R for Actuarial Statistics

Krishna Kumar Shrestha

2024-04-19

Table of contents

Preface

Welcome to *Foundations of Statistical Modeling and Analysis: A Computational Approach*. I'm excited to be your guide through the intricate world of statistical science and programming. My name is Krishna Kumar Shrestha, and as an educator deeply passionate about mathematics and its applications, I've witnessed firsthand the transformative power of statistical analysis in various domains, from finance to healthcare and beyond.

This course draws heavily from BMS102 Actuarial Statistics I and BMS152 Actuarial Statistics II and emphasizes the practical implementation of statistical concepts using the R programming language. This book focuses primarily on practical aspects rather than theoretical discussions. Through the R programming environment, you'll not only master the fundamental syntax and object types of R programming but also delve into probability distributions, data analysis, statistical inference, and regression theory.

Approach each chapter with curiosity and diligence, embracing the challenges and celebrating the breakthroughs. Remember, mastery is a journey, not a destination. With dedication and perseverance, you'll emerge equipped to tackle the complexities of statistical modeling with confidence and precision.

I extend my heartfelt gratitude to all the students, educators, and professionals who have contributed to this book's development, directly or indirectly. Your insights, feedback, and passion for learning have been invaluable.

In closing, I wish you a fulfilling and enriching learning experience. May this book serve as a guiding light on your path to mastering the computational aspects of statistical modeling and analysis.

Happy coding!

Krishna Kumar Shrestha

1 Introduction

1.1 Welcome to R programming

Hello, My name is Krishna Kumar Shrestha. I'm glad you're here! This book is a guide to learning R, a powerful computer language used for statistics and data analysis. It's written for students who are preparing for Actuarial Professional Exams that requires R programming but it's also great for anyone who wants to learn R from scratch and move to an intermediate level.

1.2 What's Inside This Book?

In this book, we will start with the basics of R and work our way up to more complex topics. I will explain things step-by-step, so you don't need to worry if you are new to programming or statistics. By the end of this book, you will know enough about R to do real-world data analysis, especially in the field of actuarial statistics. Along the way, I will give you exercises and examples to help you practice.

1.3 Why learn R?

R is a free and open-source language, which means anyone can use it and contribute to it. It is used by data scientists, statisticians and researchers all over the world. With R, you can do simple tasks like adding up numbers, as well as complex things like making charts and Running statistical tests. Its very useful tool, and once you learn it, you will have a skill that can help you in your research or in professional work

1.4 Let's Get started!

Now that you know a bit about R and what's in this book , let's get started! Follow the instructions in the next chapters to set up R and Rstudio, and then we will dive into the basics. I am excited to guide you through this journey. By the end , you will be able to use R for all sorts of data analysis, especially in the field of actuarial statistics. Let's go!

2 Setting Up Your R Environment

Before we can start coding in R, lets make sure everything is set up properly. In this chapter, we will install R and Rstudio, the tools you will use throughout this book. We will also explore the RStudio interface to understand where everything is and what each part does.

2.1 Installing R

To install R, go to the Comprehensive R Archive Network (CRAN) website and choose the version that matches your operating system(Windows, macOS or Linux). Follow the on-screen instructions to complete the installation. If you run into any issues, check the FAQs on the CRAN website or reach out to online R forums for help.

2.2 Installing Rstudio.

Once R is installed, lets install Rstudio. It's an integrated development environment(IDE) that makes working in with R easier. Visit the RStudio Website and download the desktop version for your operating system. After downloading, follow the installation steps to set up RStudio.

2.3 Exploring the RStudio Interface

When You first open Rstudio, it might look a bit overwhelming with multiple panes and buttons. Don't worry; we will break it down to understand what each part does and how to use it. Here's guide to the main components of the RStudio interface:

1. Console: This is where you can type and run R code directly. It's usually in the bottom-left corner of the screen. You can type commands here and press `Enter` to execute them.

2. Script Editor: This is where you write and save your R scripts (A script in a file containing a series of R commands). If you don't see the Script Editor when you open RStudio, you can create a new script by clicking "File" > "New File" > "R Script." It will usually appear in the top-left corner. You can write your code here and run it in the Console by pressing `Ctrl + Enter` (Windows) or `Cmd+Enter` (macOS).

3. Environment Pane: This pane shows you all the variables, data frames, and other objects you are working with. By default, it's located in the top-right corner. This pane is useful for keeping track of the data in your current R session.

4. Files Pane: this pane displays the files in your working directory(the folder where Rstudio looks for files). It's usually in the bottom-right corner. You can use it to navigate through your files and open scripts or data files.

5. Plots Pane: This is where you will see any plots or graphs you create. By default, it's also in the bottom-right corner, typically sharing space with the files Pane. If you create a plot in R, it will appear here.

6. Packages Pane: This pane shows you the R packages installed on your system and allows you to install or update packages. It is Generally located in the bottom-right corner, sharing space with the Files and Plots Panes.

7. Help Pane: This pane provides documentation and help for R commands and packages. You can usually find it in the bottom-right corner as well. When you use the `?` command or press `F1` on a function, the documentation will appear in this pane.

2.4 Customizing Your Work Space

RStudio allows you to customize your work space by resizing and moving panes. If you find that the default layout doesn't suit you, you can adjust it. To move a pane, click and drag the tab to a new location. To resize a pane, hover over the edge of a pane until you see a double-headed arrow, then click and drag to adjust the size.

8. If you want to reset the layout to the default settings, go to "Tools" > "Global Options" > "Pane Layout" and click "Reset to Default". This will return the interface to its original setup.

Exercise 2.1 What is the difference between R and RStudio? Describe the roles each plays in the process of writing and executing R code.

Exercise 2.2 Explain the differences between the console and script file in Rstudio. When would you use one over the other?

3 Working with Variables.

3.1 Introduction to Variables

In R, a Variable is a named storage location for data. You can think of a variable as a label that points to a value or set of values. Variables allow you to store results, reuse values, and perform operations without retying the same information repeatedly. To create a variable, you assign a value to a name using the assignment operator `<-`. For example:

```
x <- 42
y<- "hello world !"
```

In this example, `x` is a variable containing the numeric value 42 and `y` is a variable containing the character string "Hello,R!"

> You can use `class(x)` or `typeof(x)` to see its type

3.1.1 Best Practices when creating variable names

Choosing appropriate names for your variables is essential for code readability and maintenance. Here are some best practices when creating variable names in R:

- Meaningful Names: Use descriptive names that reflect the purpose of the variables . for example, `age`, `total_sales` , or `is_student`.
- Avoid Reserved Words: R has reserved words that are used in its syntax (like `if` , `c` , `for` etc.) Avoide using these as variable names, as it will cause errors.
- Use Underscores or CamelCase: To improve readability, use underscores (_) or camelCase (e.g., `myVariableName`). Avoid using spaces in variable names.

No Special Characters: Variable names should contain only letters, numbers, or underscores. Avoid using special characters like `@` , `#` , etc.

3.1.2 Case Sensitivity

R is case- sensitive, which means that variable names with different cases are considered separate. For example, `myVariable` and `MyVariable` are different variable. This case sensitivity allows flexibility, but it also requires careful naming to avoid errors due to mistyped variable names.

3.1.3 What's Allowed and What's Not Allowed in Variable Names

When creating variable names in R, consider the following rules to ensure that your variable names are valid:

- Allowed characters: Variable names can contain letters (a-z,A-Z), number (0,9), and underscores(_). The variable name must start with a letter or an underscore.
- Prohibited Characters: Do not use spaces, special characters, or operators. There will cause errors or lead to unexpected behavior.
- No reserved Words: Avoid using reserved words in R, such as `id`, `TRUE`, `FALSE` and others. You can find a full list of reserved words in the R documentation.

3.2 Creating Numeric Variables

Numeric variables are used to store numbers in R. They can be integers or real numbers (with decimal points), and they are essential for performing arithmetic operations, statistical analysis, and more. To create a numeric variable in R, you assign a number to a variable name using the assignment operator `<-`. Let's explore how to create a numeric variables.

```r
# Assigning an integer
age <- 25

# assigning a real number (with decimal point)
temperature <- 98.6
```

> `#` is a special character which is used for commenting in R. In this example, `age` is a numeric variable containing an integer, while `temprature` is a numeric variable with a real number (also known as a double or float).
>
> You can use `typeof()` function to find it.

3.3 Creating Character Variables

Similarly character variables are used to store textual data, which can include words, phrases, sentences, or any other combination of characters. To create a character variable, you assign a string of text to a variable name using the assignment operator `<-`. In R, text strings are enclosed in either (`'`) or double (`"`) quotation marks.

```r
# Assigning a character variable
name <- "John Doe"

# Assigning another character variable with single quotes

greeting <- 'hello ,world!' # alternatively you could have used double quotes
```

In this example , `name` and `greeting` are character variable containing strings of text.

3.4 Operators in R

R has a variety of operators, each designed for specific purposes. They can be broadly categorized into the following types:

- Assignment Operators
- Arithmetic Operators
- Comparison Operators
- Logical Operators
- Miscellaneous Operators

Let's take a closer look at each type.

3.4.1 Assignment Operators

Assignment operators are used to assign values to variables. The most common assignment operators in R is `<-` , but `=` can also be used.

```r
# using assignment operator
x <- 10 # assigns 1- to x
y=20 # assigns 20 to y
```

3.4.2 Arithmetic Operators

Arithmetic operators perform basic mathematical operators. Here's a list of common:

- Addition (+): Adds two or More numbers.
- Subtraction (-): Subtracts one value from another.
- Multiplication (*) : Multiplies two or more values
- Division (/): Divides one value by another.
- Exponentiation (^): Raises one value to the power of another.
- Modulus (%%): Returns the remainder when one value is divided by another.
- Integer Division(%/%): Returns the integer quotient when one value is divided another.

Here's an example demonstrating the use of these operators:

```
# Arithmetic operations
a <- 10
b <- 3

# Basic arithmetic
sum <- a + b           # 13
difference <- a - b    # 7
product <- a * b       # 30
quotient <- a / b      # 3.333...

# Exponentiation, Modulus, and Integer Division
power <- a ^ 2         # 100
modulus <- a %% b      # 1
integer_quotient <- a %/% b  # 3
```

3.4.3 Comparison Operators

Comparison operators are used to compare values and return a logical result (TRUE or FALSE). These operators are often used in conditional statements and for subsetting data. Here's a list of common comparison operators:

- **Equal to (==)**: Checks if two values are equal.
- **Not equal to (!=)**: Checks if two values are not equal.
- **Greater than (>)**: Checks if one value is greater than another.

- **Less than (`<`)**: Checks if one value is less than another.
- **Greater than or equal to (`>=`)**: Checks if one value is greater than or equal to another.
- **Less than or equal to (`<=`)**: Checks if one value is less than or equal to another.

```r
# Comparison operations
x <- 10
y <- 20

is_equal <- x == y   # FALSE
is_not_equal <- x != y  # TRUE
is_x_greater_than_y <- x > y  # FALSE
is_x_less_than_y <- x < y  # TRUE
```

3.4.4 Logical Operators

Logical operators are used to combine logical conditions or negate them. They are often used in conditional statements and for filtering data. Here's a list of common logical operators:

- **AND (`&`)**: Returns TRUE if both conditions are true.
- **OR (`|`)**: Returns TRUE if at least one condition is true.
- **NOT (`!`)**: Negates a condition, turning TRUE to FALSE or vice versa.

```r
# Logical operations
a <- TRUE
b <- FALSE

and_result <- a & b    # FALSE
or_result <- a | b     # TRUE
not_result <- !a       # FALSE
```

3.5 Assignment

1. Given a principal of $10,000, an annual interest rate of 5%, and a time period of 3 years, calculate the simple interest using the formula: `I = P * r * t`, where `P` is the principal, `r` is the interest rate, and `t` is the time in years. Write an R script to calculate this simple interest.

2. A principal of $5,000 is compounded annually at a rate of 4% for 5 years. Calculate the compound interest using the formula: `A = P * (1 + r/n)^(n * t)`, where `n` is the number of compounding periods per year. Write an R script to calculate the final accumulated amount after 5 years.
3. Given a future value of $20,000, a discount rate of 6%, and a time period of 4 years, calculate the present value using the formula: `PV = FV / (1 + r)^t`. Write an R script to find the present value with the given parameters.
4. Given an asset with an initial cost of $50,000 and a residual value of $10,000, use the straight-line method to calculate annual depreciation over 8 years. The formula is: `Depreciation = (Cost - Residual Value) / Useful Life`. Write an R script to calculate the annual depreciation.
5. A company has a revenue of $100,000 and total expenses of $70,000. Calculate the net profit margin using the formula: `Net Profit Margin = (Net Income / Revenue) * 100`, where `Net Income = Revenue - Total Expenses`. Write an R script to find the net profit margin.
6. Given a net income of $50,000 and total equity of $200,000, calculate the return on equity using the formula: `ROE = (Net Income / Equity) * 100`. Write an R script to compute this ratio.
7. A company has a net income of $60,000 and 30,000 shares outstanding. Calculate the earnings per share using the formula: `EPS = Net Income / Total Shares Outstanding`. Write an R script to find the EPS.
8. Given a revenue of $120,000, operating expenses of $60,000, and other costs of $10,000, calculate the operating profit margin using the formula: `Operating Profit Margin = (Operating Profit / Revenue) * 100`, where `Operating Profit = Revenue - Operating Expenses - Other Costs`. Write an R script to calculate this.
9. If total liabilities are $150,000 and total equity is $200,000, calculate the debt-to-equity ratio using the formula: `Debt-to-Equity = Total Liabilities / Total Equity`. Write an R script to compute this ratio.
10. Given a net income of $40,000 and total assets of $250,000, calculate the return on assets using the formula: `ROA = (Net Income / Total Assets) * 100`. Write an R script to compute this ratio.
11. If a company pays annual dividends of $2 per share and the stock price is $40, calculate the dividend yield using the formula: `Dividend Yield = (Dividend per Share / Stock Price) * 100`. Write an R script to calculate the dividend yield.
12. Given earnings before interest and taxes (EBIT) of $100,000 and annual interest expenses of $20,000, calculate the interest coverage ratio using the formula: `Interest Coverage Ratio = EBIT / Interest Expense`. Write an R script to calculate this ratio.
13. If the share price is $50 and earnings per share is $5, calculate the price-to-earnings ratio using the formula: `P/E Ratio = Share Price / EPS`. Write an R script to compute this ratio.
14. Given a beginning inventory of $30,000, purchases of $50,000, and an ending inventory of $20,000, calculate the cost of goods sold using the formula: `COGS = Beginning Inventory + Purchases - Ending Inventory`. Write an R script to calculate COGS.

15. If current assets are $100,000 and current liabilities are $60,000, calculate the working capital using the formula: `Working Capital = Current Assets - Current Liabilities`. Write an R script to compute this.
16. Given a gross profit of $70,000 and revenue of $120,000, calculate the gross profit margin using the formula: `Gross Profit Margin = (Gross Profit / Revenue) * 100`. Write an R script to find this ratio.
17. If a company invests $30,000 in new equipment, record this as a capital expenditure. Write an R script to assign this capital expenditure to a variable.

4 Basic Data Types and Data Structures in R

4.0.1 Introduction to Data type

Data types are foundational in any programming language, and R is no exception. Understanding data types allows you to work with data efficiently, ensuring that your code behaves as expected. In R, data types refer to the kind of data a variable or object holds. Let's explore the common data types in R and why it's essential to know about them.

4.0.1.1 Why Data Types Matter

Data types define the operations and functions you can perform on a given object. For example, you can perform arithmetic on numeric types but not on character types. Knowing the data type of a variable helps you avoid errors and choose appropriate methods to manipulate data.

4.0.1.2 Common Data Types in R

R has several fundamental data types that you will encounter frequently:

- **Numeric**: This data type represents numbers, including integers and floating-point numbers. Examples include 42, -3.14, and 1000.5.
- **Character**: Character data types represent text or string data. They are enclosed in quotes, such as `"Hello, world!"` or `'R programming'`.
- **Logical**: The logical data type has only two possible values: `TRUE` or `FALSE`. It is often used in conditionals and logical operations.
- **Factor**: Factors represent categorical data, which can have a fixed number of unique values (called levels). Factors are useful for statistical analysis and plotting.
- **Date and Date-Time**: R has specific data types for representing dates and date-time values. The `Date` type stores calendar dates, while `POSIXct` and `POSIXlt` represent date-time values with timestamps.

4.0.2 Working with Data Types

To understand the data type of a variable in R, you can use functions like **`class()`** and **`typeof()`**. These functions help you identify what type of data you're working with, allowing you to make appropriate operations and conversions.

- **Class and Typeof**: The **`class()`** function reveals the class of an object, while **`typeof()`** shows the internal storage mode of the data. These functions are helpful when debugging or when you need to determine a variable's type.

4.0.3 Type Conversion

In R, you often work with multiple data types, sometimes needing to convert one type into another. This process, called type conversion, allows you to perform operations that are specific to certain data types, like arithmetic operations on numeric types or string manipulations on character types.

4.0.3.1 Implicit Type Conversion

R has a built-in mechanism for converting data types implicitly, especially when mixing data types in operations. For example, if you add a number to a character, R converts the character to a numeric value (if possible) before performing the operation. This can be helpful but sometimes leads to unexpected behavior if you don't expect it.

```
#result <- 5 + "10"  # "10" is converted to numeric, and the result is 15
```

However, implicit conversion can raise errors if the conversion isn't possible.

4.0.3.2 Explicit Type Conversion

Explicit type conversion is when you manually convert a variable from one type to another. This approach provides more control and reduces the risk of unintended conversions. R offers several functions for explicit type conversion:

- **Converting to Numeric**: To convert a character or logical variable to numeric, you can use the **`as.numeric()`** function. This is useful when you need to perform arithmetic on a variable.

  ```
  char_num <- "42"
  converted_num <- as.numeric(char_num)  # Converts "42" to numeric 42
  ```

- **Converting to Character**: To convert numeric or logical variables to character, use the **`as.character()`** function. This is helpful when creating labels or generating text output.

  ```
  num_var <- 123
  char_var <- as.character(num_var)  # Converts 123 to "123"
  ```

- **Converting to Logical**: The **`as.logical()`** function converts numeric or character variables to logical values. Numeric values other than zero are **`TRUE`**, while zero is **`FALSE`**. Character strings "TRUE" and "FALSE" convert to their corresponding logical values.

  ```
  num_var <- 1
  logical_var <- as.logical(num_var)  # Converts 1 to TRUE

  char_var <- "TRUE"
  logical_var <- as.logical(char_var)  # Converts "TRUE" to TRUE
  ```

- **Converting to Factor**: To convert character variables into factors, use the **`as.factor()`** function. Factors are useful when working with categorical data.

  ```
  char_var <- "apple"
  factor_var <- as.factor(char_var)  # Converts "apple" to a factor with one level: "ap
  ```

- **Converting to Date**: R offers functions to convert character data into dates. The **`as.Date()`** function is used for date-only data, while **`as.POSIXct()`** and **`as.POSIXlt()`** are used for date-time data.

  ```
  date_str <- "2024-04-20"
  date_var <- as.Date(date_str)  # Converts "2024-04-20" to a date object
  ```

4.0.4 Introduction to Data Structures

Data structures are the building blocks for organizing, storing, and manipulating data in R. Understanding the variety of data structures available in R is essential for effective data analysis and programming. In this section, we will explore the most common data structures in R and their key characteristics.

4.0.4.1 Why Data Structures Matter

Data structures determine how you can access, manipulate, and store data. The choice of data structure can affect performance, flexibility, and ease of use. Knowing when to use a specific data structure helps you write efficient and maintainable R code.

4.0.4.2 Common Data Structures in R

R offers a range of data structures to suit different types of data and analysis requirements. Here are some of the most commonly used ones:

- **Vectors**: Vectors are the most basic data structure in R, representing a one-dimensional array of elements. Vectors can contain numeric, character, or logical values. They are the building blocks for more complex structures.
- **Matrices**: Matrices are two-dimensional arrays with rows and columns. They are useful for mathematical operations and can contain numeric or character data.
- **Data Frames**: Data frames are tabular data structures, resembling spreadsheets or SQL tables. They consist of rows and columns, with each column potentially having a different data type. Data frames are commonly used in data analysis and statistical modeling.
- **Lists**: Lists are versatile data structures that can hold a collection of different data types, including vectors, matrices, data frames, and even other lists. Lists are useful for storing complex or nested data.
- **Factors**: Factors represent categorical data with a fixed number of levels. They are useful for statistical analysis and creating plots with specific categories.

4.0.4.3 When to Use Different Data Structures

Choosing the right data structure depends on your data and what you need to do with it. Here are some guidelines for selecting the appropriate data structure:

- Use **vectors** when you need a simple one-dimensional array for calculations or operations.
- Use **matrices** for two-dimensional data that require matrix-based operations.
- Use **data frames** for tabular data where you need to manipulate columns or work with data sets.
- Use **lists** when you need a flexible structure that can hold mixed data types or complex/nested data.
- Use **factors** for categorical data where specific levels are important.

4.0.5 Vectors

4.0.5.1 Creating Vectors

Vectors are versatile and can hold numeric, character, or logical data. You can create vectors in several ways:

- **Using `c()`**: The `c()` function (short for "combine" or "concatenate") is the most common method to create vectors. It allows you to combine individual elements or other vectors into a single vector.

  ```
  numeric_vector <- c(1, 2, 3, 4, 5)
  char_vector <- c("apple", "banana", "cherry")
  logical_vector <- c(TRUE, FALSE, TRUE)
  ```

This method is suitable for creating vectors from known values or combining existing vectors.

- **Using `seq()`**: The `seq()` function creates a sequence of numbers. It is ideal when you need vectors with a specific range, interval, or length.

  ```
  sequence_vector <- seq(1, 10, by = 2)  # Creates a vector [1, 3, 5, 7, 9]
  ```

Use this method when you need vectors with regular sequences or patterns.

- **Using `rep()`**: The `rep()` function creates a vector by repeating a value or pattern a specified number of times.

  ```
  repeated_vector <- rep(3, 5)  # Creates a vector [3, 3, 3, 3, 3]
  ```

This method is useful when you need vectors with repeated values.

- **Using `:` Operator**: The colon operator creates a sequence of numbers from a start to an end value.

  ```
  colon_vector <- 1:5  # Creates a vector [1, 2, 3, 4, 5]
  ```

Use this method for simple sequences with consecutive numbers.

4.0.5.2 Operations with Vectors

Vectors support a wide range of operations, allowing you to manipulate and transform data efficiently. Here are some common operations:

- **Arithmetic Operations**: You can perform arithmetic operations on numeric vectors, such as addition, subtraction, multiplication, and division.

```r
vector1 <- c(1, 2, 3)
vector2 <- c(4, 5, 6)

# Element-wise operations
sum_vector <- vector1 + vector2  # [5, 7, 9]
```

- **Logical Operations**: Logical vectors can be used for comparison and conditional operations.

  ```r
  vector1 <- c(1, 2, 3)
  vector2 <- c(3, 2, 1)

  # Logical comparisons
  equal_vector <- vector1 == vector2  # [FALSE, TRUE, FALSE]
  ```

- **Indexing and Slicing**: You can access specific elements in a vector using indexing. This allows you to extract subsets of a vector.

  ```r
  vector <- c(10, 20, 30, 40, 50)

  # Indexing to get the second element
  second_element <- vector[2]  # 20

  # Slicing to get a subset of elements
  subset_vector <- vector[1:3]  # [10, 20, 30]
  ```

- **Combining Vectors**: Vectors can be combined using **`c()`** or concatenated using **`append()`**.

  ```r
  vector1 <- c(1, 2, 3)
  vector2 <- c(4, 5, 6)

  combined_vector <- c(vector1, vector2)  # [1, 2, 3, 4, 5, 6]
  ```

4.0.5.3 Operations with Vectors of Different Lengths

4.0.5.4 Recycling in R

When you add, subtract, or perform other operations on vectors of unequal lengths, R uses a process called "recycling." Recycling involves reusing elements from the shorter vector to match the length of the longer vector. This behavior is useful but can lead to unintentional errors if not managed properly.

- **Adding Vectors of Different Lengths**: If you add two vectors of unequal lengths, R recycles the shorter vector to match the length of the longer one.

```r
vector1 <- c(1, 2, 3)
vector2 <- c(10, 20)

# Recycling happens here
result <- vector1 + vector2  # [11, 22, 13]
```

```
Warning in vector1 + vector2: longer object length is not a multiple of shorter
object length
```

In the above example, `vector2` is shorter, so its elements are recycled to match the length of `vector1`. This leads to `10` and `20` being reused, resulting in `[11, 22, 13]`.

- **Subtraction with Vectors of Different Lengths**: Recycling also applies when subtracting vectors of unequal lengths.

```r
vector1 <- c(100, 200, 300)
vector2 <- c(10, 20)

# Recycling results in
result <- vector1 - vector2  # [90, 180, 290]
```

```
Warning in vector1 - vector2: longer object length is not a multiple of shorter
object length
```

Similarly, `vector2` is recycled to match the length of `vector1`, leading to `10` and `20` being reused.

4.0.5.5 Avoiding Recycling Errors

While recycling can be useful, it's essential to be aware of potential errors when vectors have lengths that don't divide evenly. If the shorter vector's length doesn't align with the longer vector, R will issue a warning, indicating the recycling pattern might be incorrect.

```r
vector1 <- c(1, 2, 3)
vector2 <- c(10, 20, 30, 40)

result <- vector1 + vector2  # [11, 22, 33, 41]
```

```
Warning in vector1 + vector2: longer object length is not a multiple of shorter
object length
```

```r
# Warning message: longer object length is not a multiple of shorter object length
```

In this case, the warning indicates that the recycling might lead to unexpected results. To avoid such issues, ensure vectors have lengths that divide evenly or explicitly align their lengths.

4.0.5.6 Filtering Vectors

Filtering vectors is a common operation where you select elements based on specific conditions or criteria. This is useful when you need to extract a subset of data from a larger vector.

- **Logical Filtering**: You can create logical vectors with `TRUE` or `FALSE` values to select elements from another vector.

```r
numbers <- c(1, 2, 3, 4, 5, 6)

# Create a logical vector to filter even numbers
is_even <- numbers %% 2 == 0

# Use the logical vector to filter
even_numbers <- numbers[is_even]  # [2, 4, 6]
```

In this example, `is_even` is a logical vector indicating which elements are even numbers. By using this logical vector to index `numbers`, you can extract the even numbers.

- **Subsetting with Conditions**: You can also filter vectors based on conditions.

```r
ages <- c(15, 25, 35, 45, 55)

# Select ages greater than or equal to 35
adults <- ages[ages >= 35]  # [35, 45, 55]
```

Here, the condition **`ages >= 35`** creates a logical vector used to filter the **`ages`** vector, resulting in a subset of adults.

4.0.6 Matrices

4.0.6.1 Introduction to Matrices

A matrix is a two-dimensional data structure with rows and columns. Matrices can contain numeric, character, or logical data. Most commonly, they are used for numeric computations, linear algebra, and data analysis.

4.0.6.2 Creating Matrices

To create a matrix, you must specify the data, the number of rows, and the number of columns. The total number of elements must be a multiple of the product of rows and columns; otherwise, R may throw an error or recycle elements unexpectedly.

4.0.6.2.1 Using `matrix()`

The **`matrix()`** function creates a matrix from a vector, given a specified number of rows and columns. It's important to ensure that the total number of elements is appropriate for the desired matrix structure.

```r
# Creating a 2x3 matrix with dimension names
data <- 1:6
mat <- matrix(data, nrow = 2, ncol = 3)

# Naming the rows and columns
rownames(mat) <- c("Row1", "Row2")
colnames(mat) <- c("Col1", "Col2", "Col3")

# The matrix with dimension names:
#     Col1 Col2 Col3
# Row1    1    3    5
# Row2    2    4    6
```

This example creates a 2x3 matrix and names the rows and columns. Dimension names are helpful for data referencing and indexing.

4.0.6.2.2 Character Matrix

Character matrices are created similarly to numeric matrices, but they contain text data.

```r
# Creating a 2x3 character matrix
char_data <- c("A", "B", "C", "D", "E", "F")
char_mat <- matrix(char_data, nrow = 2, ncol = 3)

# Naming the rows and columns
rownames(char_mat) <- c("Row1", "Row2")
colnames(char_mat) <- c("Col1", "Col2", "Col3")

# The character matrix with dimension names:
#      Col1 Col2 Col3
# Row1    A    C    E
# Row2    B    D    F
```

If the number of elements doesn't align with the specified rows and columns, R will issue a warning or error. Here's an example of incorrect matrix creation:

```r
# Attempting to create a 2x3 matrix with only 5 elements
mat <- matrix(1:5, nrow = 2, ncol = 3)
```

```
Warning in matrix(1:5, nrow = 2, ncol = 3): data length [5] is not a
sub-multiple or multiple of the number of rows [2]
```

```r
# This leads to a warning:
# Warning message: data length [5] is not a sub-multiple or multiple of the number of rows
```

In this case, R attempts to recycle the elements to fill the matrix, but the recycling is not a proper multiple, leading to a warning. The recycling can result in unexpected behavior if the data structure isn't consistent.

4.0.6.2.3 Using `cbind()` and `rbind()`

These functions combine existing vectors into a matrix. Again, it's crucial to ensure the number of elements aligns with the desired matrix dimensions.

```r
# Correctly combining two 2-element vectors into a 2x2 matrix
col1 <- c(1, 2)
col2 <- c(3, 4)
mat <- cbind(col1, col2)  # [1, 3] and [2, 4]

# Correctly combining two 3-element vectors into a 2x3 matrix
row1 <- c(1, 2, 3)
row2 <- c(4, 5, 6)
mat <- rbind(row1, row2)  # [1, 2, 3] and [4, 5, 6]
```

However, if the combined vectors don't match the required dimensions, recycling or errors may occur.

```r
# Incorrectly combining two vectors of unequal lengths into a matrix
col1 <- c(1, 2)
col2 <- c(3, 4, 5)
mat <- cbind(col1, col2)  # Warning due to different lengths
```

```
Warning in cbind(col1, col2): number of rows of result is not a multiple of
vector length (arg 1)
```

In this case, the second vector has more elements than the first, leading to recycling or a warning.

4.0.6.3 Matrix Operation

4.0.6.3.1 Addition and Subtraction

Matrix addition and subtraction are element-wise operations, meaning corresponding elements are added or subtracted.

```r
m1 <- matrix(1:4, nrow = 2, ncol = 2)
m2 <- matrix(5:8, nrow = 2, ncol = 2)

# Addition
m_add <- m1 + m2

# Subtraction
m_sub <- m1 - m2

print(m_add)
```

```
     [,1] [,2]
[1,]    6   10
[2,]    8   12
```

```r
print(m_sub)
```

```
     [,1] [,2]
[1,]   -4   -4
[2,]   -4   -4
```

4.0.6.3.2 Scalar Multiplication

Scalar multiplication multiplies each element in a matrix by a scalar (a single number).

```r
scalar <- 3
m_scalar_mult <- scalar * m1
print(m_scalar_mult)
```

```
     [,1] [,2]
[1,]    3    9
[2,]    6   12
```

4.0.6.3.3 Matrix Multiplication

Matrix multiplication requires that the number of columns in the first matrix equals the number of rows in the second matrix.

```r
m3 <- matrix(c(1, 2, 3, 4, 5, 6), nrow = 2, ncol = 3)
m4 <- matrix(c(7, 8, 9, 10, 11, 12), nrow = 3, ncol = 2)

m_mult <- m3 %*% m4
print(m_mult)
```

```
     [,1] [,2]
[1,]   76  103
[2,]  100  136
```

4.0.6.3.4 Transposition

Transposing a matrix flips rows and columns.

```
m_transpose <- t(m1)
print(m_transpose)
```

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

4.0.6.3.5 Matrix Inversion

The inverse of a matrix `A` is a matrix `B` such that `A %*% B = I`, where `I` is the identity matrix.

```
m_invertible <- matrix(c(4, 7, 2, 6), nrow = 2, ncol = 2)
m_inverse <- solve(m_invertible)
print(m_inverse)
```

```
     [,1] [,2]
[1,]  0.6 -0.2
[2,] -0.7  0.4
```

4.0.6.3.6 Determinant

The determinant is a scalar value that can be computed from a square matrix.

```
det_val <- det(m_invertible)
print(det_val)
```

```
[1] 10
```

4.0.6.3.7 Eigenvalues and Eigenvectors

Eigenvalues and eigenvectors are used in many applications, including principal component analysis.

```
eigen_result <- eigen(m_invertible)
print(eigen_result$values)  # Eigenvalues
```

```
[1] 8.872983 1.127017
```

```r
print(eigen_result$vectors)  # Eigenvectors
```

```
           [,1]       [,2]
[1,] -0.3796908 -0.5713345
[2,] -0.9251135  0.8207173
```

4.0.7 Data Frames

Data frames are a key structure in R, designed to hold tabular data. Each column in a data frame can contain different data types, such as numeric, character, or factor. This chapter explores various ways to create data frames, common operations, and a few advanced techniques.

Data frames are similar to tables in relational databases or Excel spreadsheets. Each row represents an observation, and each column represents a variable.

4.0.8 Creating a Data Frame from Vectors

One of the most straightforward ways to create a data frame is from vectors. You can use the **`data.frame()`** function to combine multiple vectors into a data frame.

```r
# Create individual vectors
names <- c("Alice", "Bob", "Charlie")
ages <- c(25, 30, 35)
salaries <- c(50000, 60000, 70000)

# Create a data frame
df <- data.frame(Name = names, Age = ages, Salary = salaries)
print(df)
```

```
     Name Age Salary
1   Alice  25  50000
2     Bob  30  60000
3 Charlie  35  70000
```

4.0.8.1 Basic Data Frame Operations

Data frames support common operations like adding rows and columns, selecting subsets of data, and modifying values.

4.0.9 Adding Columns

You can add columns to an existing data frame by assigning a new variable to it.

```
df$Location <- c("New York", "Los Angeles", "Chicago")
print(df)
```

```
     Name Age Salary    Location
1   Alice  25  50000    New York
2     Bob  30  60000 Los Angeles
3 Charlie  35  70000     Chicago
```

4.0.10 Adding Rows

To add rows, you can use **`rbind()`**, which concatenates data frames by rows.

```
new_row <- data.frame(
  Name = "David",
  Age = 40,
  Salary = 80000,
  Location = "San Francisco"
)

df <- rbind(df, new_row)
print(df)
```

```
     Name Age Salary      Location
1   Alice  25  50000      New York
2     Bob  30  60000   Los Angeles
3 Charlie  35  70000       Chicago
4   David  40  80000 San Francisco
```

4.0.11 Selecting Data

You can select subsets of data using indexing, conditions, or specific functions like **`subset()`**.

```
# Select a single column
print(df$Name)
```

```
[1] "Alice"   "Bob"     "Charlie" "David"
```

```
# Select multiple columns
print(df[, c("Name", "Age")])
```

```
     Name Age
1   Alice  25
2     Bob  30
3 Charlie  35
4   David  40
```

```
# Select rows based on conditions
df_older_than_30 <- subset(df, Age > 30)
print(df_older_than_30)
```

```
     Name Age Salary      Location
3 Charlie  35  70000       Chicago
4   David  40  80000 San Francisco
```

4.0.12 Modifying Values

To update values in a data frame, you can use indexing and assignment.

```
# Update a specific value
df$Salary[df$Name == "Alice"] <- 55000
print(df)
```

```
     Name Age Salary      Location
1   Alice  25  55000      New York
2     Bob  30  60000   Los Angeles
3 Charlie  35  70000       Chicago
4   David  40  80000 San Francisco
```

```
# Update multiple values with conditions
df$Location[df$Department == "HR"] <- "Boston"
print(df)
```

```
     Name Age Salary      Location
1   Alice  25  55000      New York
2     Bob  30  60000   Los Angeles
3 Charlie  35  70000       Chicago
4   David  40  80000 San Francisco
```

4.0.13 Advanced Data Frame Operations

4.0.14 Merging Data Frames

To merge or join data frames, you can use the **`merge()`** function, specifying the common columns for joining.

```r
df2 <- data.frame(
  Name = c("Alice", "Charlie", "David"),
  Project = c("Alpha", "Beta", "Gamma")
)

# Merge data frames on the 'Name' column
df_merged <- merge(df, df2, by = "Name")
print(df_merged)
```

```
     Name Age Salary      Location Project
1   Alice  25  55000      New York   Alpha
2 Charlie  35  70000       Chicago    Beta
3   David  40  80000 San Francisco   Gamma
```

4.0.15 Applying Functions to Data Frames

Functions can be applied to rows or columns of a data frame using **`apply()`**, **`lapply()`**, or **`sapply()`**.

```r
# Find the mean salary
mean_salary <- mean(df$Salary)
print(mean_salary)
```

```
[1] 66250
```

```r
# Apply a function to each row
row_summary <- apply(df[, -1], 1, function(x) sum(as.numeric(x[2:3])))  # Exclude Name
```

```
Warning in FUN(newX[, i], ...): NAs introduced by coercion

Warning in FUN(newX[, i], ...): NAs introduced by coercion

Warning in FUN(newX[, i], ...): NAs introduced by coercion

Warning in FUN(newX[, i], ...): NAs introduced by coercion
```

```r
print(row_summary)
```

```
[1] NA NA NA NA
```

4.0.16 Lists in R

Lists in R are one of the most flexible data structures, allowing you to store different types of objects within a single list. In this chapter, we'll cover the basics of lists, ways to create them, and operations to access, modify, and manipulate their elements.

A list in R is a collection of objects that can be of varying types and lengths. You can store vectors, data frames, matrices, functions, or even other lists within a list.

4.0.17 Creating a List

To create a list in R, you use the **`list()`** function, passing in the objects you want to include in the list.

```r
# Create a simple list with different types of objects
my_list <- list(
  numbers = c(1, 2, 3, 4, 5),
  letters = c("a", "b", "c"),
  matrix = matrix(1:9, nrow = 3),
  df = data.frame(Name = c("Alice", "Bob"), Age = c(25, 30))
)

print(my_list)
```

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

$letters
[1] "a" "b" "c"

$matrix
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

$df
   Name Age
1 Alice  25
2   Bob  30
```

In this example, **`my_list`** contains a numeric vector, a character vector, a matrix, and a data frame.

4.0.18 Named Lists

You can create lists with named elements for easier access.

```r
# Create a named list
named_list <- list(
  Name = "John",
  Age = 40,
  Salary = 60000
)

print(named_list)
```

```
$Name
[1] "John"

$Age
[1] 40

$Salary
[1] 60000
```

4.0.19 Accessing List Elements

You can access elements in a list by index or by name. R allows for flexible methods to extract data from lists.

4.0.20 Index-based Access

Use double square brackets `[[ ]]` to access elements by index. Single square brackets `[ ]` return a subset of the list as another list.

```r
# Access the first element (numeric vector)
first_element <- my_list[[1]]
print(first_element)
```

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

```r
# Access the first two elements as a sublist
subset_list <- my_list[1:2]
print(subset_list)
```

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

$letters
[1] "a" "b" "c"
```

4.0.21 Name-based Access

When lists have named elements, you can access them using the element's name.

```r
# Access the 'letters' element
letters_element <- my_list$letters
print(letters_element)
```

```
[1] "a" "b" "c"
```

```r
# Alternatively, using double square brackets with the name
letters_element2 <- my_list[["letters"]]
print(letters_element2)
```

```
[1] "a" "b" "c"
```

4.0.22 Modifying List Elements

Lists in R are mutable, meaning you can change their elements, add new ones, or remove them.

4.0.23 Changing Existing Elements

You can replace existing elements by assigning new values to them.

```r
# Change the first element to a new vector
my_list[[1]] <- c(10, 20, 30)
print(my_list)
```

```
$numbers
[1] 10 20 30

$letters
[1] "a" "b" "c"

$matrix
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

$df
   Name Age
1 Alice  25
2   Bob  30
```

```r
# Change a named element
my_list$letters <- c("x", "y", "z")
print(my_list)
```

```
$numbers
[1] 10 20 30

$letters
[1] "x" "y" "z"

$matrix
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

$df
   Name Age
1 Alice  25
2   Bob  30
```

4.0.24 Adding New Elements

You can add new elements to a list by assigning them with a new index or name.

```r
# Add a new element with a named index
my_list$new_element <- "Hello, R!"
print(my_list)
```

```
$numbers
[1] 10 20 30

$letters
[1] "x" "y" "z"

$matrix
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

$df
   Name Age
1 Alice  25
2   Bob  30
```

```
$new_element
[1] "Hello, R!"
```

```
# Add a new element with a numerical index
my_list[[6]] <- c("extra", "items")
print(my_list)
```

```
$numbers
[1] 10 20 30

$letters
[1] "x" "y" "z"

$matrix
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

$df
   Name Age
1 Alice  25
2   Bob  30

$new_element
[1] "Hello, R!"

[[6]]
[1] "extra" "items"
```

4.0.25 Removing Elements

To remove elements from a list, you can use **NULL** or the **-** operator with the index.

```
# Remove the third element
my_list[[3]] <- NULL
print(my_list)
```

```
$numbers
```

```
[1] 10 20 30

$letters
[1] "x" "y" "z"

$df
   Name Age
1 Alice  25
2   Bob  30

$new_element
[1] "Hello, R!"

[[5]]
[1] "extra" "items"
```

```r
# Remove the first element
my_list <- my_list[-1]
print(my_list)
```

```
$letters
[1] "x" "y" "z"

$df
   Name Age
1 Alice  25
2   Bob  30

$new_element
[1] "Hello, R!"

[[4]]
[1] "extra" "items"
```

5 Control Structures in R: Conditionals and Loops

Control structures in R allow you to manipulate the flow of your programs, making them more flexible and dynamic. In this chapter, we'll focus on conditionals and loops, showing you how to implement decision-making and repetition in your R scripts.

5.1 Conditionals

Conditionals in R let you execute different blocks of code based on specific conditions. The most common conditional structure is the "if-else" statement, but there are other variations as well.

5.1.1 If-Else Statements

An "if-else" statement allows you to execute different code blocks depending on whether a condition is **`TRUE`** or **`FALSE`**. The general syntax is as follows:

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

```
NULL
```

Let's consider a simple example to illustrate the concept:

```r
x <- 10

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

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

In this example, the condition `x > 5` is evaluated. Since it's `TRUE`, the first block of code is executed, printing "x is greater than 5."

5.1.2 Else IF Statements

You can use "else if" statements to check multiple conditions in sequence. This is useful when there are more than two possible outcomes.

```
score <- 85

if (score >= 90) {
  print("Grade: A")
} else if (score >= 80) {
  print("Grade: B")
} else {
  print("Grade: C")
}
```

```
[1] "Grade: B"
```

In this example, the "else if" statement allows us to assign grades based on a range of scores. If the first condition isn't met, the next one is checked, and so on, until a condition is `TRUE` or the "else" block is executed.

5.1.3 Nested IF statements

You can also nest "if" statements within other "if" statements to create more complex decision-making logic.

```
age <- 18
has_id <- TRUE

if (age >= 18) {
  if (has_id) {
    print("Allowed to enter")
  } else {
    print("ID required")
  }
} else {
  print("Not allowed to enter")
}
```

```
[1] "Allowed to enter"
```

In this example, the outer "if" checks if the person is 18 or older. If true, the nested "if" checks if they have a valid ID. This kind of nesting is useful when you need to apply additional conditions within a broader context.

5.1.4 Switch Statements

A "switch" statement allows you to execute different code blocks based on the value of a variable. This is similar to "if-else" but often more readable when dealing with multiple cases.

```
day <- "Tuesday"

switch(day,
  "Monday" = print("Start of the workweek"),
  "Friday" = print("End of the workweek"),
  "Saturday" = print("Weekend!"),
  "Sunday" = print("Weekend!"),
  print("A regular weekday")
)
```

```
[1] "A regular weekday"
```

In this example, the **`switch()`** function selects a code block to execute based on the value of **`day`**. If the value doesn't match any specified cases, the default code block is executed (the last statement without a named case).

5.2 Loops

Loops are a fundamental concept in programming, allowing you to repeat code blocks multiple times. In R, loops come in various forms, including "for," "while," and "repeat" loops. Let's explore these in detail.

5.2.1 For Loops

A "for" loop iterates over a sequence, executing a code block for each element. This is useful when you know the exact number of iterations.

```r
# Iterate over a vector
vec <- c(1, 2, 3, 4, 5)

for (i in vec) {
  print(paste("Value:", i))
}
```

```
[1] "Value: 1"
[1] "Value: 2"
[1] "Value: 3"
[1] "Value: 4"
[1] "Value: 5"
```

In this example, the loop iterates over the vector `vec`, printing each value with a prefix "Value:". You can also use "for" loops with other sequences, such as character vectors or list indices.

```r
# Iterate over the names of a data frame
df <- data.frame(Name = c("Alice", "Bob", "Charlie"), Age = c(25, 30, 35))

for (name in names(df)) {
  print(paste("Column:", name))
}
```

```
[1] "Column: Name"
[1] "Column: Age"
```

Here, the loop iterates over the column names of a data frame, printing the column names one by one.

5.2.2 While Loops

A "while" loop continues to execute as long as a specified condition is **`TRUE`**. This is useful when the number of iterations is unknown or dependent on some condition.

```r
# Example of a simple while loop
counter <- 1

while (counter <= 5) {
  print(paste("Counter:", counter))
  counter <- counter + 1
}
```

```
[1] "Counter: 1"
[1] "Counter: 2"
[1] "Counter: 3"
[1] "Counter: 4"
[1] "Counter: 5"
```

In this example, the loop continues as long as **`counter <= 5`**. Once the condition is **`FALSE`**, the loop terminates.

5.2.3 Repeat Loops

A "repeat" loop is similar to a "while" loop, but it doesn't require a condition to start. It continues until an explicit **`break`** statement is encountered.

```r
# Example of a repeat loop with a break condition
x <- 0

repeat {
  x <- x + 1
  if (x >= 5) {
    break
  }
  print(paste("x is:", x))
}
```

```
[1] "x is: 1"
[1] "x is: 2"
```

```
[1] "x is: 3"
[1] "x is: 4"
```

In this example, the loop continues indefinitely until the condition **`x >= 5`** is met, at which point the **`break`** statement stops the loop.

5.2.4 Nested Loops

You can nest loops within other loops to handle more complex tasks. This is useful for working with multi-dimensional data, such as matrices or data frames.

```
# Example of nested loops to iterate over a matrix
matrix <- matrix(1:9, nrow = 3, ncol = 3)

for (i in 1:nrow(matrix)) {
  for (j in 1:ncol(matrix)) {
    print(paste("Element at (", i, ",", j, ") is", matrix[i, j]))
  }
}
```

```
[1] "Element at ( 1 , 1 ) is 1"
[1] "Element at ( 1 , 2 ) is 4"
[1] "Element at ( 1 , 3 ) is 7"
[1] "Element at ( 2 , 1 ) is 2"
[1] "Element at ( 2 , 2 ) is 5"
[1] "Element at ( 2 , 3 ) is 8"
[1] "Element at ( 3 , 1 ) is 3"
[1] "Element at ( 3 , 2 ) is 6"
[1] "Element at ( 3 , 3 ) is 9"
```

In this example, nested loops are used to iterate over the rows and columns of a matrix, printing each element with its coordinates.

5.3 Loop Control Statements

In addition to **`break`**, R provides **`next`** to skip an iteration and continue with the next one, and **`return`** to exit a loop and return a value (in functions).

```
# Example of using 'next' to skip even numbers
for (i in 1:10) {
  if (i %% 2 == 0) {  # Check if even
    next  # Skip even numbers
  }
  print(i)  # Print odd numbers
}
```

```
[1] 1
[1] 3
[1] 5
[1] 7
[1] 9
```

Here, the **next** statement is used to skip even numbers, resulting in only odd numbers being printed.

5.3.1 Things to Consider While using Loops in R

1. Loop Termination

   Ensure that loops have clear termination conditions to avoid infinite loops. This can happen in `while` or `repeat` loops where the exit condition might not be properly defined or achieved.

   ```
   # Example of a safe loop with clear termination
   counter <- 1
   while (counter <= 10) {
     print(counter)
     counter <- counter + 1  # Ensures the loop will eventually end
   }
   ```

   ```
   [1] 1
   [1] 2
   [1] 3
   [1] 4
   [1] 5
   [1] 6
   [1] 7
   [1] 8
   [1] 9
   [1] 10
   ```

Ensure that `for` loops have a well-defined sequence and dont need exceed the intended iteration bounds.

2. Efficiency

   Loops can be less efficient compared to vectorized operations in R. Consider vectorized approaches when possible to improve performance.

```r
# Inefficient loop-based approach to add two vectors
vec1 <- c(1, 2, 3, 4, 5)
vec2 <- c(6, 7, 8, 9, 10)

result <- c()
for (i in 1:length(vec1)) {
  result[i] <- vec1[i] + vec2[i]
}

# Efficient vectorized approach
result <- vec1 + vec2
```

In this example, the second approach is faster because it leverages R's Vectorized operations, which are optimized for performance.

3. Loop Scope and Variable Reuse

   Consider the scope of variables within loops and avoid variable reuse when it may cause confusion or errors.

```r
# Correct scope and variable reuse
for (i in 1:5) {
  temp <- i * 2
  print(temp)
}
```

```
[1] 2
[1] 4
[1] 6
[1] 8
[1] 10
```

```r
# Potential variable conflict
temp <- 10
for (i in 1:5) {
  temp <- i * 2  # Reuses the 'temp' variable, which may cause conflicts
}
```

Using unique variable names within loops helps avoid accidental overwriting of external variables.

4. Avoiding Side effects

   Loops can sometimes cause unintended side effects, especially when modifying global variables or external data structures. To avoid this, ensure that loop operations are confined to local scope when possible.

   ```
   # Correct: Loop works with local variables
   result <- c()
   for (i in 1:5) {
     local_result <- i * 2
     result <- c(result, local_result)
   }

   # Incorrect: Loop alters global variable unintentionally
   global_var <- 10
   for (i in 1:5) {
     global_var <- global_var + i  # Modifies a global variable in the loop
   }
   ```

In the second example, Modifying `global_var` inside the loop can lead to unintended consequences outside the loop.

5. Loop Control Statements.

   Use `break` , `next`, and `return` carefully to control the flow within loops. These statements should be used with clear logic to avoid unexpected loop termination or skipping.

```
# Correct use of break and next
for (i in 1:10) {
  if (i == 5) {
    break  # Exit the loop when i equals 5
  }
  if (i %% 2 == 0) {
    next  # Skip even numbers
  }
  print(i)
}
```

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

6. Readability and Maintainability

   Write loops in a way that's easy to understand and maintain . This includes adding comments, using clear variable names, and avoiding overly complex nested loops.

   ```r
   # Readable loop with comments and clear variable names
   for (number in 1:10) {
     if (number %% 2 == 0) {
       next  # Skip even numbers
     }
     print(paste("Odd number:", number))
   }
   ```

   ```
   [1] "Odd number: 1"
   [1] "Odd number: 3"
   [1] "Odd number: 5"
   [1] "Odd number: 7"
   [1] "Odd number: 9"
   ```

6 Linear Regression

6.0.1 Introduction to Linear Regression

Linear regression aims to create a model that predicts or explains the dependent variable based on independent variables. It's called "linear" because the relationship between the variables is modeled as a straight line or a plane in a multi-dimensional space.

Linear regression is widely used in various fields, including finance, economics, biology, and engineering, to understand relationships and make forecasts. It can be used for simple predictions or as a building block for more complex models.

6.0.1.1 The Linear Regression Equation

The linear regression equation represents the expected relationship between the dependent variable and the independent variables. In its simplest form, the linear regression equation is:

$y = \beta_0 + \beta_1 * x_1 + \beta_2 * x_2 + \epsilon$

- y is the dependent variable (the variable being predicted or explained).
- $\beta_0$is the intercept (the value of y when all x_k values are zero).
- $\beta_1, \beta_2, \beta_3, ..., \beta_k$ are the coefficients (representing the effect of each independent variable on y).
- $x_1, x_2, x_3, ..., x_k$ are the independent variables (the predictors).
- is the error term (representing the variability not explained by the model).

The goal of linear regression is to find the values of $\beta_1, \beta_2, \beta_3, ..., \beta_k$ that minimize the sum of squared errors, which is the difference between the predicted values and the actual values.

6.0.1.2 Assumptions of Linear Regression

Linear regression relies on several key assumptions. Violations of these assumptions can affect the validity and reliability of the model. The primary assumptions are:

- **Linearity**: The relationship between the dependent variable and each independent variable must be linear. If the relationship is non-linear, linear regression may not be appropriate.
- **Independence**: The observations must be independent of each other. This assumption is critical in cases where data may have a time-related structure or where observations could be correlated.
- **Homoscedasticity**: The variance of the errors must be constant across different values of the independent variables. Uneven variance can lead to inaccurate estimates and biased statistical tests.
- **Normality of Errors**: The residuals (errors) should be approximately normally distributed. This assumption is crucial for hypothesis testing and confidence intervals.
- **No Multicollinearity**: The independent variables should not be highly correlated with each other. High multicollinearity can lead to unstable estimates and make it difficult to interpret the model.

These assumptions should be checked and validated to ensure the linear regression model's robustness and reliability.

6.0.2 Fitting Linear Regression in R

To start with linear regression, you need to load the dataset into your R session. We'll use the `prestige` dataset from the `carData` package, which contains information on various professions, including education, income, women, prestige, census, and type.

6.0.2.1 Loading the Data

First, ensure that the `carData` package is installed and loaded, then load the `prestige` dataset.

```
# Install and load the carData package if needed
if(!require(car)) {
  install.packages("car")
}
```

```
Loading required package: car

Warning: package 'car' was built under R version 4.3.2

Loading required package: carData

Warning: package 'carData' was built under R version 4.3.2
```

```r
library(carData)

# Load the Prestige dataset
data("Prestige")
```

6.0.2.2 Fitting Simple Linear regression

To create a simple linear regression model in R, you can use the **`lm()`** function, which stands for "linear model." The basic structure of the function is as follows:

```r
# Fit a simple linear regression model
reg1 <- lm(prestige ~ education, data = Prestige)
```

In this example, **`prestige`** is the dependent variable, and **`education`** is the independent variable. The tilde **`~`** separates the dependent variable on the left from the independent variable(s) on the right. The **`data`** parameter specifies the dataset used for the model.

After fitting the model, you can use the **`summary()`** function to get detailed information about the linear regression model:

```r
# Get a summary of the linear regression model
summary(reg1)
```

```

Call:
lm(formula = prestige ~ education, data = Prestige)

Residuals:
     Min       1Q   Median       3Q      Max
-26.0397  -6.5228   0.6611   6.7430  18.1636

Coefficients:
```

```
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  -10.732      3.677  -2.919  0.00434 **
education      5.361      0.332  16.148  < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 9.103 on 100 degrees of freedom
Multiple R-squared:  0.7228,    Adjusted R-squared:   0.72
F-statistic: 260.8 on 1 and 100 DF,  p-value: < 2.2e-16
```

This command provides a comprehensive summary of the model, including coefficients, standard errors, t-values, and other statistics that help evaluate the model's performance.

6.0.2.3 Understanding the Model Summary

Here's what the summary output contains and what each section means:

- **Call**: This section shows the formula used to fit the model, confirming the variables and dataset.
- **Coefficients**:
  - **(Intercept)**: The estimated value of the dependent variable when the independent variable is zero.
  - **Education**: The estimated change in the dependent variable for each one-unit increase in education. A positive coefficient indicates a positive relationship.
- **Standard Errors**: These values represent the uncertainty or variability in the coefficient estimates. Smaller standard errors indicate more precise estimates.
- **t-values and p-values**:
  - **t-values**: Used to test the null hypothesis that the coefficient is zero. Higher absolute t-values suggest the coefficient is significant.
  - **p-values**: Indicate the probability of observing the estimated coefficient if the null hypothesis is true. Smaller p-values (e.g., less than 0.05) suggest statistical significance.
- **Residual Standard Error**: Represents the typical size of the residuals (differences between observed and predicted values). Lower values suggest a better fit.
- **R-squared and Adjusted R-squared**:

  - **R-squared**: Indicates the proportion of variance in the dependent variable explained by the independent variable(s). Higher R-squared values suggest a stronger relationship.
  - **Adjusted R-squared**: Adjusted for the number of predictors, providing a more accurate measure of fit for models with multiple independent variables.
- **F-statistic and p-value**: These values test the overall significance of the model. A significant p-value indicates that the model provides a statistically significant relationship.

From the Above Summary Output:

- The intercept in approximately -10.732, indicating the expected prestige when education in zero
- The coefficient for `education` is 5.361, indicating that for every additional unit of education, prestige increases by 5.361
- The model has R-squared value of 0.7228, suggesting that 72% of the variance in prestige is explained by education.
- The p-value for `education` is very low, indicating that the relationship is statistically significant.

6.0.3 Fitting a Multiple Linear Regression Model with Log-Transformed Variables

Multiple linear regression allows for more complex modeling by including multiple independent variables to predict a dependent variable. In this case, we fit a multiple linear regression model with **`prestige`** as the dependent variable, and **`education`**, a log transformation of **`income`**, and **`women`** as the independent variables.

The use of log transformations is common when the independent variable has a skewed distribution or when we want to model the relative change (such as percentages). Here's the model definition and summary:

```r
# Fitting the multiple linear regression with log-transformed income
reg2 <- lm(prestige ~ education + log(income) + women, data = Prestige)
summary(reg2)
```

```

Call:
lm(formula = prestige ~ education + log(income) + women, data = Prestige)

Residuals:
    Min      1Q  Median      3Q     Max
-17.364  -4.429  -0.101   4.316  19.179

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) -110.9658    14.8429  -7.476 3.27e-11 ***
education      3.7305     0.3544  10.527  < 2e-16 ***
log(income)   13.4382     1.9138   7.022 2.90e-10 ***
women          0.0469     0.0299   1.568     0.12
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 7.093 on 98 degrees of freedom
Multiple R-squared:  0.8351,    Adjusted R-squared:   0.83
F-statistic: 165.4 on 3 and 98 DF,  p-value: < 2.2e-16
```

6.0.3.1 Interpreting the Log-Transformed Coefficient

When we include a log-transformed variable in a regression model, the interpretation of its coefficient changes slightly. Instead of a linear increase for each unit, the coefficient represents the change in the dependent variable for a percentage change in the independent variable.

In the `summary(reg2)`, the coefficient for `log(income)` indicates how much `prestige` is expected to change for a percentage change in `income`. To interpret it, you can divide the coefficient by 100. For example, if the coefficient for `log(income)` is 13.4382, the interpretation becomes:

- A 1% increase in `income` leads to an expected increase of about 0.13 points in `prestige`.

6.0.3.2 Handling Log Transformation of Variables with Zeros

If a variable contains zero values, direct log transformation is not possible, as the logarithm of zero is undefined. To address this issue, a common practice is to add a small constant (like 1) to every observation before applying the log transformation. This ensures that all values are positive and can be transformed safely.

6.0.4 Using Categorical Variables in Linear Regression

Categorical variables can play a significant role in linear regression models. This guide focuses on how to include categorical independent variables in linear regression. When the dependent variable is categorical, alternative methods like logistic regression or multinomial regression are more appropriate.

For this example, we consider the **Prestige** dataset and fit a linear regression model where **prestige** is the dependent variable, and the independent variables include **education**, a log transformation of **income**, and a categorical variable, **type**.

```r
# Fit a regression with a categorical variable
reg3 <- lm(prestige ~ education + log(income) + type, data = Prestige)
summary(reg3)
```

```
Call:
lm(formula = prestige ~ education + log(income) + type, data = Prestige)

Residuals:
    Min      1Q  Median      3Q     Max 
-13.511  -3.746   1.011   4.356  18.438 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept) -81.2019    13.7431  -5.909 5.63e-08 ***
education     3.2845     0.6081   5.401 5.06e-07 ***
log(income)  10.4875     1.7167   6.109 2.31e-08 ***
typeprof      6.7509     3.6185   1.866   0.0652 .  
typewc       -1.4394     2.3780  -0.605   0.5465    
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 6.637 on 93 degrees of freedom
  (4 observations deleted due to missingness)
Multiple R-squared:  0.8555,    Adjusted R-squared:  0.8493 
F-statistic: 137.6 on 4 and 93 DF,  p-value: < 2.2e-16
```

6.0.4.1 Understanding Categorical Variables in Regression

In this model, **type** is a categorical or factor variable with three levels: **bc** (blue collar), **prof** (professional, managerial, and technical), and **wc** (white collar). R automatically recognizes it as a factor and treats it accordingly. The missing level in the coefficient summary (**wc** in this case) is considered the baseline or reference group. This means the other levels are compared to this baseline. If you want to get the estimate for the baseline group, remember to add the intercept value.

6.0.4.2 Changing the Reference Group

R automatically selects the first level as the reference group. If you want to change the reference group, you can manually reorder the factor levels.

```r
# Change the reference group to 'bc'
Prestige$type <- factor(Prestige$type, levels = c("bc", "wc", "prof"))

# Fit the regression model with the new reference group
reg3_reorder <- lm(prestige ~ education + log(income) + type, data = Prestige)
summary(reg3_reorder)
```

```

Call:
lm(formula = prestige ~ education + log(income) + type, data = Prestige)

Residuals:
    Min      1Q  Median      3Q     Max 
-13.511  -3.746   1.011   4.356  18.438 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept) -81.2019    13.7431  -5.909 5.63e-08 ***
education     3.2845     0.6081   5.401 5.06e-07 ***
log(income)  10.4875     1.7167   6.109 2.31e-08 ***
typewc       -1.4394     2.3780  -0.605   0.5465    
typeprof      6.7509     3.6185   1.866   0.0652 .  
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 6.637 on 93 degrees of freedom
  (4 observations deleted due to missingness)
Multiple R-squared:  0.8555,    Adjusted R-squared:  0.8493 
F-statistic: 137.6 on 4 and 93 DF,  p-value: < 2.2e-16
```

By reordering the levels, you change the baseline reference group. In this example, the new reference group is `bc` (blue collar).

6.0.4.3 Showing All Factor Levels

If you want to show all factor levels in the coefficient summary, including the baseline, you can remove the intercept in the model. This approach will display separate estimates for each factor level.

```r
# Fit the regression model without intercept to show all factor levels
reg3_showall <- lm(prestige ~ 0 + education + log(income) + type, data = Prestige)
summary(reg3_showall)
```

```

Call:
lm(formula = prestige ~ 0 + education + log(income) + type, data = Prestige)

Residuals:
    Min      1Q  Median      3Q     Max 
-13.511  -3.746   1.011   4.356  18.438 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
education     3.2845     0.6081   5.401 5.06e-07 ***
log(income)  10.4875     1.7167   6.109 2.31e-08 ***
typebc      -81.2019    13.7431  -5.909 5.63e-08 ***
typewc      -82.6413    13.7875  -5.994 3.86e-08 ***
typeprof    -74.4510    15.1175  -4.925 3.65e-06 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 6.637 on 93 degrees of freedom
  (4 observations deleted due to missingness)
Multiple R-squared:  0.9835,    Adjusted R-squared:  0.9826 
F-statistic:  1107 on 5 and 93 DF,  p-value: < 2.2e-16
```

With this approach, all factor levels are displayed with their estimates, allowing you to see the effect of each level independently.

6.0.5 Categorical Variables with Interaction Terms in Linear Regression

In linear regression, interaction terms allow us to study how the relationship between one independent variable and the dependent variable changes depending on the level of another independent variable. When working with categorical variables, interactions can reveal nuanced insights about the data.

Here is an example of a linear regression model with interaction terms using the **`Prestige`** dataset. The model examines how **`prestige`** is influenced by interactions between **`type`** (a categorical variable with levels **`bc`**, **`wc`**, and **`prof`**) and **`education`** as well as **`log(income)`**.

```r
# Fit a regression model with interaction terms
reg4 <- lm(prestige ~ type * (education + log(income)), data = Prestige)

# Alternate ways to define the same interaction model
reg4a <- lm(prestige ~ education + log(income) +
              type + log(income):type + education:type,
            data = Prestige)

reg4b <- lm(prestige ~ education * type +
              log(income) * type, data = Prestige)
```

6.0.6 Using Interaction Terms in Regression

Interaction terms can be defined using the `*` operator in the formula. In this example, `type * (education + log(income))` creates interaction terms between `type` and `education`, and `type` and `log(income)`. This can also be broken down into multiple explicit terms, as shown in `reg4a` and `reg4b`.

6.0.7 Interpretation of Coefficients with Interaction Terms

To illustrate the interpretation of coefficients with interaction terms, consider the following table, which shows the coefficients for each variable across different levels of `type`:

	bc	**wc**	**prof**
Intercept	-120.05	-120.05 + 30.24 = -89.81	-120.05 + 85.16 = -34.89
log(income)	15.98	15.98 - 8.16 = 7.82	15.98 - 9.43 = 6.55
education	2.34	2.34 + 3.64 = 5.98	2.34 + 0.697 = 3.037

This table highlights that:

- The intercept differs based on the level of `type`.
- For `bc`, the intercept is -120.05, but for `wc` and `prof`, it is adjusted based on the interaction terms.
- Similar adjustments occur for `log(income)` and `education`, showing the effect of the interaction term on these coefficients.

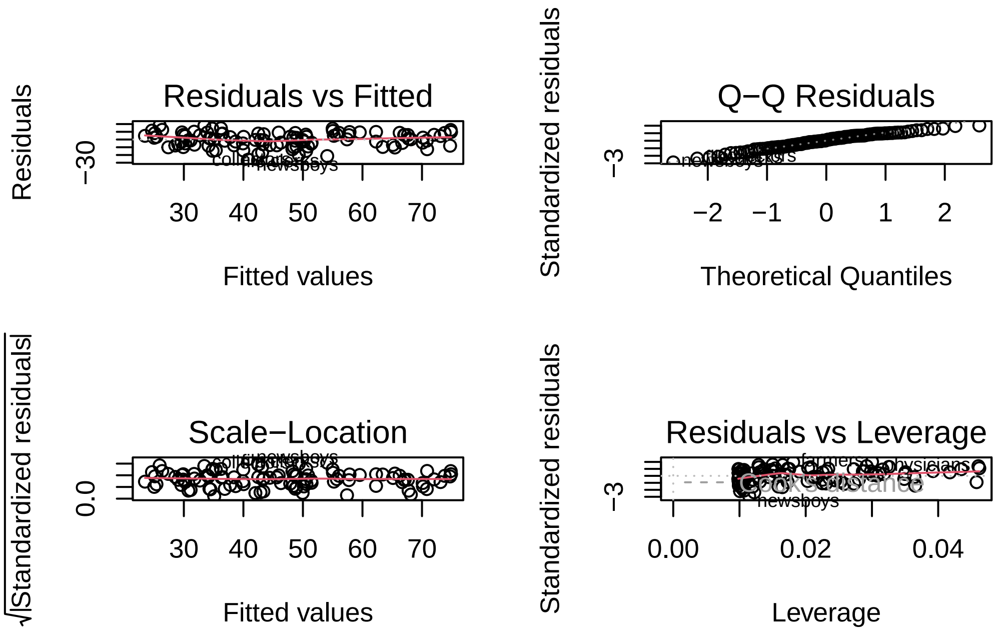
6.0.8 Generating Diagnostic Plots in R

To create diagnostic plots for a regression model, use the **`plot()`** function on your linear model object (**`lm`**). This will generate four key plots that help assess the assumptions of linear regression:

1. **Residuals vs. Fitted**: Checks for non-linearity and heteroscedasticity.
2. **Normal Q-Q**: Evaluates if the residuals follow a normal distribution.
3. **Scale-Location**: Assesses homoscedasticity.
4. **Residuals vs. Leverage**: Identifies influential points.

To create these plots in a single output, you can use **`par(mfrow = c(2, 2))`** to arrange them in a 2x2 grid.

```r
# Assuming you have a linear regression model named 'reg'
par(mfrow = c(2, 2))  # Set the plot layout to 2x2
plot(reg1)             # Generate the diagnostic plots
```



6.0.8.1 Interpreting Diagnostic Plots

Now let's go through each plot and describe what to look for, including potential issues and troubleshooting tips.

1. **Residuals vs. Fitted**

   - This plot should ideally show no distinct pattern. A pattern or curve suggests non-linearity, indicating that a transformation or polynomial regression may be needed. If the spread of residuals widens or narrows as fitted values increase, it suggests heteroscedasticity, which can be addressed with robust standard errors or transformation.

2. **Normal Q-Q**

   - This plot helps check if residuals follow a normal distribution. If the points follow a straight line, residuals are approximately normal. Deviations from this line, especially at the ends, suggest non-normality, which could require transformation or using non-parametric models.

3. **Scale-Location**

   - This plot checks for homoscedasticity. A horizontal line with even spread of points suggests constant variance of residuals. If the points form a funnel or exhibit other patterns, it indicates heteroscedasticity.

4. **Residuals vs. Leverage**

   - This plot helps identify influential points. Points with high leverage (far from the x-axis) or large residuals could significantly affect the regression results. Consider removing outliers or using robust regression techniques if influential points are identified.

6.0.9 Predicting Values in R with Linear Regression

The `predict()` function requires two key inputs: the fitted regression model and a data frame containing the new data points for which you want predictions.

First, let's create a data frame with the values for which you want to predict `prestige`:

```
# Create a new data frame with education, log(income), and women values
new_data <- data.frame(
  education = c(12, 14),   # Example education values for two points
  income = c(15000, 25000), # Example income values (before log transformation)
  women = c(20, 30)         # Example women percentage values
)

# Log-transform the income column
new_data$log_income = log(new_data$income)
```

Next, use the **`predict()`** function with the fitted model **`reg2`** to get the predicted **`prestige`** values:

```
# Predict prestige using the fitted model and new data
predictions <- predict(reg2, newdata = new_data)
print(predictions)  # Display predicted prestige values
```

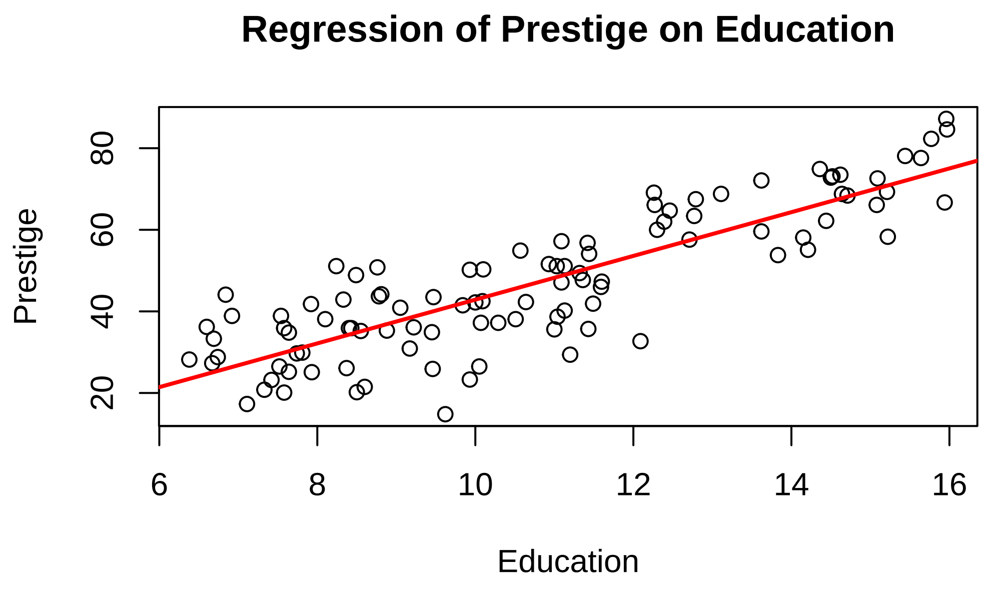
```
       1        2 
63.95751 78.75207 
```

6.0.10 Plotting Regression Line for Simple Linear Regression in Base R

```
# Fit the simple linear regression model
reg1 <- lm(prestige ~ education, data = Prestige)

# Create a scatter plot with base R
plot(Prestige$education, Prestige$prestige,
     xlab = "Education",
     ylab = "Prestige",
     main = "Regression of Prestige on Education")

# Add the regression line
abline(reg1, col = "red", lwd = 2)  # Add regression line with red color and thicker line
```



- `plot()` creates the scatter plot with axes labeled and a title.
- `abline()` with a linear model as an argument adds the regression line.

6.0.11 Optimizing Regression Models: Forward and Backward Selection Techniques

6.0.12 Forward Selection

Forward selection starts with no predictors and adds them one at a time, selecting the variable that improves the model's performance the most at each step. The process continues until adding new predictors does not significantly enhance the model.

6.0.12.1 How Forward Selection Works

1. **Initial Model**: Start with an intercept-only model.
2. **Variable Addition**: Add the predictor that results in the greatest improvement in the model, usually evaluated using a criterion like AIC (Akaike Information Criterion), BIC (Bayesian Information Criterion), or adjusted R-squared.
3. **Iteration**: Repeat step 2, adding one variable at a time, until no additional variables improve the model.

6.0.12.2 Advantages and Disadvantages of Forward Selection

- **Advantages**:
  - Relatively simple and computationally less expensive.
  - Can be useful when there are many potential predictors.
- **Disadvantages**:
  - May miss important interactions or nonlinear effects.
  - Can suffer from multicollinearity if variables are highly correlated.

6.0.13 Backward Selection

Backward selection starts with a model that includes all potential predictors and removes them one at a time, dropping the variable that least improves the model at each step. The process continues until removing variables does not significantly affect the model.

6.0.13.1 How Backward Selection Works

1. **Initial Model**: Start with a model that includes all predictors.
2. **Variable Removal**: Remove the predictor that has the least impact on the model, usually evaluated using a criterion like AIC, BIC, or adjusted R-squared.
3. **Iteration**: Repeat step 2, removing one variable at a time, until further removal significantly worsens the model.

6.0.13.2 Advantages and Disadvantages of Backward Selection

- **Advantages**:
  - Can handle a large number of predictors and consider all of them.
  - May capture interactions or nonlinear effects better than forward selection.
- **Disadvantages**:
  - Requires more computation, especially with a large set of predictors.
  - Can be sensitive to multicollinearity.

6.0.14 Application in R

To apply forward or backward selection in R, you can use the **`step()`** function, which performs both types of selection. Here's an example of backward selection:

```
# Fit a model with all predictors
full_model <- lm(prestige ~ education + log(income) + women, data = Prestige)

# Perform backward selection
backward_model <- step(full_model, direction = "backward")
```

```
Start:  AIC=403.57
prestige ~ education + log(income) + women

              Df Sum of Sq     RSS    AIC
<none>                      4929.9 403.57
- women        1     123.8  5053.6 404.09
- log(income)  1    2480.4  7410.3 443.14
- education    1    5574.4 10504.3 478.73
```

```
backward_model
```

```


Call:
lm(formula = prestige ~ education + log(income) + women, data = Prestige)

Coefficients:
(Intercept)    education  log(income)        women  
  -110.9658       3.7305      13.4382       0.0469  
```

And an example of forward selection:

```
# Fit an intercept-only model
null_model <- lm(prestige ~ 1, data = Prestige)

# Perform forward selection
forward_model <- step(null_model, direction = "forward", scope = ~ education + log(income)
```

```
Start:  AIC=581.41
```

```
prestige ~ 1

              Df Sum of Sq   RSS    AIC
+ education    1   21608.4  8287 452.54
+ log(income)  1   16417.5 13478 502.15
<none>                     29895 581.41
+ women        1     418.6 29477 581.97

Step:  AIC=452.54
prestige ~ education

              Df Sum of Sq    RSS    AIC
+ log(income)  1    3233.4 5053.6 404.09
+ women        1     876.7 7410.3 443.14
<none>                     8287.0 452.54

Step:  AIC=404.09
prestige ~ education + log(income)

        Df Sum of Sq    RSS    AIC
+ women  1    123.75 4929.9 403.57
<none>               5053.6 404.09

Step:  AIC=403.57
prestige ~ education + log(income) + women
```

```
forward_model
```

```

Call:
lm(formula = prestige ~ education + log(income) + women, data = Prestige)

Coefficients:
(Intercept)    education  log(income)        women
  -110.9658       3.7305      13.4382       0.0469
```

References