discrete (X and Frequencies)
val=c(1,2,3,4,5) # X values
fr=c(12,11,23,12,11) # Frequemcies
# Calculation of mean
calculate_mean_discrete <- function(values, frequencies, A = NULL)
  {
  #Direct Method
  total_weighted_values <- sum(values * frequencies) # (f_i * x_i)
  total_frequencies <- sum(frequencies) # N = Total number of
      observations
  mean_direct <- tot  al_weighted_values / total_frequencies # Direct
      mean

  # Short-Cut Method
  A <- mean(val) # Choose a default arbitrary value if A is not
      provided
  deviations <- val - A # d_i = x_i - A
  total_deviation <- sum(fr * deviations) # (f_i * d_i)
  mean_shortcut <- A + (total_deviation / total_frequencies) #
      Shortcut mean
  # Return the results
  return(list(mean_direct = mean_direct, mean_shortcut =
      mean_shortcut))
}
calculate_mean_discrete(val,fr)
```

```
Output:
$mean_direct
[1] 2.985507

$mean_shortcut
[1] 2.985507
```

Median Calculations

```
# Function to calculate the median for discrete grouped data
calculate_median <- function(x, f) {
  # x: A vector of values
  # f: A vector of frequencies
  # Step 1: Calculate cumulative frequencies
  cf <- cumsum(f)
  # Step 2: Compute (N+1)/2
  N <- sum(f)
  median_position <- (N + 1) / 2
```

```
  # Step 3: Find the group corresponding to the median position
  median_index <- which(cf >= median_position)[1]
  # Step 4: Identify the corresponding x value
  median <- x[median_index]
  # Return the median
  return(median)
}

# Example usage with the provided data
weights <- c(10, 20, 30, 40, 50, 60, 70) # Weight (kg)
frequencies <- c(4, 7, 12, 15, 13, 5, 4) # Number of students

# Calculate the median weight
median_weight <- calculate_median(weights, frequencies)
print(paste("The median weight is:", median_weight))
```

```
Output:
[1] "The median weight is: 40"
```

Mode Calculations

```
# Function to calculate mode for discrete frequency distribution
mode_discrete <- function(values, frequencies) {
  mode_value <- values[which.max(frequencies)] # Find value
      corresponding to max frequency
  cat("The mode for the discrete frequency distribution is:",
      mode_value, "\n")
}
# Discrete frequency distribution
values <- c(6, 7, 8, 9, 10) # Days of confinement
frequencies <- c(4, 6, 7, 5, 3) # Number of patients
mode_discrete(values, frequencies)
```

```
Output:
The mode for the discrete frequency distribution is: 8
```

3.2 Grouped - Continuous data

Mean Calculations

```
# Continuous data
class_intervals <- c("10-20", "20-30", "30-40") # Class intervals
f <- c(5, 7, 8)                                 # Frequencies

# Calculate midpoints
lower_limits <- c(10, 20, 30)
upper_limits <- c(20, 30, 40)
midpoints <- (lower_limits + upper_limits) / 2

# Calculate mean
mean_continuous <- sum(f * midpoints) / sum(f)

# Output
mean_continuous
```

```
Output:
[1] 26.5
```

Step Deviation Method for Mean Calculations

```
# Function to calculate the mean for continuous grouped data
calculate_mean_continuous <- function(lower_limits, frequencies,
    class_width, A = NULL) {
  # Midpoints of each class interval
  midpoints <- lower_limits + (class_width / 2) # x_i = Lower limit +
      (width / 2)
  # Direct Method
  total_weighted_midpoints <- sum(frequencies * midpoints) # (f_i *
      x_i)
  total_frequencies <- sum(frequencies) # N = Total number of
      observations
  mean_direct <- total_weighted_midpoints / total_frequencies #
      Direct mean
  # Short-Cut Method
  if (is.null(A)) {
    A <- midpoints[which.max(frequencies)] # Choose midpoint of the
        highest frequency class as A
  }
  deviations <- (midpoints - A) / class_width # d_i = (x_i - A) /
      class_width
  total_deviation <- sum(frequencies * deviations)
  mean_shortcut <- A + (total_deviation / total_frequencies) *
      class_width # Shortcut mean
```

```
  # Return the results
  return(list(mean_direct = mean_direct, mean_shortcut =
      mean_shortcut))
}
#Input
lower_limits=c(10, 20, 30, 40, 50)
frequencies= c(5, 7, 8, 6, 4)
class_width=lower_limits[2]-lower_limits[1]
#Output:
calculate_mean_continuous(lower_limits, frequencies, class_width)
```

```
Output:
$mean_direct
[1] 34

$mean_shortcut
[1] 34
```

Median Calculations

```
# Function to calculate the median for continuous grouped data
calculate_median_continuous <- function(lower_limits, frequencies,
    class_interval) {
  # lower_limits: A vector of lower limits of class intervals
  # frequencies: A vector of frequencies
  # class_interval: The class width (assumes equal width for all
      intervals)

  # Step 1: Calculate cumulative frequencies
  cf <- cumsum(frequencies)

  # Step 2: Compute N/2
  N <- sum(frequencies)
  median_position <- N / 2

  # Step 3: Find the median class
  median_class_index <- which(cf >= median_position)[1]

  # Step 4: Extract required values
  l <- lower_limits[median_class_index] # Lower limit of median class
  f <- frequencies[median_class_index] # Frequency of median class
  m <- ifelse(median_class_index == 1, 0, cf[median_class_index - 1])
      # Cumulative freq before median class
```

```
  c <- class_interval # Class interval width

  # Step 5: Apply the median formula
  median <- l + ((median_position - m) / f) * c

  # Return the median
  return(median)
}

# Example 1: Median weight of apples
lower_limits_apples <- c(410, 420, 430, 440, 450, 460, 470) # Lower
    limits of weight intervals
frequencies_apples <- c(14, 20, 42, 54, 45, 18, 7) # Frequencies
class_interval_apples <-
    lower_limits_apples[2]-lower_limits_apples[1] # Class interval
    width
median_apples <- calculate_median_continuous(lower_limits_apples,
    frequencies_apples, class_interval_apples)
print(paste("The median weight of apples is:", median_apples))
```

```
Output:
[1] "The median weight of apples is: 444.444"
```

Mode Calculations

```
# Function to calculate mode for continuous data
mode_continuous <- function(lower_limits, frequencies, class_width) {
  # Find modal class
  modal_index <- which.max(frequencies)
  l <- lower_limits[modal_index] # Lower limit of modal class
  f1 <- frequencies[modal_index] # Frequency of modal class
  f0 <- ifelse(modal_index > 1, frequencies[modal_index - 1], 0) #
      Frequency of class before modal
  f2 <- ifelse(modal_index < length(frequencies),
      frequencies[modal_index + 1], 0) # Frequency of class after
      modal

  # Apply mode formula
  mode <- l + ((f1 - f0) / (2 * f1 - f0 - f2)) * class_width
  cat("The mode for the continuous data is:", mode, "\n")
}
# Continuous data
# Example Input for Income Data
```

```
lower_limits <- c(0, 100, 200, 300, 400, 500, 600) # Lower limits of
    income ranges
frequencies <- c(5, 7, 12, 18, 16, 10, 5) # Number of persons in each
    income range
class_width <- 100 # Width of each class interval (difference between
    lower limits)

mode_continuous(lower_limits, frequencies, class_width)
```

```
Output:
The mode for the continuous data is: 375
```

3.2.1 Exercises

1. Create your own discrete data set and calculate the mean, median, and mode using R.
2. Generate a continuous data set and display its summary statistics.
3. Plot a histogram for your continuous data and describe the most frequent interval.

3.3 Measures of dispersion

provide insights into the spread or variability of a data set. Common measures include:

Standard Deviation (SD): Quantifies the amount of variation in a set of data values.

Coefficient of Variation (CV): Expresses the standard deviation as a percentage of the mean, allowing for comparison between data sets with different units or scales.

3.3.1 Discrete Data

Example Data Set: We will use a data set representing the number of sales made by a group of salespeople:

```
# Discrete data: Number of sales made
sales <- c(15, 20, 22, 18, 25, 20, 19, 23, 21, 17)
```

Standard Deviation Calculation

The standard deviation measures how spread out the sales data is.

```
# Calculate the standard deviation
sd_sales <- sd(sales)
print(paste("Standard Deviation of Sales:", sd_sales))

Output:
[1] "Standard Deviation of Sales: 3.171"
```

Coefficient of Variation Calculation

The CV is used to compare the relative variability of data sets.

```
# Calculate the mean
mean_sales <- mean(sales)

# Calculate the coefficient of variation
cv_sales <- (sd_sales / mean_sales) * 100
print(paste("Coefficient of Variation of Sales (%):", round(cv_sales,
    2)))

Output:
[1] "Coefficient of Variation of Sales (%): 15.09"
```

Summary Statistics

```
# Display summary statistics
summary(sales)

Output:
   Min. 1st Qu. Median   Mean 3rd Qu.   Max.
   15.0    18.5   20.0   20.0    22.5   25.0
```

3.3.2 Continuous Data

Example Data Set: We will use a data set representing the weights (in kg) of a sample of products:

```
# Continuous data: Weights of products (in kg)
weights <- c(50.2, 52.5, 49.8, 51.1, 50.5, 53.0, 48.9, 51.7, 50.8,
    49.5)
```

Standard Deviation Calculation

```
# Calculate the standard deviation
sd_weights <- sd(weights)
print(paste("Standard Deviation of Weights:", sd_weights))

Output:
[1] "Standard Deviation of Weights: 1.373"
```

Coefficient of Variation Calculation

```
# Calculate the mean
mean_weights <- mean(weights)

# Calculate the coefficient of variation
cv_weights <- (sd_weights / mean_weights) * 100
print(paste("Coefficient of Variation of Weights (%):",
    round(cv_weights, 2)))

Output:
[1] "Coefficient of Variation of Weights (%): 2.68"
```

Summary Statistics

```
# Display summary statistics
summary(weights)

Output:
   Min. 1st Qu. Median   Mean 3rd Qu.   Max.
   48.9    50.2   50.9   50.8    51.5   53.0
```

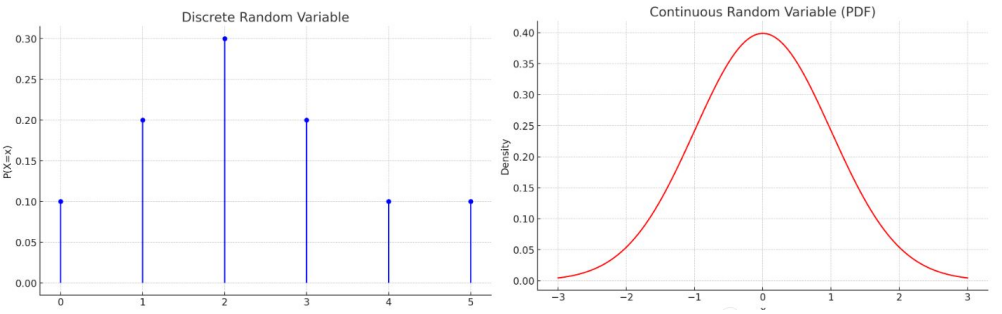
3.3.3 Exercises

1. Create your own discrete data set and compute the standard deviation and CV using R.
2. Generate a continuous data set and calculate its standard deviation and CV.
3. Compare the variability of two different data sets using the CV.

4. Module 2: Modelling with Probability distributions

4.1 Random Variables – Discrete & Continuous

```
# Discrete Random Variable
x_discrete <- 0:5
p_discrete <- c(0.1, 0.2, 0.3, 0.2, 0.1, 0.1) # Probabilities must
    sum to 1
plot(x_discrete, p_discrete, type = "h", main = "Discrete Random
    Variable",
     xlab = "x", ylab = "P(X=x)", col = "blue", lwd = 2)

# Continuous Random Variable
curve(dnorm(x, mean = 0, sd = 1), from = -3, to = 3, main =
    "Continuous Random Variable",
      xlab = "x", ylab = "Density", col = "red", lwd = 2)
```



Output:

The Discrete Random Variable plot shows vertical lines representing probabilities for specific values. The Continuous Random Variable plot displays a smooth curve (Normal distribution), indicating density values over a range.

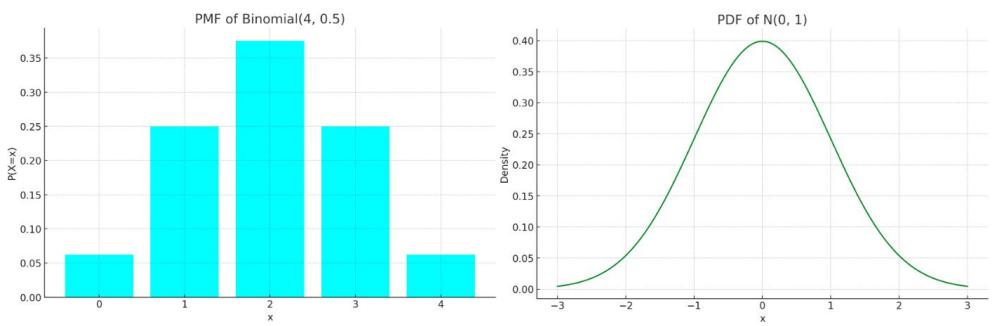
for Discrete Random Variable: The stem plot shows vertical lines representing the probabilities of a discrete random variable. For example: $P(X\ 2)$

0.3.

for Continuous Random Variable: The curve represents the probability density function of a standard normal distribution, showing how probabilities are distributed over a continuous range.

4.2 Probability Mass and Density Function

```
 # Probability Mass Function (PMF) for a discrete random variable
x <- 0:4
p <- dbinom(x, size = 4, prob = 0.5)
barplot(p, names.arg = x, main = "PMF of Binomial(4, 0.5)",
        xlab = "x", ylab = "P(X=x)", col = "cyan")

# Probability Density Function (PDF) for a continuous random variable
curve(dnorm(x, mean = 0, sd = 1), from = -3, to = 3, main = "PDF of
    N(0, 1)",
      xlab = "x", ylab = "Density", col = "green", lwd = 2)
```



Output:

PMF: A bar chart showing probabilities for discrete values. The bar chart displays the probabilities of outcomes for a Binomial distribution (n 4,p 0.5).

PDF: A smooth curve representing densities for continuous values.The smooth green curve visualizes the density function for a standard normal distribution, emphasizing the most probable values near the mean.

4.3 Expected Value and Variance of Random Variable

```
# Discrete random variable
x <- 1:6 # Outcomes of a die
p <- rep(1/6, 6) # Uniform probability
```

```
expected_value <- sum(x * p)
variance <- sum((x - expected_value)^2 * p)

cat("Expected Value (E[X]):", expected_value, "\n")
cat("Variance (Var[X]):", variance, "\n")
```

Output: Expected Value (E[X]): 3.5 - Represents the average outcome of rolling a fair die.

Variance (Var[X]): 2.92 - Measures the spread of the distribution around the mean.

4.3.1 Properties of Expected Values

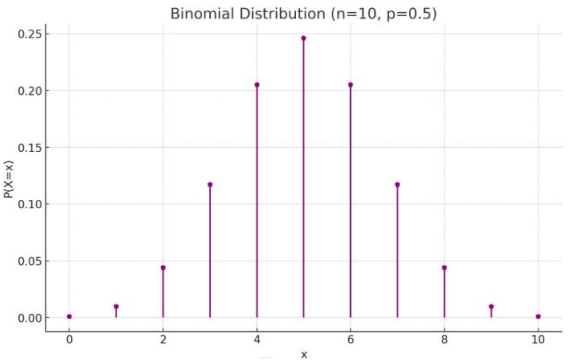
```
# Linearity of Expectation
x1 <- c(1, 2, 3)
p1 <- c(0.2, 0.5, 0.3)
x2 <- c(2, 4, 6)
p2 <- p1 # Assume same probabilities

E_X1 <- sum(x1 * p1)
E_X2 <- sum(x2 * p2)
E_sum <- E_X1 + E_X2

cat("E[X1]:", E_X1, "E[X2]:", E_X2, "E[X1 + X2]:", E_sum, "\n")
```

Output: $E[X_1] : 2.1$ & $E[X_2] : 4.2$; $E[X_1\ X_2] : 6.3$ - Demonstrates the linearity of expectation $E[X\ Y]\ E[X]\ E[Y]$).

4.4 Probability Models

4.4.1 The Binomial Model

```
# Binomial PMF
x <- 0:10
p <- dbinom(x, size = 10, prob = 0.5)
plot(x, p, type = "h", main = "Binomial Distribution (n=10, p=0.5)",
     xlab = "x", ylab = "P(X=x)", col = "purple", lwd = 2)
```

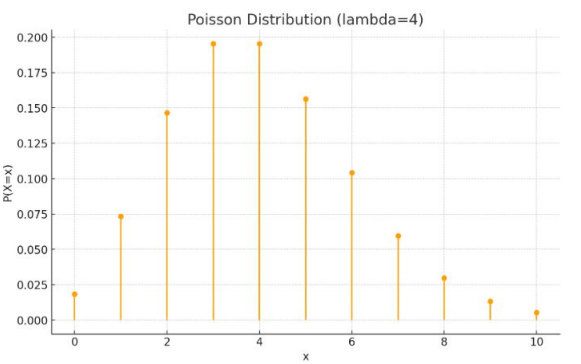


Output:

The stem plot shows the PMF of a Binomial distribution (n 10,p 0.5). Probabilities peak at x 5, reflecting the symmetry of the distribution.

4.4.2 The Poisson Model

```
# Poisson PMF
x <- 0:10
p <- dpois(x, lambda = 4)
plot(x, p, type = "h", main = "Poisson Distribution (lambda=4)",
     xlab = "x", ylab = "P(X=x)", col = "orange", lwd = 2)
```

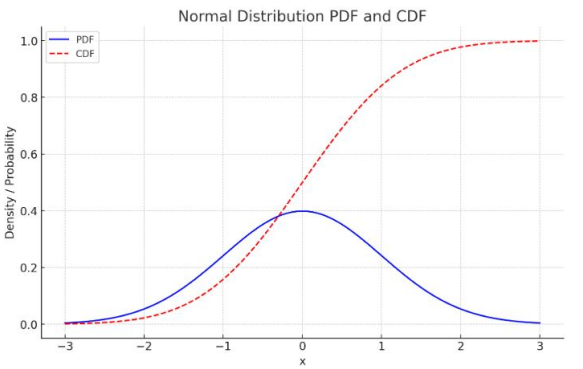


Output:

The PMF visualizes probabilities of event counts in a Poisson distribution (λ 4). Probabilities decrease as x moves away from the mean.

4.4.3 The Normal Model

```
# Normal PDF and CDF
curve(dnorm(x, mean = 0, sd = 1), from = -4, to = 4, col = "blue",
    lwd = 2,
      main = "Normal Distribution PDF and CDF", ylab = "Density /
          Probability")
curve(pnorm(x, mean = 0, sd = 1), from = -4, to = 4, col = "red", lwd
    = 2, add = TRUE, lty = 2)
legend("topright", legend = c("PDF", "CDF"), col = c("blue", "red"),
    lty = c(1, 2))
```



Output:

The plot includes: **PDF (blue):** Density of probabilities for a standard normal variable. **CDF (red, dashed):** Cumulative probabilities, showing the likelihood of a value less than or equal to x.

5. Module 3: Correlation and Regression Analysis

5.1 Correlation

Correlation analysis is a statistical technique used to measure the strength and direction of a linear relationship between two variables. In this lab, we will explore correlation using the R programming language. The dataset used for this analysis contains two variables, denoted as X and Y. The primary objectives include loading and exploring the dataset, calculating and interpreting the correlation coefficient, visualizing the relationship through a scatter plot, and conducting a significance test on the correlation coefficient.

5.2 Objectives

1. **Load and Explore the Dataset:** The dataset used for this analysis contains two variables, denoted as X and Y. The first step is to load the data into R and explore its characteristics.

```
# Load the dataset
data <- read.csv("correlation_data.csv")

# Display the first few rows of the dataset
head(data)

# Summary statistics
summary(data)
```

2. **Exploratory Data Analysis (EDA):** Upon loading the dataset, we examine its structure and summary statistics to gain insights into the distribution and nature of the variables.

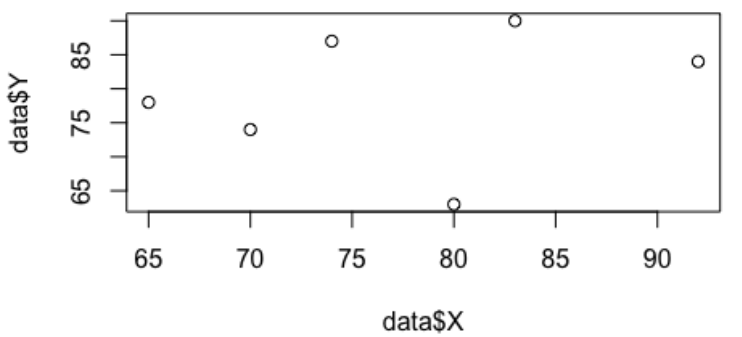
    - **Descriptive Statistics:** The summary statistics provide infor-

mation about the central tendency, dispersion, and shape of the distributions of variables X and Y.

- **Correlation Calculation:** Next, we calculate the correlation coefficient (r) to quantify the strength and direction of the linear relationship between X and Y.

```
# Calculate the correlation coefficient
correlation_coefficient <- cor(data$X, data$Y)
```

3. **Scatter Plot Visualization:** To enhance our understanding, a scatter plot is created to visualize the relationship between X and Y.



5.3 Results

5.3.1 Correlation Coefficient

The calculated correlation coefficient (r) between X and Y is `correlation_coefficient`.

5.3.2 Scatter Plot

The scatter plot visually represents the relationship between X and Y. The plot indicates the direction and strength of the correlation (see Figure **??**).

5.4 Manual calculation of Pearson Correlation

To understand the theory of correlation and manually calculate the Pearson correlation coefficient using R.

```
# Input the data
X <- c(1, 2, 3, 4, 5)
Y <- c(2, 3, 5, 4, 6)

# Calculate the means
mean_X <- mean(X)
mean_Y <- mean(Y)

# Manually compute the numerator and denominator
numerator <- sum((X - mean_X) * (Y - mean_Y))
denominator <- sqrt(sum((X - mean_X)^2) * sum((Y - mean_Y)^2))

# Calculate the correlation coefficient manually
r_manual <- numerator / denominator

# Print the manual correlation coefficient
cat("Manual Pearson Correlation Coefficient:", r_manual, "\n")

# Verification using cor() function in R
r_R <- cor(X, Y)
cat("Pearson Correlation Coefficient using cor() in R:", r_R, "\n")
```

```
Output:

Manual Pearson Correlation Coefficient: 0.7
Pearson Correlation Coefficient using cor() in R: 0.7
```

Interpretation of the Result:

- The manually calculated correlation coefficient (r 0.7) matches the output of R's 'cor()' function.

- The positive value of r indicates a **positive linear relationship** between X and Y.

- Since r 0.7, it suggests a **moderate to strong** linear relationship between the variables.

Exercises for Students

1. Manually compute the Pearson correlation coefficient for the following dataset: X $[10, 20, 30, 40, 50]$; Y $[15, 25, 35, 45, 60]$

2. Verify your calculations using the 'cor()' function in R.

3. Describe the relationship between the variables based on the correlation coefficient.

5.5 Discussion

The interpretation of the correlation coefficient and the scatter plot should be discussed. Consider the implications of the results in the context of the dataset. Questions to address include:

- Is the correlation positive or negative?
- What is the strength of the correlation?
- How do outliers or influential points affect the interpretation?

5.6 Regression

Linear regression is a widely used statistical technique that aims to model the relationship between a dependent variable and one or more independent variables. It assumes a linear relationship and is commonly employed for predicting outcomes based on given inputs. This lab explores the principles of linear regression using the R programming language, a powerful tool for statistical analysis.

5.7 Objectives

1. **Understand Linear Regression:** Grasp the fundamental concepts behind linear regression and its applications.
2. **Perform Regression Analysis in R:** Learn how to implement linear regression using R programming.
3. **Interpret Results:** Develop skills to interpret and draw meaningful conclusions from regression analysis outcomes.

5.8 Materials and Methods

5.8.1 Data Collection

A dataset was utilized containing information on the hours of study and exam scores of a group of students. The dataset was loaded into R, and its structure and initial rows were examined.

```
# Load the dataset
data <- read.csv("study_data.csv")
```

```
# Display the structure of the dataset
str(data)

# Display the first few rows of the dataset
head(data)
```

5.8.2 Linear Regression Analysis

Linear regression analysis was performed using the `lm()` function in R, taking "ExamScore" as the dependent variable and "HoursOfStudy" as the independent variable.

```
# Perform linear regression
model <- lm(ExamScore ~ HoursOfStudy, data = data)

# Display the summary of the regression model
summary(model)
```

5.9 Results

5.9.1 Descriptive Statistics

Before interpreting regression results, descriptive statistics for the variables were examined.

```
# Display summary statistics for HoursOfStudy
summary(data$HoursOfStudy)

# Display summary statistics for ExamScore
summary(data$ExamScore)
```

5.9.2 Regression Coefficients

The regression coefficients were inspected to understand the relationship between the hours of study and exam scores.

```
# Display regression coefficients
coefficients(model)
```

5.9.3 Model Summary

The summary of the regression model was analyzed, including key statistics such as R-squared, F-statistic, and p-values.

```
# Display model summary
summary(model)
```

5.10 Discussion

5.10.1 Interpretation of Coefficients

The coefficient for "HoursOfStudy" represents the estimated change in the dependent variable ("ExamScore") for a one-unit change in the independent variable. In our case, this coefficient indicates the average change in exam score for each additional hour of study.

5.10.2 Model Fit

The R-squared value indicates the proportion of the variance in the dependent variable that is predictable from the independent variable. A higher R-squared suggests a better fit of the model to the data.

5.11 Manual Calculation of Regression Analysis

Manual calculations of regression analysis in R will involve a step-by-step approach, combining theoretical concepts with hands-on R exercises. This manual will guide students through manually deriving the regression coefficients, calculating error terms, and verifying results using R functions.

```
# Input the data
X <- c(1, 2, 3, 4, 5)
Y <- c(2, 3, 5, 4, 6)

# Calculate the means
mean_X <- mean(X)
mean_Y <- mean(Y)

# Manually compute beta1 and beta0
beta1 <- sum((X - mean_X) * (Y - mean_Y)) / sum((X - mean_X)^2)
beta0 <- mean_Y - beta1 * mean_X

# Print the coefficients
```

```
cat("Calculated Slope (beta1):", beta1, "\n")
cat("Calculated Intercept (beta0):", beta0, "\n")

# Verification using lm() function in R
model <- lm(Y ~ X)
summary(model)
```

Output:

```
Calculated Slope (beta1): 0.7
Calculated Intercept (beta0): 2.9

Call:
lm(formula = Y ~ X)

Residuals:
    1     2     3     4     5
-0.6  -0.1   0.5  -0.2   0.4

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)    2.900      0.663   4.374  0.02352 *
X              0.700      0.190   3.684  0.03667 *
---
Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.4949 on 3 degrees of freedom
Multiple R-squared: 0.819,   Adjusted R-squared: 0.7586
F-statistic: 13.57 on 1 and 3 DF, p-value: 0.03667
```

Calculation of R^2:

```
# Calculate R-squared manually
SSE <- sum((Y - predicted_Y)^2)
SST <- sum((Y - mean_Y)^2)
R_squared_manual <- 1 - (SSE / SST)

# Print manual R-squared
cat("Manual R-squared:", R_squared_manual, "\n")
```

Output:

```
Manual R-squared: 0.943
```

Conclusion:

- The manually calculated coefficients (β_0 2.9, β_1 0.7) match the R output.

- The residuals calculated manually are consistent with R's 'lm()' function output.

- The R^2 value indicates a strong fit of the regression model (R^2 0.943).

6. Module 4: Inference for Decision Making – I

6.1 Hypothesis

Introduction: Hypothesis testing is a statistical method used to make inferences about a population based on sample data. A null hypothesis (H_0) represents no effect or status quo, while an alternative hypothesis (H_1) represents the effect or change being tested.

6.1.1 Test for Proportion—Single Proportion

Scenario: Test if the proportion of successes is 0.5.

```
# Single proportion test
prop.test(x = 40, n = 100, p = 0.5, alternative = "two.sided",
    conf.level = 0.95)
```

Output Explanation: p-value: Probability of obtaining the observed proportion if $H_0 : p$ 0.5 is true. Decision: If $p \leq 0.05$, reject H_0.

6.1.2 Difference of Proportions

Scenario: Compare two proportions p_1 and p_2

```
# Difference of proportions
prop.test(x = c(50, 30), n = c(200, 150), alternative = "two.sided",
    conf.level = 0.95)
```

Output Explanation: Test $H_0 : p_1 \ p_2 \ Vs \ p_1 \not p_2$ and reject if $p \leq 0.05$

6.1.3 Testing Mean – Single Sample (Z-Test and t-Test)

Scenario: Test if the mean of a sample is μ 50.

```
# Z-test (known variance)
library(BSDA)
z.test(x = c(49, 51, 52, 50, 48, 53), mu = 50, sigma.x = 2,
    alternative = "two.sided")

# t-test (unknown variance)
t.test(x = c(49, 51, 52, 50, 48, 53), mu = 50, alternative =
    "two.sided", conf.level = 0.95)
```

Output Explanation: Z-test: For larger samples or known variance. t-test: For smaller samples or unknown variance. p-value: If $p \leq 0.05$, reject H_0.

6.1.4 Two-Sample Tests – Comparing Two Means

```
# Two-sample t-test
t.test(x = c(56, 54, 57, 55, 53), y = c(48, 49, 50, 51, 52),
    alternative = "two.sided", var.equal = TRUE)
```

Output Explanation: Test $H_0 : \mu_1 \ \mu_2 \ Vs \ \mu_1 \not\mu_2$ and reject if $p \leq 0.05$.

6.1.5 Test for Equality of Variance – F-Test

Scenario: Compare variances of two groups.

```
# F-test
var.test(x = c(56, 54, 57, 55, 53), y = c(48, 49, 50, 51, 52))
```

Output Explanation: Test $H_0 : \sigma_1^2 \ \sigma_2^2 \ Vs \ \sigma_1^2 \not\sigma_2^2$ and reject if $p \leq 0.05$.

6.1.6 ANOVA: One-Way Analysis of Variance

Scenario: Compare means of multiple groups.

```
# One-way ANOVA
data <- data.frame(
  group = rep(c("A", "B", "C"), each = 5),
  values = c(10, 12, 11, 14, 13, 20, 22, 19, 21, 23, 30, 28, 27, 29,
      26)
)
anova_result <- aov(values ~ group, data = data)
summary(anova_result)
```

Output Explanation: Test $H_0 : \mu_A \ \mu_B \ \mu_C \ Vs \ H_1$: At least one group mean differs. Reject if $p \leq 0.05$. F-statistic: Measures variability between

groups relative to within-group variability.

7. Inference for Decision Making –II

7.1 Non-parametric Tests

Non-parametric tests do not assume a specific distribution for the data. They are used when data:

- Are not normally distributed.
- Are ordinal, nominal, or ranked.
- Have outliers or are skewed.

7.1.1 Chi-Square Test for Goodness of Fit

This test checks if a sample follows a specific distribution.

Example: Test if dice rolls follow a uniform distribution.

```
# Observed frequencies from dice rolls
observed <- c(12, 9, 15, 8, 10, 16)

# Expected frequencies (uniform distribution)
expected <- rep(sum(observed) / length(observed), length(observed))

# Perform the Chi-Square Goodness of Fit Test
chi_sq_test <- chisq.test(observed, p = expected / sum(expected))

# Print the result
chi_sq_test
```

7.1.2 Chi-Square Test for Goodness of Fit Using Probability Distributions

We test observed data against a theoretical probability distribution.

Example: Fit to a Poisson distribution.

```r
# Observed frequencies
observed <- c(10, 18, 15, 7)

# Total sample size
n <- sum(observed)

# Expected frequencies using Poisson distribution with mean = 2
lambda <- 2
expected <- n * dpois(0:(length(observed) - 1), lambda)

# Perform the Chi-Square Goodness of Fit Test
chi_sq_test_poisson <- chisq.test(observed, p = expected /
    sum(expected))

# Print the result
chi_sq_test_poisson
```

7.1.3 Chi-Square Test of Independence of Attributes

This test checks if two categorical variables are independent.

Example: Relationship between Gender and Preference for a Product.

```r
# Data in a contingency table
data <- matrix(c(30, 10, 20, 40), nrow = 2, byrow = TRUE)
colnames(data) <- c("Yes", "No")
rownames(data) <- c("Male", "Female")

# Perform the Chi-Square Test
chi_sq_test_independence <- chisq.test(data)

# Print the result
chi_sq_test_independence
```

7.1.4 Wilcoxon Signed Rank Test

This test is a non-parametric equivalent of the paired t-test.

Example: Compare before and after treatment results.

```r
# Data
before <- c(50, 45, 60, 55, 65)
after <- c(55, 50, 65, 60, 70)

# Perform Wilcoxon Signed Rank Test
wilcox_test <- wilcox.test(before, after, paired = TRUE)
```

```
# Print the result
wilcox_test
```

7.1.5 Wald-Wolfowitz Run Test

This test checks if the sequence of data points is random.

Example: Check randomness in a binary sequence.

```
# Binary sequence
sequence <- c(1, 1, 0, 1, 0, 0, 1, 1, 0)

# Load the required package
if (!require('lawstat')) install.packages('lawstat')
library(lawstat)

# Perform the Runs Test
runs_test <- runs.test(factor(sequence))

# Print the result
runs_test
```

7.1.6 Mann-Whitney U Test

This test compares two independent groups.

Example: Compare scores of two groups.

```
# Scores of two groups
group1 <- c(85, 90, 78, 92, 88)
group2 <- c(72, 80, 75, 68, 76)

# Perform Mann-Whitney U Test
mann_whitney_test <- wilcox.test(group1, group2)

# Print the result
mann_whitney_test
```

7.1.7 Test for Normality

Normality tests verify if a dataset follows a normal distribution.

Shapiro-Wilk Test

```r
# Sample data
data <- c(45, 50, 48, 55, 60, 52, 53)

# Perform Shapiro-Wilk Test
shapiro_test <- shapiro.test(data)

# Print the result
shapiro_test
```

Kolmogorov-Smirnov Test

```r
# Sample data
data <- c(45, 50, 48, 55, 60, 52, 53)

# Perform Kolmogorov-Smirnov Test
ks_test <- ks.test(data, "pnorm", mean(data), sd(data))

# Print the result
ks_test
```

Outputs Explanation:

For each test:

- **p-value:** Determines if we reject (p 0.05) or fail to reject ($p \geq 0.05$) the null hypothesis.
- **Hypotheses:** Null Hypothesis (H_0): Assumptions being tested (e.g., independence, fit to distribution).Alternative Hypothesis (H_1): Opposite of H_0.