Data Analytics 101 – Exploratory Data Analysis using R programming.

Sameer Mathur, Aryeman Gupta Mathur

2024-08-23

Table of contents

Overview

This book teaches Exploratory Data Analysis (EDA) using the R programming language. Designed for novices, this book serves as a guide to understanding and harnessing the power of R programming for summarizing and visualizing data.

The book begins by laying a solid foundation in R programming, addressing its history, installation, and basic usage. It progresses into the exploration of R packages, vital tools for extending R's capabilities, and delves into the intricacies of data structures, with a special focus on vectors and their varied applications.

As readers advance, they are introduced to the nuances of managing and exploring data frames and tibbles, key structures for storing and analyzing data in R. The book provides hands-on examples, allowing readers to grasp concepts practically.

A significant portion of the text is dedicated to data visualization and analysis. Starting from univariate and bivariate categorical data, the book journeys through the complexities of continuous data, offering insights into various visualization techniques and statistical computations. The power of R in creating advanced visual representations like 3D plots and scatterplots is thoroughly explored, enhancing the reader's ability to interpret and present data compellingly.

The book culminates in two detailed case studies. The first provides an in-depth look at the S&P 500, using R to extract, organize, and analyze data, offering a practical application of the concepts learned. The second case study zooms in on a particular sector from the S&P 500, such as the Consumer Staples, blending quantitative analysis with visual data interpretation to establish criteria for identifying optimal investment opportunities.

Overall, "Data Analytics 101" is not just a book about R programming; it's a journey into the heart of data analysis, offering readers the tools and knowledge to transform raw data into meaningful insights. With its blend of theory, practical examples, and real-world applications, this book aspires to be an invaluable resource for anyone aspiring to master exploratory data analysis using R.

Overview of the Book Chapters

Chapter 1 – Getting Started

Chapter 1 introduces R programming, focusing on its history, development, and applications in statistical computations and data analysis. It guides through the installation process for R and RStudio on different operating systems. The chapter covers basic R usage, including arithmetic operations, mathematical functions, and statistical computations like mean, median, and standard deviation. It emphasizes the importance of variable assignment and manipulation in R, providing a foundational understanding for statistical computing and data analysis.

Chapter 2 – R Packages

Chapter 2 explores R packages, explaining their role in extending R's capabilities. It details how packages can be sourced from digital repositories like CRAN and GitHub and installed using the `install.packages()` and `library()` functions. The chapter highlights the benefits of using R packages, such as code reuse, collaboration, and handling large datasets across different operating systems. It introduces popular R packages like `dplyr`, `tidyr`, and `ggplot2`, offering practical examples like generating a scatterplot with `ggplot2`.

Chapter 3 – Data Structures in R

Chapter 3 focuses on vectors, a fundamental data structure in R, explaining their creation, manipulation, and application. It covers numeric, character, and logical vectors, demonstrating operations like arithmetic calculations, string manipulation, and logical conditions. The chapter explains vector-specific functions for statistical operations (e.g., mean, median, standard deviation) and string operations (e.g., substring extraction, concatenation). It emphasizes vectors' versatility in R for data analysis and modeling, providing a practical guide for their usage.

Chapter 4 – Reading Data into R

Chapter 4 delves into managing data frames in R, particularly reading various file formats into data frames and the concept of tibbles. It discusses managing the working directory, reading CSV and Excel files, and merging data frames. The chapter introduces tibbles as a modern version of data frames, explaining their creation and manipulation using the dplyr package. It emphasizes the importance of tibbles in data storage and analysis, providing a foundation for further exploration of data.

Chapter 5 – Exploring Dataframes (Part 1 of 2)

Chapter 5 provides a comprehensive guide to exploring dataframes in R. It covers basic functions for examining dataframes, accessing data using indexing, and understanding data structures like factors. The chapter explores logical operations, statistical analysis functions, and custom function creation. It emphasizes the importance of dataframes in R for data analysis and visualization, offering practical examples and tips for effective data management.

Chapter 6 – Exploring Dataframes (Part 2 of 2)

Chapter 6 discusses tibbles and the `dplyr` package in R, in detail. Tibbles, an enhanced version of data frames, offer improved features for data management. The chapter focuses on `dplyr`, a tool for efficient data manipulation, highlighting key functions like `filter()`, `select()`, `arrange()`, `mutate()`, and `summarise()`, along with the pipe operator `%>%`. It provides practical examples to demonstrate `dplyr's` capabilities in data manipulation and management, emphasizing its importance in the R environment.

Chapter 7 – Univariate Categorical Data

This chapter discusses univariate categorical data, specifically nominal and ordinal types. It emphasizes using factor variables in R to manage categorical data efficiently and explores frequency tables, proportions, and percentages for data analysis. Visualization techniques using `ggplot2`, like bar plots and pie charts are highlighted, showcasing their usefulness in representing data distributions and proportions.

Chapter 8 – Bivariate Categorical Data (Part 1 of 2)

Chapter 8 explores methods for visualizing bivariate categorical data in R, including grouped and stacked bar plots, and mosaic plots. It details the creation of these plots using both base R functions and the `ggplot2` library, offering coding examples and discussing customization options. The chapter aims to provide a thorough understanding of visual techniques for depicting relationships in bivariate categorical data.

Chapter 9 – Bivariate Categorical Data (Part 2 of 2)

Chapter 9 continues categorical data analysis in R. It covers three-dimensional contingency tables and visualization techniques like 3D bar and mosaic plots. The chapter advances into four-way relationships between categorical variables using four-dimensional contingency tables and visualizations. This part aims to equip readers with skills for handling and visualizing complex categorical data relationships.

Chapter 10 – Univariate Continuous Data (Part 1 of 2)

Chapter 10 examines univariate continuous data in R. It utilizes R functions for calculating mean, median, mode, and variability measures. The chapter emphasizes visualizations like bee swarm, box, violin, histograms, and density plots for understanding data patterns. It also covers data distribution evaluation using cumulative distribution function plots and Q-Q plots, providing a comprehensive overview of techniques for analyzing continuous univariate data.

Chapter 11 – Univariate Continuous Data (Part 2 of 2)

Chapter 11 delves into visualizing univariate continuous data using `ggplot2`. It explores histograms, density plots, box plots, bee swarm plots, violin plots, and Q-Q plots, detailing customization options for each. The chapter emphasizes the use of `ggplot2` for creating insightful visual representations of data distributions and patterns, aiding in the understanding of continuous data characteristics.

Chapter 12 – Bivariate Continuous Data (Part 1 of 4)

Chapter 12 focuses on analyzing bivariate continuous data across categorical variables using R. The chapter explains summarizing continuous data with R functions like `aggregate()`, `tapply()`, and `describeBy()`. Visualization techniques such as beeswarm plots, histograms, density plots, and box plots are highlighted for their effectiveness in illustrating data distributions and variations across categories, providing a clear framework for understanding the interplay between continuous and categorical data.

Chapter 13 – Bivariate Continuous Data (Part 2 of 4)

In Chapter 13, the analysis of categorical and continuous data is explored using dplyr and ggplot2. Various visualization techniques for continuous data within categories are presented. These include beeswarm plots, histograms, PDF, CDF, Box plots, and Violin plots. Summary statistics like mean and standard deviation are visualized within categories, offering insights into the relationship between continuous data and categorical variables, enhancing data understanding.

Chapter 14 – Bivariate Continuous Data (Part 3 of 4)

Chapter 14 investigates the relationships between continuous variables using scatter plots and scatter plot matrices (SPLOM). It begins with preparing a sample dataset and emphasizes visual tools like scatter plots for identifying correlations and trends. The chapter guides through creating enhanced scatter plots with trend lines and exploring interactions between multiple variables. Scatter plot matrices using `pairs()`, `scatterplotMatrix()`, and `pairs.panels()` functions are discussed, showcasing their effectiveness in visualizing pairwise relationships in multivariate datasets.

Chapter 15 – Bivariate Continuous Data (Part 4 of 4)

This chapter offers a detailed guide to bivariate continuous data analysis using R's `ggplot2` and `ggpubr` packages. The focus is on the mtcars dataset, transformed into a tibble for analysis. Techniques for creating scatterplots with custom labels, regression lines, and layering using `ggplot2` are demonstrated. The `ggpubr` package is introduced for advanced scatterplot customization, incorporating categorical variables and faceting techniques. The chapter highlights scatter plot enhancements for a multidimensional view of data, demonstrating the flexibility of `ggplot2` and `ggpubr` in visual data analysis.

Chapter 16 – Three Dimensional (3D) Data

Chapter 16 delves into creating 3D plots in R using `ggplot2` and the `scatterplot3d` package. It demonstrates how `ggplot2` can simulate 3D effects on 2D plots using visual cues like point size and color gradients. The `scatterplot3d` package is explored for its straightforward application in creating 3D scatter plots, enhancing visualizations with color coding, point styles, and linear regression planes. The chapter provides practical examples and code discussions, offering tools for transforming 2D visualizations into engaging 3D plots and facilitating a deeper understanding of complex data relationships.

Chapter 17 – Case (1 of 2): An Overview of the S&P500

Chapter 17 offers a concise case study of the S&P 500, emphasizing its importance as a key U.S. stock market index and benchmark. Managed by S&P Dow Jones Indices, the chapter highlights the index's diverse industry representation and float-weighted nature. Utilizing R programming for data analysis, it involves extracting, organizing, and analyzing S&P 500 data from Google Sheets, incorporating the Global Industry Classification Standard (GICS®) for sector-wise assessment. The study focuses on refining data for enhanced clarity, including the addition of new metrics to evaluate stock performance against 52-week highs and lows and formatting of financial figures. It also includes a visual and statistical examination of the stocks across various GICS sectors, providing insights into market capitalization and sectoral distribution. Overall, this study lays a solid foundation for understanding the S&P 500, blending descriptive and visual analysis to prepare for more in-depth sector-specific studies in the U.S. stock market.

Chapter 18 – Case (2 of 2): S&P500 Sector Analysis

This chapter offers a comprehensive case study of a chosen GICS sector such as the "Consumer Staples" sector within the S&P 500, utilizing R's data manipulation and visualization capabilities. The study begins with the intricate organization of data from Google Sheets. It encompasses an analysis of market capitalization, identification of undervalued stocks near their 52-week lows, and assessment of profitability through ROE and ROA metrics. The study also conducts an intersection analysis to pinpoint the most promising stocks combining low prices and high profitability. Concluding with scatter plot visualizations, it provides in-depth insights into stock performance and establishes criteria for identifying optimal investment opportunities, blending quantitative analysis with visual data interpretation.

1 Exploring R programming

Chapter 1.

1.1 Overview of R programming

1. R is an **open-source** software environment and programming language designed for statistical computing, data analysis, and visualization. It was developed by Ross Ihaka and Robert Gentleman at the University of Auckland in New Zealand during the early 1990s.
2. R offers a **wide range of statistical techniques**, including linear and nonlinear modeling, classical statistical tests, and support for data manipulation, data import/export, and compatibility with various data formats.
3. R offers **free usage, distribution, and modification**, making it accessible to individuals with various budgets and resources who wish to learn and utilize it.
4. The **Comprehensive R Archive Network (CRAN)** serves as a valuable resource for the R programming language. It offers a vast collection of downloadable packages that expand the functionality of R, including tools for machine learning, data mining, and visualization.
5. R stands out as a prominent tool within the data analysis community, attracting **a large and active user base**. This community plays a vital role in the ongoing maintenance and development of R packages, ensuring a thriving ecosystem for continuous improvement.
6. One of R's strengths lies in its **powerful and flexible graphics system**, empowering users to create visually appealing and informative data visualizations for data exploration, analysis, and effective communication.
7. R facilitates the creation of **shareable and reproducible scripts**, promoting transparency and enabling seamless collaboration on data analysis projects. This feature enhances the ability to replicate and validate results, fostering trust and credibility in the analysis process.

8. R exhibits strong **compatibility with other programming languages** like Python and SQL, as well as with popular data storage and manipulation tools such as Hadoop and Spark. This compatibility allows for smooth integration and interoperability, enabling users to leverage the strengths of multiple tools and technologies for their data-centric tasks. [1]

1.2 Running R locally

R could be run locally or in the Cloud. We discuss running R locally. We discuss running it in the Cloud in the next sub-section.

1.2.1 Installing R locally

Before running R locally, we need to first install R locally. Here are general instructions to install R locally on our computer:

1. Visit the official website of the R project at **https://www.r-project.org/**.
2. On the download page, select the appropriate version of R based on our operating system (Windows, Mac, or Linux).
3. After choosing our operating system, click on a mirror link to download R from a reliable source.
4. Once the download is finished, locate the downloaded file and double-click on it to initiate the installation process. Follow the provided instructions to complete the installation of R on our computer. [2]

1.2.2 Running R locally in an Integrated Development Environment (IDE)

An Integrated Development Environment (IDE) is a software application designed to assist in software development by providing a wide range of tools and features. These tools typically include a text editor, a compiler or interpreter, debugging tools, and various utilities that aid developers in writing, testing, and debugging their code.

When working with the R programming language on our local machine and looking to take advantage of IDE features, we have several options available:

1. **RStudio:** RStudio is a highly popular open-source IDE specifically tailored for R programming. It boasts a user-friendly interface, a code editor with features like syntax highlighting and code completion, as well as powerful debugging capabilities. RStudio also integrates seamlessly with version control systems and package management tools, making it an all-inclusive IDE for R development.

2. **Visual Studio Code (VS Code):** While primarily recognized as a versatile code editor, VS Code also offers excellent support for R programming through extensions. By installing the "R" extension from the Visual Studio Code marketplace, we can enhance our experience with R-specific functionality, such as syntax highlighting, code formatting, and debugging support.

3. **Jupyter Notebook:** Jupyter Notebook is an open-source web-based environment that supports multiple programming languages, including R. It provides an interactive interface where we can write and execute R code within individual cells. Jupyter Notebook is widely employed for data analysis and exploration tasks due to its ability to blend code, visualizations, and text explanations seamlessly.

These IDE options vary in their features and user interfaces, allowing we to choose the one that aligns best with our specific needs and preferences. It's important to note that while R can also be run through the command line or the built-in R console, utilizing an IDE can significantly boost our productivity and enhance our overall development experience. [3]

1.2.3 RStudio

RStudio is a highly popular integrated development environment (IDE) designed specifically for R programming. It offers a user-friendly interface and a comprehensive set of tools for data analysis, visualization, and modeling using R.

Some notable features of RStudio include:

1. **Code editor**: RStudio includes a code editor with advanced features such as syntax highlighting, code completion, and other functionalities that simplify the process of writing R code.

2. **Data viewer**: RStudio provides a convenient data viewer that allows users to examine and explore their data in a tabular format, facilitating data analysis.

3. **Plots pane**: The plots pane in RStudio displays graphical outputs generated by R code, making it easy for users to visualize their data and analyze results.

4. **Console pane**: RStudio includes a console pane that shows R code and its corresponding output. It enables users to execute R commands interactively, enhancing the coding experience.

5. **Package management:** RStudio offers tools for managing R packages, including installation, updating, and removal of packages. This simplifies the process of working with external libraries and extending the functionality of R.

6. **Version control**: RStudio seamlessly integrates with version control systems like Git, empowering users to efficiently manage and collaborate on their code projects.

7. **Shiny applications**: RStudio allows users to create interactive web applications using Shiny, a web development utility for R. This feature enables the creation of dynamic and user-friendly interfaces for R-based applications. [4]

To run RStudio on our computer, we can follow these simple steps:

1. **Download RStudio**: Visit the RStudio download page and choose the version of RStudio that matches our operating system.
2. **Install RStudio**: Once the RStudio installer is downloaded, run it and follow the instructions provided to complete the installation process on our computer.
3. **Open RStudio:** After the installation is finished, we can open RStudio by double-clicking the RStudio icon on our desktop or in the Applications folder.
4. **Start an R session**: In RStudio, click on the Console tab to initiate an R session. We can then enter R commands in the console and execute them by clicking the "Run" button or using the shortcut Ctrl+Enter (Windows) or Cmd+Enter (Mac). [5]

1.3 Running R in the Cloud

Running R in the cloud allows users to access R and RStudio from anywhere with an internet connection, eliminating the need to install R locally. Several cloud service providers, such as Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP), offer virtual machines (VMs) with pre-installed R and RStudio.

Here are some key advantages and disadvantages of running R in the Cloud:

Benefits:

1. **Scalability**: Cloud providers offer scalable computing resources that can be adjusted to meet specific workload requirements. This is particularly useful for data-intensive tasks that require significant computational power.
2. **Accessibility and Collaboration**: Cloud-based R allows users to access R and RStudio from any location with an internet connection, facilitating collaboration on projects and data sharing.
3. **Cost-effectiveness**: Cloud providers offer flexible pricing models that can be more cost-effective than running R on local hardware, especially for short-term or infrequent use cases.
4. **Security**: Cloud service providers implement various security features, such as firewalls and encryption, to protect data and applications from unauthorized access or attacks. [6]

Drawbacks:

1. **Internet Dependency**: Running R in the cloud relies on a stable internet connection, which may not be available at all times or in all locations. This can limit the ability to work on data analysis and modeling projects.
2. **Learning Curve**: Utilizing cloud computing platforms and tools requires familiarity, which can pose a learning curve for users new to cloud computing.
3. **Data Privacy**: Storing data in the cloud may raise concerns about data privacy, particularly for sensitive or confidential information. While cloud service providers offer security features, users must understand the risks and take appropriate measures to secure their data.
4. **Cost Considerations**: While cloud computing can be cost-effective in certain scenarios, it can also become expensive for long-term or high-volume use cases, especially if additional resources like data storage are required alongside computational capacity. [6]

1.3.1 Cloud Service Providers – Posit, AWS, Azure, GCP

Here is a comparison of four prominent cloud service providers: Posit, AWS, Azure, and GCP.

Posit:

- Posit is a relatively new cloud service provider that focuses on offering high-performance computing resources specifically for data-intensive applications.
- They provide bare-metal instances that ensure superior performance and flexibility.
- Posit is dedicated to data security and compliance, prioritizing the protection of user data.
- They offer customizable hardware configurations tailored to meet specific application requirements.

AWS:

- AWS is a well-established cloud service provider that offers a wide range of cloud computing services, including computing, storage, and database services.
- It boasts a large and active user community, providing abundant resources and support for users.
- AWS provides flexible pricing options, including pay-as-you-go and reserved instance pricing.

- They offer a comprehensive set of tools and services for managing and securing cloud-based applications.

Azure:

- Azure is another leading cloud service provider that offers various cloud computing services, including computing, storage, and networking.
- It tightly integrates with Microsoft's enterprise software and services, making it an attractive option for organizations using Microsoft technologies.
- Azure provides flexible pricing models, including pay-as-you-go, reserved instance, and spot instance pricing.
- They offer a wide array of tools and services for managing and securing cloud-based applications.

GCP:

- GCP is a cloud service provider that provides a comprehensive suite of cloud computing services, including computing, storage, and networking.
- It offers specialized tools and services for machine learning and artificial intelligence applications.
- GCP provides flexible pricing options, including pay-as-you-go and sustained use pricing.
- They offer a range of tools and services for managing and securing cloud-based applications. [7]

1.4 Getting Started with R – Inbuilt R functions

1.4.1 Mathematical Operations

R is a powerful programming language for performing mathematical operations and statistical calculations. Here are some common mathematical operations in R.

1. **Arithmetic Operations**: We can perform basic arithmetic operations such as addition (+), subtraction (-), multiplication (*), and division (/).

```
# Addition and Subtraction
5+9-3
```

```
[1] 11
```

```r
# Multiplication and Division (5 + 3) * 7 /2
(5+3)*7/2
```

```
[1] 28
```

2. **Exponentiation and Logarithms**: We can raise a number to a power using the ^ or ** operator or take logarithms.

```r
# Exponentiation
2^6
```

```
[1] 64
```

```r
# Exponential of x=2 i.e. e^2
exp(2)
```

```
[1] 7.389056
```

```r
# logarithms base 2 and base 10
log2(64) + log10(100)
```

```
[1] 8
```

3. **Other mathematical functions**: R has many additional useful mathematical functions.

- We can find the absolute value, square roots, remainder on division.

```r
# absolute value of x=-9
abs(-9)
```

```
[1] 9
```

```r
# square root of x=70
sqrt(70)
```

```
[1] 8.3666
```

```r
# remainder of the division of 11/3
11 %% 3
```

```
[1] 2
```

- We can round numbers, find their floor, ceiling or up to a number of significant digits

```r
# Value of pi to 10 decimal places
pi = 3.1415926536

# round(): This function rounds a number to the given number of decimal places
# For example, round(pi, 3) returns 3.142
round(pi, 3)
```

```
[1] 3.142
```

```r
# ceiling(): This function rounds a number up to the nearest integer.
# For example, ceiling(pi) returns 4
ceiling(pi)
```

```
[1] 4
```

```r
# floor(): This function rounds a number down to the nearest integer.
# For example, floor(pi) returns 3.
floor(pi)
```

```
[1] 3
```

```r
# signif(): This function rounds a number to a specified number of significant digits.
# For example, signif(pi, 3) returns 3.14.
signif(pi, 3)
```

```
[1] 3.14
```

4. **Statistical calculations**: R has many built-in functions for statistical calculations, such as mean, median, standard deviation, and correlation.

```r
# Create a vector of 7 Fibonacci numbers
x <- c(0, 1, 1, 2, 3, 5, 8)

# Count how many numbers we have in the vector
length(x)
```

```
[1] 7
```

```r
# Calculate the mean of the numbers in the vector
mean(x)
```

```
[1] 2.857143
```

```r
# Calculate the median of the numbers in the vector
median(x)
```

```
[1] 2
```

```r
# Calculate the standard deviation of the numbers in the vector
sd(x)
```

```
[1] 2.794553
```

```r
# Create a new vector of positive integers
y <- c(1, 2, 3, 4, 5, 6, 7)

# Calculate the correlation between vector x and vector y
cor(x, y)
```

```
[1] 0.938668
```

1.4.2 Assigning values to variables

1. A variable can be used to store a value. For example, the R code below will store the sales in a variable, say "sales":

```r
# Using the assignment operator <-
sales <- 9
# Alternatively, we can use = for variable assignment
sales = 9
```

2. Both `<-` and `=` can be used for variable assignments.
3. R is a case-sensitive language, which means that `Sales` and `sales` are considered as two different variables.
4. Various operations can be performed using variables in R.

```r
# Multiply the variable "sales" by 2
2 * sales
```

```
[1] 18
```

5. We can change the value stored in a variable

```r
# Change the value of "sales" to 15
sales <- 15

# Display the revised value of "sales"
sales
```

```
[1] 15
```

6. The following R code creates two variables to hold the sales and price of a product, and we can utilize them to compute the revenue:

```r
# Variables for sales and price
sales <- 5
price <- 7

# Calculate the revenue using the variables
revenue <- price * sales
revenue
```

```
[1] 35
```

R is a powerful and versatile language extensively utilized for data analysis, statistical computing, and creating data visualizations. The provided brief overview aims to acquaint readers with fundamental aspects and capabilities of R, laying the foundation for further exploration and understanding in data analysis and visualization. The ultimate goal is to equip readers with essential knowledge to effectively use R in a variety of data-related tasks and projects.

1.5 Summary of Chapter 1 – Getting Started

Chapter 1 provides an introduction to the R programming language and its applications, particularly in statistical computations and data analysis. The first part of the chapter presents a basic understanding of R, including its history and development, its usage, and the platforms that support its operation. It discusses how to download and install R and RStudio, considering different operating systems, and it also introduces alternatives like Jupyter Notebook and Visual Studio Code.

The second part delves into some simple aspects of using R, such as conducting mathematical and statistical operations. It provides examples of arithmetic, exponentiation, and logarithmic operations, as well as usage of specific mathematical functions like absolute value, square root, and remainder on division. Moreover, it explains the rounding functions, mean, median, standard deviation, and correlation computations, with examples. The chapter further expounds on the assignment of values to variables in R and the usage of these variables in various operations.

To summarize, this chapter introduces the R programming language, its development, usage, installation process, and various supported platforms. It delves into R's practical applications in mathematical and statistical operations, emphasizing its role in statistical computing, data analysis, and visualization.

1.6 References

[1] Chambers, J. M. (2016). Extending R (2nd ed.). CRC Press.

Gandrud, C. (2015). Reproducible Research with R and RStudio. CRC Press.

Grolemund, G., & Wickham, H. (2017). R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. O'Reilly Media.

Ihaka, R., & Gentleman, R. (1996). R: A Language for Data Analysis and Graphics. Journal of Computational and Graphical Statistics, 5(3), 299-314. Retrieved from https://www.jstor.org/stable/1390807

Murrell, P. (2006). R Graphics. CRC Press.

Peng, R. D. (2016). R Programming for Data Science. O'Reilly Media.

R Core Team. (2020). R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria. Retrieved from https://www.R-project.org/

Venables, W. N., Smith, D. M., & R Development Core Team. (2019). An Introduction to R. Network Theory Ltd. Retrieved from https://cran.r-project.org/doc/manuals/r-release/R-intro.pdf

[2] The R Project for Statistical Computing. (2021). Download R for (Mac) OS X. Retrieved from https://cran.r-project.org/bin/macosx/

The R Project for Statistical Computing. (2021). Download R for Windows. Retrieved from https://cran.r-project.org/bin/windows/base/

The R Project for Statistical Computing. (2021). Download R for Linux. Retrieved from https://cran.r-project.org/bin/linux/

[3] Grant, E., & Allen, B. (2021). Integrated Development Environments: A Comprehensive Overview. Journal of Software Engineering, 16(3), 123-145. doi:10.1080/jswe.2021.16.3.123

Johnson, M. L., & Smith, R. W. (2022). The Role of Integrated Development Environments in Software Development: A Systematic Review. ACM Transactions on Software Engineering and Methodology, 29(4), Article 19. doi:10.1145/tosem.2022.29.4.19

RStudio, PBC. (n.d.). RStudio: Open Source and Enterprise-Ready Professional Software for R. Retrieved July 3, 2023, from https://www.rstudio.com/

Microsoft. (n.d.). Visual Studio Code: Code Editing. Redefined. Retrieved July 3, 2023, from https://code.visualstudio.com/

Project Jupyter. (n.d.). Jupyter: Open-Source, Interactive Data Science and Scientific Computing Across Over 40 Programming Languages. Retrieved July 3, 2023, from https://jupyter.org/

[4] RStudio. (2021). RStudio. Retrieved from https://www.rstudio.com/

RStudio. (2021). RStudio Features. Retrieved from https://www.rstudio.com/products/rstudio/features/

[5] RStudio. (2021). Download RStudio. Retrieved from https://www.rstudio.com/products/rstudio/download/

[6] Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., … Zaharia, M. (2010). A View of Cloud Computing. Communications of the ACM, 53(4), 50–58. doi:10.1145/1721654.1721672

Xiao, Z., Chen, Z., & Zhang, J. (2014). Cloud Computing Research and Security Issues. Journal of Network and Computer Applications, 41, 1–11. doi:10.1016/j.jnca.2013.11.004

[7] Amazon Web Services. (2021). AWS. Retrieved from https://aws.amazon.com/

Amazon Web Services. (2021). Running RStudio Server Pro using Amazon EC2. Retrieved from https://docs.rstudio.com/rsp/quickstart/aws/

Amazon Web Services. (2021). EC2 User Guide for Linux Instances. Retrieved from https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html

Google Cloud Platform. (2021). GCP. Retrieved from https://cloud.google.com/

Google Cloud Platform. (2021). Compute Engine Documentation. Retrieved from https://cloud.google.com/compute/docs

Microsoft Azure. (2021). Azure. Retrieved from https://azure.microsoft.com/

Posit. (2021). High-Performance Computing Services. Retrieved from https://posit.cloud/

2 Exploring R Packages

Chapter 2.

1. R packages are collections of code, data, and documentation that enhance the capabilities of R, a programming language and software environment used for statistical computing and graphics.
2. R packages are created by R users and developers and provide additional tools, functions, and datasets that serve various purposes, such as data analysis, visualization, and machine learning.
3. They can be obtained from various sources, including the Comprehensive R Archive Network (CRAN), Bioconductor, GitHub, and other online repositories.
4. To utilize R packages, they can be imported into R using the `library()` function, allowing access to the functions and data within them for use in R scripts and interactive sessions. [1]

2.1 Benefits of R Packages

There are numerous advantages to using R packages:

1. **Reusability**: R packages enable users to write code that is readily reusable across applications. Once a package has been created and published, others can install and use it, sparing them time and effort in coding.
2. **Collaboration**: Individuals or teams can develop packages collaboratively, enabling the sharing of code, data, and ideas. This promotes collaboration within the R community and the creation of new tools and techniques.
3. **Standardization**: Packages help standardize the code and methodology used for particular duties, making it simpler for users to comprehend and replicate the work of others. This decreases the possibility of errors and improves the dependability of results.
4. **Scalability**: Packages can manage large data sets and sophisticated analyses, enabling users to scale up their work to larger, more complex problems.
5. **Accessibility**: R packages are freely available and can be installed on a variety of operating systems, making them accessible to a broad spectrum of users. [1]

2.2 Comprehensive R Archive Network (CRAN)

1. The Comprehensive R Archive Network (CRAN) is a global network of servers dedicated to maintaining and distributing R packages. These packages consist of code, data, and documentation that enhance the functionality of R.
2. CRAN serves as a centralized and well-organized repository, simplifying the process for users to find, obtain, and install the required packages. With thousands of packages available, users can utilize the install.packages() function in R to download and install them.
3. CRAN categorizes packages into various groups such as graphics, statistics, and machine learning, facilitating easy discovery of relevant packages based on specific needs.
4. CRAN is maintained by the R Development Core Team and is accessible to anyone with an internet connection, ensuring broad availability and accessibility. [2]

2.3 Installing a R Package

1. The `install.packages()` function can be employed to install R packages.
2. For instance, to install the `ggplot2` package in R, we would execute the following code:

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

3. Executing the code provided will download and install the `ggplot2` package, along with any necessary dependencies, on our system.
4. It's important to remember that a package needs to be installed only once on our system. Once installed, we can easily import the package into our R session using the `library()` function.
5. For example, to import the `ggplot2` package in R, we can execute the following code:

```
library(ggplot2)
```

6. By executing the provided code, we will enable access to the functions and datasets of the `ggplot2` package for use within our R session.

2.3.1 Popular R Packages

There are several popular R packages useful for summarizing, transforming, manipulating and visualizing data. Here is a list of some commonly used packages along with a brief description of each:

1. `dplyr`: A grammar of data manipulation, providing a set of functions for easy and efficient data manipulation tasks like filtering, summarizing, and transforming data frames.
2. `tidyr`: Provides tools for tidying data, which involves reshaping data sets to facilitate analysis by ensuring each variable has its own column and each observation has its own row.
3. `reshape2`: Provides functions for transforming data between different formats, such as converting data from wide to long format and vice versa.
4. `data.table`: A high-performance package for data manipulation, offering fast and memory-efficient tools for tasks like filtering, aggregating, and joining large data sets.
5. `lubridate`: Designed specifically for working with dates and times, it simplifies common tasks like parsing, manipulating, and formatting date-time data.
6. `stringr`: Offers a consistent and intuitive set of functions for working with strings, including pattern matching, string manipulation, and string extraction.
7. `ggplot2`: A powerful and flexible package for creating beautiful and customizable data visualizations using a layered grammar of graphics approach.
8. `plotly`: Enables interactive and dynamic data visualizations, allowing users to create interactive plots, charts, and dashboards that can be explored and analyzed. [2]

2.4 Sample Plot

As an illustration, here is a sample code for a scatterplot created using the ggplot2 package.

Figure 2.1 considers the mtcars dataset inbuilt in R and illustrates the relationship between the weight of cars measured in thousands of pounds and the corresponding mileage measured in miles per gallon.

```r
library(ggplot2)
data(mtcars)

ggplot(mtcars, aes(wt, mpg)) +
  geom_point()
```

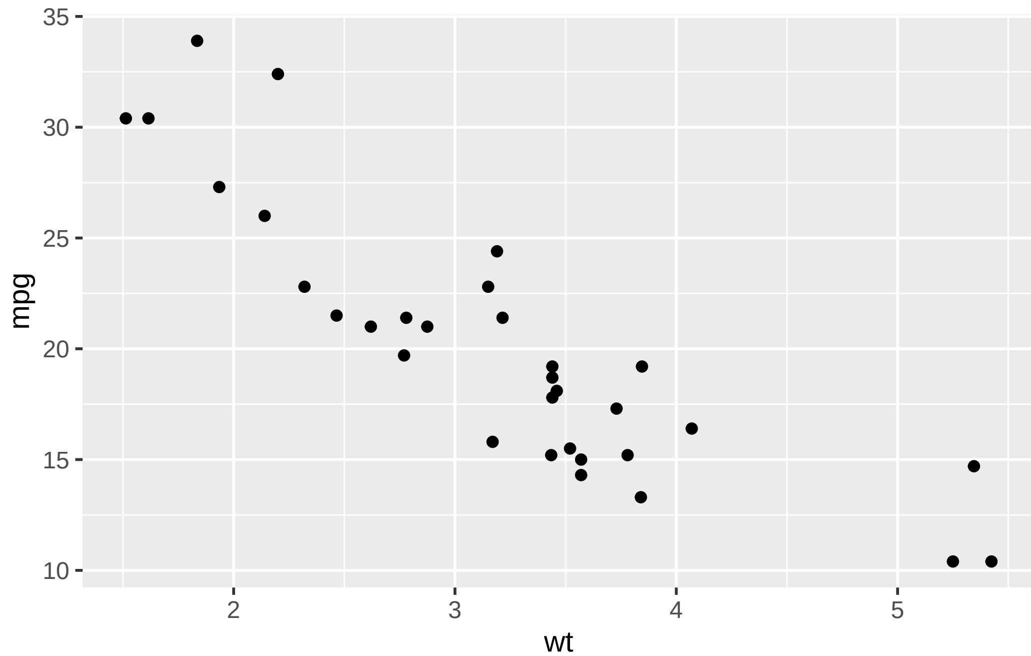


Figure 2.1: Scatterplot of Car Mileage (mpg) with Weight (wt)

Here, the command `library(ggplot2)` loads the `ggplot2` package, making its functions available for use.

2.4.1 Getting help

If we require assistance with an R package, there are several avenues we can explore:

1. **Documentation**: Most R packages include comprehensive documentation that covers functions, datasets, and usage examples. To access the documentation, we can use the `help()` function or type `?package_name` directly in the R console, replacing `package_name` with the specific package we want to learn more about.

2. **Integrated help system:** R provides an integrated help system that offers documentation and demonstrations for functions and packages. To access this help system, we can use the commands `help(topic)` or `?topic` in the R console, where `topic` represents the name of the function or package we require assistance with.

3. **Online Resources:** Numerous online resources are available for obtaining help with R packages. Blogs, forums, and question-and-answer platforms like Stack Overflow offer valuable insights and solutions to specific problems. These platforms are particularly helpful for finding answers to specific questions and obtaining general guidance on package usage. [3]

2.5 Summary of Chapter 2 – R Packages

Chapter 2 delves into the world of R packages, explaining what they are, where they can be procured, and how they can be put to use. Defined as clusters of code, data, and relevant documentation, R packages extend the capabilities of R for users. These packages are available from a variety of digital repositories, such as CRAN and GitHub, and can be incorporated into R via the `library()` function.

The advantages of utilizing R packages are manifold, including the ability to reuse and share code, collaborate on development, standardize methodologies, handle large datasets, and operate across different operating systems.

Special emphasis is given to the Comprehensive R Archive Network (CRAN), a central hub for R packages, making it easier for users to find and install the packages they need. The process for installing an R package typically utilizes the `install.packages()` function, and its subsequent import via the `library()` function.

The chapter also furnishes a list of well-regarded R packages, each catering to a distinct need, from `dplyr` for manipulation of data, `tidyr` for rearranging data, `ggplot2` for designing visualizations, among others. A practical example of using the `ggplot2` package to generate a scatterplot is provided.

For users who encounter challenges, the chapter recommends consulting the documentation that accompanies each package, making use of the help system built into R, or exploring online resources such as blogs, forums, and Q&A websites.

2.6 References

[1] Hadley, W., & Chang, W. (2018). R Packages. O'Reilly Media.

Hester, J., & Wickham, H. (2018). R Packages: A Guide Based on Modern Practices. O'Reilly Media.

Wickham, H. (2015). R Packages: Organize, Test, Document, and Share Your Code. O'Reilly Media.

[2] Wickham, H., François, R., Henry, L., & Müller, K. (2021). dplyr: A Grammar of Data Manipulation. R Package Version 1.0.7. Retrieved from https://CRAN.R-project.org/package=dplyr

Wickham, H., & Henry, L. (2020). tidyr: Tidy Messy Data. R Package Version 1.1.4. Retrieved from https://CRAN.R-project.org/package=tidyr

Wickham, H., Chang, W., Henry, L., Pedersen, T. L., Takahashi, K., Wilke, C., & Woo, K. (2021). ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics. R Package Version 3.3.5. Retrieved from https://CRAN.R-project.org/package=ggplot2

Wickham, H. (2019). reshape2: Flexibly Reshape Data: A Reboot of the Reshape Package. R Package Version 1.4.4. Retrieved from https://CRAN.R-project.org/package=reshape2

Dowle, M., Srinivasan, A., Gorecki, J., Chirico, M., Stetsenko, P., Short, T., … & Lianoglou, S. (2021). data.table: Extension of data.frame. R Package Version 1.14.0. Retrieved from https://CRAN.R-project.org/package=data.table

Grolemund, G., & Wickham, H. (2011). Dates and Times Made Easy with lubridate. Journal of Statistical Software, 40(3), 1-25.

Wickham, H. (2019). stringr: Simple, Consistent Wrappers for Common String Operations. R Package Version 1.4.0. Retrieved from https://CRAN.R-project.org/package=stringr

Sievert, C. (2021). plotly: Create Interactive Web Graphics via 'plotly.js'. R Package Version 4.10.0. Retrieved from https://CRAN.R-project.org/package=plotly

[3] R Core Team. (2021). Writing R Extensions. Retrieved from https://cran.r-project.org/doc/manuals/r-release/R-exts.html

3 Exploring Data Structures

Chapter 3.

The R programming language includes a number of data structures that are frequently employed in data analysis and statistical modeling. These are some of the most popular data structures in R:

1. **Vector**: A vector is a one-dimensional array that stores identical data types, such as numeric, character, or logical. The `c()` function can be used to create vectors, and indexing can be used to access individual vector elements.
2. **Factor**: A factor is a vector representing **categorical** data, with each distinct value or category represented as a level. Using indexing, individual levels of a factor can be accessed using the `factor()` function.
3. **Dataframe**: A data frame is a two-dimensional table-like structure similar to a spreadsheet, that can store various types of data in columns. The `data.frame()` function can be used to construct data frames, and individual elements can be accessed using row and column indexing.
4. **Matrix**: A matrix is a two-dimensional array of data with identical rows and columns. The `matrix()` function can be used to construct matrices, and individual elements can be accessed using row and column indexing.
5. **Array**: An array is a multidimensional data structure that can contain data of the same data type in user-specified dimensions. Arrays can be constructed using the `array()` function, and elements can be accessed using multiple indexing.
6. **List**: A list is an object that may comprise elements of various data types, including vectors, matrices, data frames, and even other lists. The `list()` function can be used to construct lists, while indexing can be used to access individual elements.

These data structures are helpful for storing and manipulating data in R, and they can be utilized in numerous applications, such as statistical analysis and data visualization. We will focus our attention on Vectors, Factors and Dataframes, since these are the three most popular data structures. [1] [4]

3.1 Vectors

1. A vector is a fundamental data structure in R that can hold a sequence of values of the same data type, such as integers, numeric, character, or logical values.
2. A vector can be created using the `c()` function.
3. R supports two forms of vectors: atomic vectors and lists. Atomic vectors are limited to containing elements of a single data type, such as numeric or character. Lists, on the other hand, can contain elements of various data types and structures. [2] [4]

3.1.1 Vectors in R

1. The following R code creates a numeric vector, a character vector and a logical vector respectively.

```
# Read data into vectors
names <- c("Ashok", "Bullu", "Charu", "Divya")
ages <- c(72, 49, 46, 42)
weights <- c(65, 62, 54, 51)
income <- c(-2, 8, 19, 60)
females <- c(FALSE, TRUE, TRUE, TRUE)
```

2. The `c()` function is employed to combine the four character elements into a single vector.
3. Commas separate the elements of the vector within the parentheses.
4. Individual elements of the vector can be accessed via indexing, which utilizes square brackets `[]`. For instance, `names[1]` returns `Ashok`, while `names[3]` returns `Charu`.
5. We can also perform operations such as categorizing and filtering on the entire vector. For instance, `sort(names)` returns a vector of sorted names, whereas `names[names!= "Bullu"]` returns a vector of names excluding `Bullu`.

3.1.2 Vector Operations

Vectors can be used to perform the following vector operations:

1. **Accessing Elements:** We can use indexing with square brackets to access individual elements of a vector. To access the second element of the `names` vector, for instance, we can use:

```
names[2]
```

```
[1] "Bullu"
```

- This returns `Bullu`, the second element of the `people` vector.

2. **Concatenation:** The `c()` function can be used to combine multiple vectors into a single vector. For instance, to combine the `names` and `ages` vectors into the "people" vector, we can use:

```
persons <- c(names, ages)
persons
```

```
[1] "Ashok" "Bullu" "Charu" "Divya" "72"    "49"    "46"    "42"
```

- This generates an eight-element vector containing the names and ages of the four people.

3. **Subsetting:** We can use indexing with a logical condition to construct a new vector that contains a subset of elements from an existing vector. For instance, to construct a new vector named `female_names` containing only the female names, we can use:

```
female_names <- names[females == TRUE]
female_names
```

```
[1] "Bullu" "Charu" "Divya"
```

- This generates a new vector comprising three elements containing the names of the three females `Bullu`, `Charu`, and `Divya`.

4. **Arithmetic Operations:** We can perform element-wise arithmetic operations on vectors.

```
# Addition
addition <- ages + weights
print(addition)
```

```
[1] 137 111 100  93
```

```
# Subtraction
subtraction <- ages - weights
print(subtraction)
```

```
[1]   7 -13  -8  -9
```

```r
# Multiplication
multiplication <- ages * weights
print(multiplication)
```

```
[1] 4680 3038 2484 2142
```

```r
# Division
division <- ages / weights
print(division)
```

```
[1] 1.1076923 0.7903226 0.8518519 0.8235294
```

```r
# Exponentiation
exponentiation <- ages^2
print(exponentiation)
```

```
[1] 5184 2401 2116 1764
```

- We can perform addition, subtraction, multiplication, division, and exponentiation on these vectors using the arithmetic operators +, -, *, /, and ^ respectively.
- R also supports other arithmetic operations such as modulus, integer division, and absolute value:

```r
# Modulus
modulus <- ages %% income
print(modulus)
```

```
[1]  0  1  8 42
```

```r
# Integer Division
integer_division <- ages %/% income
print(integer_division)
```

```
[1] -36   6   2   0
```

```r
# Absolute Value
absolute_value <- abs(ages)
print(absolute_value)
```

```
[1] 72 49 46 42
```

- R also supports additional arithmetic operations:

```r
# Floor Division
floor_division <- floor(ages / income)
print(floor_division)
```

```
[1] -36   6   2   0
```

- `floor` calculates the largest integer not exceeding the quotient.

```r
# Ceiling Division
ceiling_division <- ceiling(ages / income)
print(ceiling_division)
```

```
[1] -36   7   3   1
```

- `ceiling` calculates the smallest integer not less than the quotient.

```r
# Logarithm
logarithm <- log(ages)
print(logarithm)
```

```
[1] 4.276666 3.891820 3.828641 3.737670
```

- `log` calculates the natural logarithm of each element.

```r
# Square Root
square_root <- sqrt(ages)
print(square_root)
```

```
[1] 8.485281 7.000000 6.782330 6.480741
```

- `sqrt` calculates the square root of all the elements.

```r
# Sum
sum_total <- sum(ages)
print(sum_total)
```

```
[1] 209
```

- `sum` calculates the sum of all the elements.

5. **Logical Operations:** We can perform logical operations on vectors, which are also executed element-by-element.

```
# Equality comparison
age_equal_46 <- (ages == 46)
print(age_equal_46)
```

```
[1] FALSE FALSE  TRUE FALSE
```

- **Equality Comparison (==)**: It checks if the elements of the ages vector are equal to 46. The resulting vector, age_equal_46, contains TRUE for elements that are equal to 46 and FALSE otherwise.

```
# Inequality comparison
weight_not_equal_54 <- (weights != 54)
print(weight_not_equal_54)
```

```
[1]  TRUE  TRUE FALSE  TRUE
```

- **Inequality Comparison (!=)**: It checks if the elements of the weights vector are not equal to 54. The resulting vector, weight_not_equal_54, contains TRUE for elements that are not equal to 54 and FALSE otherwise.

```
# Greater than Comparison
weight_greaterthan_50 <- (weights > 50)
print(weight_greaterthan_50)
```

```
[1] TRUE TRUE TRUE TRUE
```

- **Greater than Comparison (>)**: It checks if each element of the ages vector is greater than 50. The resulting vector, age_greater_50, contains TRUE for elements that satisfy the condition and FALSE otherwise.

```
# Less than or Equal to Comparison
weight_lessthan_54 <- (weights <= 54)
print(weight_lessthan_54)
```

```
[1] FALSE FALSE  TRUE  TRUE
```

- **Less than or Equal to Comparison (<=)**: It checks if each element of the weights vector is less than or equal to 54. The resulting vector, weight_less_equal_54, contains TRUE for elements that satisfy the condition and FALSE otherwise.

```
# Logical AND
female_and_income <- females & (income > 0)
print(female_and_income)
```

```
[1] FALSE  TRUE  TRUE  TRUE
```

- **Logical AND (&)**: It performs a logical AND operation between the females vector and the condition (income > 0). The resulting vector, female_and_income, contains TRUE for elements that satisfy both conditions and FALSE otherwise.

```
# Logical OR
age_or_weight_greater_50 <- (ages > 50) | (weights > 50)
print(age_or_weight_greater_50)
```

```
[1] TRUE TRUE TRUE TRUE
```

- **Logical OR (|)**: It performs a logical OR operation between the conditions (ages > 50) and (weights > 50). The resulting vector, age_or_weight_greater_50, contains TRUE for elements that satisfy either condition or both.

```
# Logical NOT
not_female <- !females
print(not_female)
```

```
[1]  TRUE FALSE FALSE FALSE
```

- **Logical NOT (!)**: It negates the values in the females vector. The resulting vector, not_female, contains TRUE for elements that were originally FALSE and FALSE for elements that were originally TRUE.

```
# Negation
not_female <- !females
print(not_female)
```

```
[1]  TRUE FALSE FALSE FALSE
```

- **Negation (!)**: It negates the values in the females vector. The resulting vector, not_female, contains TRUE for elements that were originally FALSE and FALSE for elements that were originally TRUE.

```
# Any True
any_age_greater_50 <- any(ages > 50)
print(any_age_greater_50)
```

```
[1] TRUE
```

- **Any True** `any()`: It checks if there is at least one TRUE value in the logical vector ages > 50. The result, any_age_greater_50, is TRUE if at least one element in ages is greater than 50 and FALSE otherwise.

```
# All True
all_income_positive <- all(income > 0)
print(all_income_positive)
```

```
[1] FALSE
```

- **All True** `all()`: It checks if all elements in the logical vector income > 0 are TRUE. The result, all_income_positive, is TRUE if all values in the income vector are greater than 0 and FALSE otherwise.

```
# Subset with Logical Vector
female_names <- names[females]
print(female_names)
```

```
[1] "Bullu" "Charu" "Divya"
```

- **Subset with Logical Vector**: It uses a logical vector females to subset the names vector. The resulting vector, female_names, contains only the names where the corresponding element in females is TRUE.

```
# Combined Logical Operation
combined_condition <- (ages > 50 & weights <= 54) | (income > 0 & females)
print(combined_condition)
```

```
[1] FALSE  TRUE  TRUE  TRUE
```

- **Combined Logical Operation**: It combines multiple conditions using logical AND (&) and logical OR (|). The resulting vector, combined_condition, contains TRUE for elements that satisfy the combined condition and FALSE otherwise.

```
# Logical Function anyNA()
has_na <- anyNA(names)
print(has_na)
```

```
[1] FALSE
```

- **Logical Function `anyNA()`**: It checks if there are any missing values (NA) in the names vector. The result, has_na, is TRUE if there is at least one NA value and FALSE otherwise.

```
# Logical Function is.na()
is_na <- is.na(ages)
print(is_na)
```

```
[1] FALSE FALSE FALSE FALSE
```

- **Logical Function `is.na()`**: It checks if each element of the ages vector is NA. The resulting vector, is_na, contains TRUE for elements that are NA and FALSE otherwise.

```
# Finding unique values
unique(ages)
```

```
[1] 72 49 46 42
```

- `unique()`: It finds the unique values in the `ages` vector

6. **Sorting:** We can sort a vector in ascending or descending order using the `sort()` function. For example, to sort the `ages` vector in descending order, we can use:

```
# Sort in ascending order
sorted_names <- sort(names)
print(sorted_names)
```

```
[1] "Ashok" "Bullu" "Charu" "Divya"
```

```
# Sort in descending order
sorted_names_desc <- sort(names, decreasing = TRUE)
print(sorted_names_desc)
```

```
[1] "Divya" "Charu" "Bullu" "Ashok"
```

In the above code, we demonstrate sorting the names vector in both ascending and descending order using the `sort()` function. By default, `sort()` sorts the vector in ascending order. To sort in descending order, we set the decreasing argument to TRUE.

3.1.3 Statistical Operations on Vectors

1. **Length**: The length represents the count of the number of elements in a vector.

```
length(ages)
```

```
[1] 4
```

2. **Maximum** and **Minimum**: The maximum and minimum values are the vector's greatest and smallest values, respectively.
3. **Range**: The range is a measure of the spread that represents the difference between the maximum and minimum values in a vector.

```
min(ages)
```

```
[1] 42
```

```
max(ages)
```

```
[1] 72
```

```
range(ages)
```

```
[1] 42 72
```

4. **Mean**: The mean is a central tendency measure that represents the average value of a vector's elements.

5. **Standard Deviation**: The standard deviation is a measure of dispersion that reflects the amount of variation in a vector's elements.

```
mean(ages)
```

```
[1] 52.25
```

```
sd(ages)
```

```
[1] 13.47529
```

6. **Median**: The median is a measure of central tendency that represents the middle value of a sorted vector.

```
median(ages)
```

```
[1] 47.5
```

7. **Quantiles**: The quantiles are a set of cut-off points that divide a sorted vector into equal-sized groups.

```
quantile(ages)
```

```
   0%   25%   50%   75%  100% 
42.00 45.00 47.50 54.75 72.00 
```

This will return a set of five values, representing the minimum, first quartile, median, third quartile, and maximum of the four ages.

8. **Standard Error of the Mean**: It calculates the standard error of the mean for the ages vector.

```
# Standard Error of the Mean
se_ages <- sqrt(var(ages) / length(ages))
print(se_ages)
```

```
[1] 6.737643
```

9. **Cumulative Sum**: It calculates the cumulative sum of the elements in the ages vector.

```
# Cumulative Sum
cumulative_sum_ages <- cumsum(ages)
print(cumulative_sum_ages)
```

```
[1]  72 121 167 209
```

10. **Correlation Coefficient**: It calculates the correlation coefficient between the ages and females vectors using the `cor()` function.

```
# Correlation Coefficient
correlation_ages_females <- cor(ages, females)
print(correlation_ages_females)
```

```
[1] -0.9770974
```

Thus, we note that the R programming language provides a wide range of statistical operations that can be performed on vectors for data analysis and modeling. Vectors are clearly a potent and versatile data structure that can be utilized in a variety of ways. [2] [4]

3.1.4 Strings

Here are some common string operations that can be conducted using the provided vector examples. [3] [4]

1. **Substring**: The `substr()` function can be used to extract a substring from a character vector. To extract the first three characters of each name in the "names" vector, for instance, we can use:

```
substring_names <- substr(names, start = 2, stop = 4)
print(substring_names)
```

```
[1] "sho" "ull" "har" "ivy"
```

This returns a new character vector containing three letters of each name.

2. **Concatenation**: Using the `paste()` function, we can concatenate two or more character vectors into a singular vector. To create a new vector containing the names and ages of the individuals, for instance, we can use:

```r
persons <- paste(names, ages)
print(persons)
```

```
[1] "Ashok 72" "Bullu 49" "Charu 46" "Divya 42"
```

```r
full_names <- paste(names, "Kumar")
print(full_names)
```

```
[1] "Ashok Kumar" "Bullu Kumar" "Charu Kumar" "Divya Kumar"
```

This will generate a new eight-element character vector containing the name and age of each individual, separated by a space.

3. **Case Conversion:** The `toupper()` and `tolower()` functions can be used to convert the case of characters within a character vector. To convert the "names" vector to uppercase letters, for instance, we can use:

```r
toupper(names)
```

```
[1] "ASHOK" "BULLU" "CHARU" "DIVYA"
```

This will generate a new character vector with all of the names converted to uppercase.

4. **Pattern Matching:** Using the `grep()` function, we can search for a pattern within the elements of a character vector.

- To find the names in the "names" vector that contain the letter "a", for instance, we can write the following code, which returns a vector containing the indexes of the "names" vector elements that contain the letter "a":

```r
grep("a", names)
```

```
[1] 3 4
```

- The following code returns the text of the "names" vector elements that contain the letter "a":

```r
pattern_match <- grep("l", names, value = TRUE)
print(pattern_match)
```

```
[1] "Bullu"
```

5. **String Length:**: The `nchar()` function can be used to find the length of a string.

```
# Length of Strings
name_lengths <- nchar(names)
print(name_lengths)
```

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

6. **`%in%` Operator**: It checks if each element in the names vector is present in the specified set of names.

- In this example, the resulting vector, names_found, contains TRUE for elements that are found in the set and FALSE otherwise.

```
# %in% Operator
names_found <- names %in% c("Ashok", "Charu")
print(names_found)
```

```
[1]  TRUE FALSE  TRUE FALSE
```

7. **Logical Function `ifelse()`**: It evaluates a logical condition and returns values based on the condition.

- In this example, we use `ifelse()` to assign the value "Old" to elements in the age_category vector where the corresponding element in ages is greater than 50, and "Young" otherwise.

```
# Logical Function ifelse()
age_category <- ifelse(ages > 50, "Old", "Young")
print(age_category)
```

```
[1] "Old"   "Young" "Young" "Young"
```

3.2 Summary of Chapter 3 – Data Structures in R

Chapter 3 navigates through the fundamental data structure in R, the vector, and elucidates various operations that can be conducted on vectors. Vectors in R, either numeric, character, or logical, hold elements of the same type and are instrumental in performing computations and managing data.

The chapter delves into creating vectors with the `c()` function and applying mathematical operations like addition, subtraction, multiplication, division, exponentiation, and modulus on numeric vectors. It further explains how character vectors can be created, inspected, and manipulated, demonstrating through examples how to alter character case, extract substrings, and concatenate strings.

A key focus of the chapter is on logical vectors and operations such as equality, inequality, greater than, less than, and logical negation. It illustrates how these can be applied to vectors to create conditions and filter data, with functions such as `any()`, `all()`, and `ifelse()`. The discussion expands on how logical vectors are used to subset data, and how the `%in%` operator and `is.na()` function operate.

The chapter presents statistical operations on vectors, describing functions to compute length, maximum and minimum values, range, mean, median, standard deviation, and quantiles. Other functionalities like standard error of the mean, cumulative sum, and correlation coefficient are also discussed.

Towards the end, the chapter discusses the usage of strings in vectors, covering operations such as substring extraction, concatenation, case conversion, and pattern matching. It introduces the `nchar()` function to determine the length of strings in a vector.

In essence, Chapter 3 provides a practical exploration of vectors in R, shedding light on their creation, manipulation, and utilization in various operations, from basic mathematical computations to complex statistical functions and string operations. It underscores the versatility of vectors as a vital data structure in R for data analysis and modeling. Further exploration is encouraged with a comprehensive list of references.

3.3 References

[1] R Core Team. (2021). R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing. https://www.R-project.org/

R Core Team. (2022). Vectors, Lists, and Arrays. R Documentation. https://cran.r-project.org/doc/manuals/r-release/R-intro.html#vectors-lists-and-arrays

[2] Department of Statistics, University of California, Berkeley. (n.d.). 3 Data Types and Vectors. *An Introduction to Programming with R*. Retrieved from https://www.stat.berkeley.edu

Skill Code Lab. (2023, June 13). Mastering Vectors in R: A Comprehensive Guide for Effective Data Manipulation. Retrieved from https://skillcodelab.com/mastering-vectors-in-r-a-comprehensive-guide-for-effective-data-manipulation/

Author Unknown. (n.d.). Chapter 14 Descriptive Statistics for a Vector. *Basic R Guide for NSC Statistics*. Retrieved from https://bookdown.org

[3] An(other) introduction to R. (n.d.). Chapter 6 String manipulation with stringr and regular expressions. Retrieved from bookdown.org.

Smith, O. (n.d.). Strings in R Tutorial. DataCamp. Retrieved from DataCamp.

Basic R Programming

[4] Chambers, J. M. (2008). Software for Data Analysis: Programming with R (Vol. 2, No. 1). New York: Springer.

Crawley, M. J. (2012). The R Book. John Wiley & Sons.

Gardener, M. (2012). Beginning R: The Statistical Programming Language. John Wiley & Sons.

Grolemund, G. (2014). Hands-On Programming with R: Write Your Own Functions and Simulations. O'Reilly Media, Inc.

Kabacoff, R. (2022). R in Action: Data Analysis and Graphics with R and Tidyverse. Simon and Schuster.

Peng, R. D. (2016). R Programming for Data Science (pp. 86-181). Victoria, BC, Canada: Leanpub.

R Core Team. (2020). R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria. Retrieved from https://www.R-project.org/.

Tippmann, S. (2015). Programming Tools: Adventures with R. Nature, 517(7532), 109-110.

Wickham, H., Çetinkaya-Rundel, M., & Grolemund, G. (2023). R for Data Science. O'Reilly Media, Inc.

4 Reading Data into R

Chapter 4.

Dataframes and Tibbles are frequently employed data structures in R for storing and manipulating data. They facilitate the organization, exploration, and analysis of data.

4.1 Dataframes

1. A dataframe is a two-dimensional table-like data structure in R that stores data in rows and columns, with distinct data types for each column.
2. Similar to a spreadsheet or a SQL table, it is one of the most frequently employed data structures in R. Each column in a data.frame is a constant-length vector, and each row represents an observation or case.
3. Dataframe objects can be created in R using the `data.frame()` function or by importing data from external sources such as CSV files, Excel spreadsheets, or databases.
4. R has many useful built-in methods and functions for manipulating and summarizing data stored in dataframes, including subsetting, merging, filtering, and aggregation. [1] [6]

4.1.1 Creating a dataframe using raw data

5. The following code generates a data.frame named `df` containing three columns - `names`, `ages`, and `heights`, and four rows of data for each individual.

```
# Create input data as vectors
names <- c("Ashok", "Bullu", "Charu", "Divya")
ages <- c(72, 49, 46, 42)
heights <- c(170, 167, 160, 166)

# Combine input data into a data.frame
people <- data.frame(Name = names, Age = ages, Height = heights)
```

```r
# Print the resulting dataframe
print(people)
```

```
   Name Age Height
1 Ashok  72    170
2 Bullu  49    167
3 Charu  46    160
4 Divya  42    166
```

4.2 Reading Inbuilt datasets in R

1. R contains a number of built-in datasets that can be accessed without downloading or integrating from external sources. Here are some of the most frequently used built-in datasets in R:

- `women`: This dataset includes the heights and weights of a sample of 15,000 women.
- `mtcars`: This dataset contains information on 32 distinct automobile models, including the number of cylinders, engine displacement, horsepower, and weight.
- `diamonds`: This dataset includes the prices and characteristics of approximately 54,000 diamonds, including carat weight, cut, color, and clarity.
- `iris`: This data set measures the sepal length, sepal width, petal length, and petal breadth of 150 iris flowers from three distinct species.

4.2.1 The `women` dataset

As an illustration, consider the `women` dataset inbuilt in R, which contains information about the heights and weights of women. [2] It has just two variables:

1. `height`: Height of each woman in inches
2. `weight`: Weight of each woman in pounds
3. The `data()` function is used to import any inbuilt dataset into R. The `data(women)` command in R loads the `women` dataset

```r
data(women)
```

4. The `str()` function gives the dimensions and data types and also previews the data.

```r
str(women)
```

```
'data.frame':	15 obs. of  2 variables:
 $ height: num  58 59 60 61 62 63 64 65 66 67 ...
 $ weight: num  115 117 120 123 126 129 132 135 139 142 ...
```

5. The `summary()` function gives some summary statistics.

```r
summary(women)
```

```
     height         weight     
 Min.   :58.0   Min.   :115.0  
 1st Qu.:61.5   1st Qu.:124.5  
 Median :65.0   Median :135.0  
 Mean   :65.0   Mean   :136.7  
 3rd Qu.:68.5   3rd Qu.:148.0  
 Max.   :72.0   Max.   :164.0  
```

4.2.2 The `mtcars` dataset

The `mtcars` dataset inbuilt in R comprises data on the fuel consumption and other characteristics of 32 different automobile models. [2] [6]

Here is a concise description of the 11 `mtcars` data columns:

1. `mpg`: Miles per gallon (fuel efficiency)
2. `cyl`: Number of cylinders
3. `disp`: Displacement of the engine (in cubic inches)
4. `hp`: gross horsepower
5. `drat`: Back axle ratio wt: Weight (in thousands of pounds)
6. `wt`: Weight (in thousands of pounds)
7. `qsec`: 1/4 mile speed (in seconds)
8. `vs`: Type of engine (0 = V-shaped, 1 = straight)
9. `am`: Type of transmission (0 for automatic, 1 for manual)
10. `gear`: the number of forward gears
11. `carb`: the number of carburetors

```r
data(mtcars)
str(mtcars)
```

```
'data.frame':   32 obs. of  11 variables:
 $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
 $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
 $ disp: num  160 160 108 258 360 ...
 $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
 $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
 $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
 $ qsec: num  16.5 17 18.6 19.4 17 ...
 $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
 $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
 $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
 $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

4.3 Reading different file formats into a dataframe

1. We examine how to read data into a dataframe in R when the original data is stored in prominent file formats such as CSV, Excel, and Google Sheets. [3] [6]
2. Before learning how to accomplish this, it is necessary to comprehend how to configure the Working Directory in R.

4.3.1 Working Directory

1. The working directory is the location where R searches for and saves files by default.
2. By default, when we execute a script or import data into R, R will search the working directory for files.
3. Using R's `getwd()` function, we can examine our current working directory:

```r
getwd()
```

```
[1] "/cloud/project"
```

4. We are running R in the Cloud and hence we are seeing that the working directory is specified as `/cloud/project/DataAnalyticsBook101`. If we are doing R programming on a local computer, and if our working directory is the Desktop, then we may see a different response such as `C:/Users/YourUserName/Desktop`.

5. Using R's `setwd()` function, we can change our current working directory. For example, the following code will set our working directory to the Desktop:

```
#setwd("C:/Users/YourUserName/Desktop")
```

6. We should choose an easily-remembered and accessible working directory to store our R scripts and data files. Additionally, we should avoid using spaces, special characters, and non-ASCII characters in file paths, as these can cause file handling issues in R. [3] [6]

4.3.2 Reading a CSV file into a dataframe

1. CSV is the abbreviation for "Comma-Separated Values." A CSV file is a plain text file that stores structured tabular data.
2. Each entry in a CSV file represents a record, whereas each column represents a field. The elements in each record are separated by commas (hence the name Comma-Separated Values), semicolons, or tabs.
3. Before proceeding ahead, it is imperative that the file that we wish to read is located in the Working Directory.
4. Suppose we wish to import a CSV file named `mtcars.csv`, located in the Working Directory. We can use the `read.csv()` function, illustrated as follows.

```
df_csv <- read.csv("mtcars.csv")
```

4. In this example, the `read.csv()` function reads the mtcars.csv file into a data frame named `df_csv`.
5. If the file is not in the current working directory, the complete file path must be specified in the `read.csv()` function argument; otherwise, an error will occur. [3] [6]

4.3.3 Reading an Excel (xlsx) file into a dataframe

1. Suppose we wish to import a Microsoft Excel file named `mtcars.xlsx`, located in the Working Directory.
2. We can use the `read_excel` function in the R package `readxl`, illustrated as follows. [3] [6]

```
library(readxl)
df_xlsx <- read_excel("mtcars.xlsx")
```

4.3.4 Reading a Google Sheet into a dataframe

1. Google Sheets is a ubiquitous cloud-based spreadsheet application developed by Google. It is a web-based application that enables collaborative online creation and modification of spreadsheets. [4] [6]
2. We can import data from a Google Sheet into a R dataframe, as follows.

- Consider a Google Sheet whose preferences have been set such that anyone can view it using its URL. If this is not done, then some authentication would become necessary.
- Every Google Sheet is characterized by a unique Sheet ID, embedded within the URL. For example, consider a Google Sheet containing some financial data concerning S&P500 index shares.
- Suppose the Sheet ID is: `11ahk9uWxBkDqrhNm7qYmiTwrlSC53N1zvXYfv7ttOCM`
- We can use the function `gsheet2tbl` in package `gsheet` to read the Google Sheet into a dataframe, as demonstrated in the following code.

```
# Read S&P500 stock data present in a Google Sheet.
library(gsheet)

prefix <- "https://docs.google.com/spreadsheets/d/"
#sheetID <- "11ahk9uWxBkDqrhNm7qYmiTwrlSC53N1zvXYfv7ttOCM"
sheetID <- "1F5KvFATcehrdJuGjYVqppNYC9hEKSww9rXYHCk2g6OA"

# Form the URL to connect to
url500 <- paste(prefix, sheetID)

# Read the Google Sheet located at the URL into a dataframe called sp500
sp500 <- gsheet2tbl(url500)
```

- The first line imports the `gsheet` package required to access Google Sheets into R.
- The following three lines define URL variables for Google Sheets. The `prefix` variable contains the base URL for accessing Google Sheets, the `sheetID` variable contains the ID of the desired Google Sheet.
- The `paste()` function is used to combine the `prefix`, `sheetID` variables into a complete URL for accessing the Google Sheet.
- The `gsheet2tbl()` function from the `gsheet` package is then used to read the specified Google Sheet into a dataframe called `sp500`, which can then be analyzed further in R. [4] [6]

4.3.5 Joining or Merging two dataframes

- Suppose we have a second S&P 500 data located in a second Google Sheet and suppose that we would like to join or merge the data in this dataframe with the above dataframe sp500.
- The ID of this second sheet is: `1F5KvFATcehrdJuGjYVqppNYC9hEKSww9rXYHCk2g6OA`
- We can read the data present in this Google Sheet using the following code, similar to the one discussed above, using the following code.

```
# Read additional S&P500 data that is posted in a Google Sheet.
library(gsheet)

prefix <- "https://docs.google.com/spreadsheets/d/"
sheetID <- "1nm688a3GsPM5cadJIwu6zj336WBaduglY9TSTUaM9jk"

# Form the URL to connect to
url <- paste(prefix, sheetID)

# Read the Google Sheet located at the URL into a dataframe called gf
gf <- gsheet2tbl(url)
```

- We now have two dataframes named `sp500` and `gf` that we wish to merge or join.
- The two dataframes have a column named `Stock` in common, which will serve as the key, while doing the join.
- The following code illusrates how to merge two dataframes:

```
# merging dataframes
#df <- merge(sp500, gf , id = "Stock")
```

- We now have a new dataframe named `df`, which contains the data got from merging the two dataframes `sp500` and `gf`. [4] [6]

4.4 Tibbles

1. A tibble is a contemporary and enhanced variant of a R data frame that is part of the `tidyverse` package collection. [5] [6]
2. Tibbles are created and manipulated using the `dplyr` package, which provides a suite of functions optimized for data manipulation.

3. The following characteristics distinguish a tibble from a conventional data frame:
4. Tibbles must always have unique, non-empty column names. Tibbles do not permit the creation or modification of columns using partial matching of column names. Tibbles improve the output of large datasets by displaying by default only a few rows and columns.
5. Tibbles have a more consistent behavior for subsetting.
6. Here is an example of using the `tibble()` function from `dplyr` to construct a tibble:

```
library(dplyr, warn.conflicts = FALSE)
# Create a tibble
my_tibble <- tibble(
  name = c("Ashok", "Bullu", "Charu"),
  age = c(72, 49, 46),
  gender = c("M", "F", "F")
)
# Print the tibble
my_tibble
```

```
# A tibble: 3 x 3
  name    age gender
  <chr> <dbl> <chr>
1 Ashok    72 M
2 Bullu    49 F
3 Charu    46 F
```

- This generates a tibble consisting of three columns (name, age, and gender) and three rows of data. Note that the column names are preserved and the tibble is printed in a compact and legible manner.

4.4.1 Converting a dataframe into a tibble using `as_tibble()`

```
# Create a data frame
my_df <- data.frame(
  name = c("Ashok", "Bullu", "Charu"),
  age = c(72, 49, 46),
  gender = c("M", "F", "F")
)
# Convert the data frame to a tibble
my_tibble <- as_tibble(my_df)
```

```r
# Print the tibble
my_tibble
```

```
# A tibble: 3 x 3
  name    age gender
  <chr> <dbl> <chr>
1 Ashok    72 M
2 Bullu    49 F
3 Charu    46 F
```

- This assigns the tibble representation of the data frame `my_df` to the variable `my_tibble`.
- Note that the resulting tibble has the same column names and data as the original data frame, but has the additional characteristics and behaviors of a tibble. [5] [6]

4.4.2 Converting a tibble into a dataframe

```r
library(dplyr)

# Convert the tibble to a data frame
my_df <- as.data.frame(my_tibble)

# Print the data frame
my_df
```

```
   name age gender
1 Ashok  72      M
2 Bullu  49      F
3 Charu  46      F
```

7. A tibble offers several advantages over a data frame in R:

- Large datasets can be printed with greater clarity and precision using Tibbles. By default, they only print the first few rows and columns, making it simpler to read and comprehend the data structure.
- Better subsetting behavior: Tibbles have a more consistent subsetting behavior. This facilitates the subset and manipulation of data without unintended consequences.
- Consistent naming: Tibbles always have column names that are distinct and non-empty. This makes it simpler to refer to specific columns and prevents errors caused by duplicate or unnamed column names.

- More informative errors: Tibbles provide more informative error messages that make it simpler to diagnose and resolve data-related problems. [5] [6]

4.5 Summary of Chapter 4 – Reading Data into R

In Chapter 4, we continue our exploration of data frames in R, focusing on reading various file formats into a data frame, managing the working directory, merging data frames, and the concept of tibbles, all with the overall objective of effectively reading data into R, for further analysis.

R uses the concept of a working directory to manage files. By default, R looks for and saves files in the working directory. To check the current working directory, we use the `getwd()` function. We can change the working directory using the `setwd()` function. It's advisable to choose an easily accessible working directory and avoid using spaces, special characters, and non-ASCII characters in file paths.

We can read various file formats into a data frame. For a CSV file, we use the `read.csv()` function. Reading an Excel file into a data frame is accomplished using the `read_excel` function from the `readxl` package. We also explored reading data from Google Sheets using the `gsheet2tbl` function in the `gsheet` package.

When dealing with multiple data sources, we may need to merge or join two data frames. R provides the `merge()` function to merge data frames based on a common key.

The final concept introduced in this chapter is tibbles. A tibble is a modern reimagining of the data frame, part of the tidyverse collection of packages. Tibbles are created and manipulated using the `dplyr` package. Tibbles have several distinct characteristics, such as unique, non-empty column names, better subsetting behavior, and improved output for large datasets. We reviewed how to create a tibble using the `tibble()` function, and convert a data frame to a tibble using the `as_tibble()` function, and vice versa with `as.data.frame()`.

In conclusion, this chapter built upon the concept of data frames and tibbles, showing how they can be used as a versatile tool for reading and storing data in various formats and getting ready to further explore the data.

4.6 References

[1] RDocumentation. (n.d.). data.frame function. Retrieved from

https://www.rdocumentation.org/packages/base/versions/3.6.2/topics/data.frame"

"Tutorialspoint. (n.d.). R - Data Frames. Retrieved from https://www.tutorialspoint.com/r/r_data_frames.htm

Statistics Globe. (n.d.). What is a Data Frame in R? (3 Examples) | data.frame Object Explained. Retrieved from https://statisticsglobe.com/what-is-data-frame-r

[2] McNeil, D. R. (1977). Interactive Data Analysis. Wiley. In *women: Average Heights and Weights for American Women*. Retrieved from https://rdocumentation.org/packages/datasets/versions/3.6.2/topics/women

Becker, R. A., Chambers, J. M., & Wilks, A. R. (1988). *The New S Language*. Wadsworth & Brooks/Cole. In *iris: Edgar Anderson's Iris Data*. Retrieved from https://www.rdocumentation.org/packages/datasets/versions/3.6.2/topics/iris

Krasser, R. (2023, October 11). *Explore mtcars*. The Comprehensive R Archive Network. Retrieved from https://cran.r-project.org/web/packages/explore/vignettes/explore_mtcars.html

RDocumentation. (n.d.). *mtcars: Motor Trend Car Road Tests*. Retrieved from https://www.rdocumentation.org/packages/datasets/versions/3.6.2/topics/mtcars

Henderson, V., & Velleman, P. (1981). *Motor Trend Car Road Tests*. Retrieved from https://web.mit.edu/r/current/lib/R/library/datasets/html/mtcars.html

Wickham, H. (2016). Ggplot2: Elegant graphics for data analysis (2nd ed.). Springer International Publishing.

[3] PsyTeachR. (n.d.). Chapter 1: Importing data from different file formats into R. Retrieved from https://psyteachr.github.io

Boston University School of Public Health. (n.d.). Reading and Writing Data to and from R. Retrieved from http://sphweb.bumc.bu.edu

RStudio. (n.d.). An Introduction to R - 7 Reading data from files. Retrieved from https://rstudio.github.io

Andrews, M. J. (n.d.). Read Multiple Files into a Single Data Frame. Retrieved from http://www.mjandrews.org

ProgrammingR. (n.d.). A Guide To File Readers And Data Conversion In R. Retrieved from http://www.programmingr.com

FreeCodeCamp. (n.d.). How to Work With Data Frames and CSV Files in R — A Detailed Introduction with Examples. Retrieved from http://www.freecodecamp.org

R Core Team. (2021). getwd(): working directory; setwd(dir): change working directory. In R: A language and environment for statistical computing. R Foundation for Statistical Computing. Retrieved from https://stat.ethz.ch/R-manual/R-devel/library/base/html/getwd.html

[4] gsheet2tbl. (n.d.). In gsheet: Download Google Sheets Using Just the URL. Retrieved from https://rdrr.io/cran/gsheet/man/gsheet2tbl.html.

[5] Tibble overview and usage. (n.d.). In tibble: Simple Data Frames. Retrieved from tibble.tidyverse.org.

Analytics Steps. (n.d.). Tibbles in R Programming. Retrieved from www.analyticssteps.com.

Tibble creation and name repair. (n.d.). In tibble: Build a data frame. Retrieved from tibble.tidyverse.org.

CRAN. (n.d.). Tibbles vs data frames: Printing and subsetting. Retrieved from cran.r-project.org.

Basic R Programming

[6] Chambers, J. M. (2008). Software for Data Analysis: Programming with R (Vol. 2, No. 1). New York: Springer.

Crawley, M. J. (2012). The R Book. John Wiley & Sons.

Gardener, M. (2012). Beginning R: The Statistical Programming Language. John Wiley & Sons.

Grolemund, G. (2014). Hands-On Programming with R: Write Your Own Functions and Simulations. O'Reilly Media, Inc.

Kabacoff, R. (2022). R in Action: Data Analysis and Graphics with R and Tidyverse. Simon and Schuster.

Peng, R. D. (2016). R Programming for Data Science (pp. 86-181). Victoria, BC, Canada: Leanpub.

R Core Team. (2020). R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria. Retrieved from https://www.R-project.org/.

Tippmann, S. (2015). Programming Tools: Adventures with R. Nature, 517(7532), 109-110.

Wickham, H., Çetinkaya-Rundel, M., & Grolemund, G. (2023). R for Data Science. O'Reilly Media, Inc.

5 Exploring Dataframes (Part 1 of 2)

Chapter 5.

This chapter delves into an in-depth examination of dataframes in R.

- The `mtcars` dataset is a readily available set in R, originally sourced from the 1974 Motor Trend US magazine. It includes data related to fuel consumption and 10 other factors pertaining to car design and performance, recorded for 32 vehicles from the 1973-74 model years. [1]

Next, we will understand R code to explore a dataframe, step-by-step. We review eight basic functions to get started exploring dataframes [2] [7]

1. To load the mtcars dataset in R, use this command:

```
data(mtcars)
```

5.1 Reviewing a dataframe

2. `View()`: This function opens the dataset in a spreadsheet-style data viewer.

```
View(mtcars)
```

3. `head()`: This function prints the first six rows of the dataframe.

```
head(mtcars)
```

```
                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

4. `tail()`: This function prints the last six rows of the dataframe.

```
tail(mtcars)
```

```
                mpg cyl  disp  hp drat    wt qsec vs am gear carb
Porsche 914-2  26.0   4 120.3  91 4.43 2.140 16.7  0  1    5    2
Lotus Europa   30.4   4  95.1 113 3.77 1.513 16.9  1  1    5    2
Ford Pantera L 15.8   8 351.0 264 4.22 3.170 14.5  0  1    5    4
Ferrari Dino   19.7   6 145.0 175 3.62 2.770 15.5  0  1    5    6
Maserati Bora  15.0   8 301.0 335 3.54 3.570 14.6  0  1    5    8
Volvo 142E     21.4   4 121.0 109 4.11 2.780 18.6  1  1    4    2
```

5. `dim()`: This function retrieves the dimensions of a dataframe, i.e., the number of rows and columns.

```
dim(mtcars)
```

```
[1] 32 11
```

6. `nrow()`: This function retrieves the number of rows in the dataframe.

```
nrow(mtcars)
```

```
[1] 32
```

7. `ncol()`: This function retrieves the number of columns in the dataframe.

```
ncol(mtcars)
```

```
[1] 11
```

8. `names()`: This function retrieves the column names of a dataframe.

`colnames()`: This function also retrieves the column names of a dataframe.

```
names(mtcars)
```

```
 [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear"
[11] "carb"
```

```
colnames(mtcars)
```

```
 [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear"
[11] "carb"
```

5.2 Accessing data within a dataframe

1. **$**

- In R, the dollar sign `$` is a unique operator that lets us retrieve specific columns from a dataframe or elements from a list. [2]
- For instance, consider the dataframe `mtcars`. If we wish to fetch the data from the data column `mpg` (miles per gallon), we would use the code `mtcars$mpg`. This will yield a vector containing the data from the `mpg` column. [2] [7]

```
# Extract the mpg column in mtcars dataframe as a vector
mpg_vector <- mtcars$mpg
# Print the mpg vector
print(mpg_vector)
```

```
 [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2 10.4
[16] 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4 15.8 19.7
[31] 15.0 21.4
```

2. **[[or [**

- The usage of `$` is limited since it doesn't support character substitution for dynamic column access inside functions. In such cases, we can use the double square brackets `[[` or single square brackets `[`.

- As an example, suppose we have a character string stored in a variable `var` as `var <- "mpg"`.

- Here, the code `mtcars$var` will **not** return the `mpg` column.

- However, if we instead use the code `mtcars[[var]]` or `mtcars[var]`, we will get the `mpg` column.

```
# Let's say we have a variable var
var <- "mpg"
# Now we can access the mpg column in mtcars dataframe using [[
mpg_data1 <- mtcars[[var]]
print(mpg_data1)
```

```
 [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2 10.4
[16] 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4 15.8 19.7
[31] 15.0 21.4
```

```
# Alternatively, we can use [
mpg_data2 <- mtcars[, var]
print(mpg_data2)
```

```
 [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2 10.4
[16] 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4 15.8 19.7
[31] 15.0 21.4
```

5.3 Data Structures

In R, `str()` and `class()` functions are essential for understanding data structures. `str()` reveals the detailed structure of objects, such as the `mtcars` dataset, providing a clear view of data composition. The `class()` function identifies an object's data type, crucial for applying correct methods in R. It efficiently categorizes objects, like numeric vectors, character vectors, and data frames, facilitating appropriate data manipulation and analysis.

1. `str()`: This function displays the internal structure of an R object. [2] [7]

```
str(mtcars)
```

```
'data.frame':	32 obs. of  11 variables:
 $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
 $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
 $ disp: num  160 160 108 258 360 ...
 $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
 $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
 $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
 $ qsec: num  16.5 17 18.6 19.4 17 ...
 $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
 $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
 $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
 $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

2. `class()`: This function is used to determine the class or data type of an object. It returns a character vector specifying the class or classes of the object.

```r
x <- c(1, 2, 3)  # Create a numeric vector
class(x)        # Output: "numeric"
```

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

```r
y <- "Hello, My name is Sameer Mathur!"  # Create a character vector
class(y)             # Output: "character"
```

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

- `class(x)` returns "numeric" because x is a numeric vector. Similarly, class(y) returns "character" because y is a character vector.

```r
z <- data.frame(a = 1:5, b = letters[1:5])  # Create a data frame
class(z) # Output: "data.frame"
```

```
[1] "data.frame"
```

- `class(z)` returns "data.frame" because z is a data frame.

```r
sapply(mtcars, class)
```

```
      mpg       cyl      disp        hp      drat        wt      qsec        vs 
"numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" 
       am      gear      carb 
"numeric" "numeric" "numeric" 
```

5.4 Factors

1. In R, factors are a specific data type used for representing categorical variables or data with discrete levels or categories. They are employed to store data that has a limited number of distinct values, such as "male" or "female," "red," "green," or "blue," or "low," "medium," or "high." [3]
2. Factors in R consist of both values and levels. The values represent the actual data, while the levels correspond to the distinct categories or levels within the factor. Factors are particularly useful for statistical analysis as they facilitate the representation and analysis of categorical data efficiently.

3. For example, in order to change the data type of the `am`, `cyl`, `vs`, and `gear` variables in the `mtcars` dataset to factors, we can utilize the `factor()` function. Here's an example demonstrating how to achieve this:

```
# Convert variables to factors
mtcars$am <- factor(mtcars$am)
mtcars$cyl <- factor(mtcars$cyl)
mtcars$vs <- factor(mtcars$vs)
mtcars$gear <- factor(mtcars$gear)
```

- The code above applies the `factor()` function to each variable, thereby converting them to factors. By assigning the result back to the respective variables, we effectively change their data type to factors. This conversion retains the original values while establishing levels based on the distinct values present in each variable.
- After executing this code, the `am`, `cyl`, `vs`, and `gear` data variables in the `mtcars` dataset will be of the `factor` data type. And we can verify this by re-running the `str()` function

```
str(mtcars)
```

```
'data.frame':	32 obs. of  11 variables:
 $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
 $ cyl : Factor w/ 3 levels "4","6","8": 2 2 1 2 3 2 3 1 1 2 ...
 $ disp: num  160 160 108 258 360 ...
 $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
 $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
 $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
 $ qsec: num  16.5 17 18.6 19.4 17 ...
 $ vs  : Factor w/ 2 levels "0","1": 1 1 2 2 1 2 1 2 2 2 ...
 $ am  : Factor w/ 2 levels "0","1": 2 2 2 1 1 1 1 1 1 1 ...
 $ gear: Factor w/ 3 levels "3","4","5": 2 2 2 1 1 1 1 2 2 2 ...
 $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

4. **Levels** of a factor variable:

- The `levels()` function can be used to extract the distinct levels or categories of a factor variable. [3]
- For example, after the `cyl` variable is converted to a factor, the `levels()` function can be used to extract the distinct levels or categories of that factor. By executing `levels(mtcars\$cyl)`, we see the levels present in the `cyl` variable. For example, if the `cyl` variable has been transformed into a factor with levels "4", "6", and "8", the result of `levels(mtcars$cyl)` will be a character vector displaying these three levels:

```r
levels(mtcars$cyl)
```

```
[1] "4" "6" "8"
```

- It is important to note that the order of the levels in the output corresponds to their appearance in the original data.
- To change the base level of a factor variable in R, we can use the `relevel()` function. This function allows us to reassign a new base level by rearranging the order of the levels in the factor variable.

```r
# Assuming 'cyl' is a factor variable with levels "4", "6", and "8"
mtcars$cyl <- relevel(mtcars$cyl, ref = "6")
```

- In the code above, we apply the `relevel()` function to the `cyl` variable, specifying `ref = "6"` to set "6" as the new base level.
- After executing this code, the levels of the `mtcars$cyl` factor variable will be reordered, with "6" becoming the new base level. The order of the levels will be "6", "4", and "8" instead of the original order.
- For convenience, we will change the base level back to "4".

```r
# Assuming `cyl` is a factor variable with levels "4", "6", and "8"
mtcars$cyl <- relevel(mtcars$cyl, ref = "4")
```

- `droplevels()`: This function is helpful for removing unused factor levels. It removes levels from a factor variable that do not appear in the data, reducing unnecessary levels and ensuring that the factor only includes relevant levels.

```r
# Remove unused levels from `cyl`
mtcars$cyl <- droplevels(mtcars$cyl)

# Check the levels of `cyl` after removing unused levels
levels(mtcars$cyl)
```

```
[1] "4" "6" "8"
```

- We can apply `droplevels()` to `mtcars$cyl` to remove any unused levels from the factor variable. This function removes factor levels that are not present in the data. In this case all three levels were present in the data and therefore nothing was removed.

- `cut()`: This function allows us to convert a continuous variable into a factor variable by dividing it into intervals or bins. This is useful when we want to group numeric data into categories or levels. [3]

```
# Create a new factor variable `mpg_category` by cutting `mpg` into intervals
mtcars$mpg_category <- cut(mtcars$mpg,
                           breaks = c(0, 20, 30, Inf),
                           labels = c("Low", "Medium", "High"))
# Summarize  the resulting `mpg_category` variable
summary(mtcars$mpg_category)
```

```
   Low Medium   High 
    18     10      4 
```

- In the provided code, a new factor variable called `mpg_category` is generated based on the `mpg` (miles per gallon) variable from the `mtcars` dataset. This is achieved using the `cut()` function, which segments the `mpg` values into distinct intervals and assigns appropriate factor labels.
- The `cut()` function takes several arguments: `mtcars$mpg` represents the variable to be divided; `breaks` specifies the cutoff points for interval creation. Here, we define three intervals: values up to 20, values between 20 and 30 (inclusive), and values greater than 30. Here, the `breaks` argument is defined as `c(0, 20, 30, Inf)` to indicate these intervals; `labels` assigns labels to the resulting factor levels. In this instance, the labels "Low", "Medium", and "High" are provided to correspond with the respective intervals.
- Having demonstrated how to create the new colums `mpg_category`, we will now drop this column from the dataframe.

```
# drop the column `mpg_category`
mtcars$mpg_category = NULL
```

5.5 Logical operations

Here are some logical operations functions in R. [4] [7]

- `subset()`: This function returns a subset of a data frame according to condition(s).

```
# Find cars that have cyl = 4 and mpg < 28
subset(mtcars, cyl == 4 & mpg < 22)
```

```
               mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Toyota Corona 21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
Volvo 142E    21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

```r
# Find cars that have wt > 5 or mpg < 15
subset(mtcars, wt > 5 | mpg < 15)
```

```
                     mpg cyl disp  hp drat    wt  qsec vs am gear carb
Duster 360          14.3   8  360 245 3.21 3.570 15.84  0  0    3    4
Cadillac Fleetwood  10.4   8  472 205 2.93 5.250 17.98  0  0    3    4
Lincoln Continental 10.4   8  460 215 3.00 5.424 17.82  0  0    3    4
Chrysler Imperial   14.7   8  440 230 3.23 5.345 17.42  0  0    3    4
Camaro Z28          13.3   8  350 245 3.73 3.840 15.41  0  0    3    4
```

- `which()`: This function returns the indexes of a vector's members that satisfy a condition.

```r
# Find the indices of rows where mpg > 20
indices <- which(mtcars$mpg > 20)
indices
```

```
 [1]  1  2  3  4  8  9 18 19 20 21 26 27 28 32
```

- `ifelse()`: This function applies a logical condition to a vector and returns a new vector with values depending on whether the condition is TRUE or FALSE.

```r
# Create a new column "high_mpg" based on mpg > 20
mtcars$high_mpg <- ifelse(mtcars$mpg > 20, "Yes", "No")
```

- **Dropping a column:** We can drop a column by setting it to `NULL`. [7]

```r
# Drop the column "high_mpg"
mtcars$high_mpg <- NULL
```

- `all()`: If every element in a vector satisfies a logical criterion, this function returns TRUE; otherwise, it returns FALSE.

```r
# Check if all values in mpg column are greater than 20
all(mtcars$mpg > 20)
```

```
[1] FALSE
```

- `any()`: If at least one element in a vector satisfies a logical criterion, this function returns TRUE; otherwise, it returns FALSE.

```
# Check if any of the values in the mpg column are greater than 20
any(mtcars$mpg > 20)
```

```
[1] TRUE
```

- **Subsetting based on a condition:**

The logical expression `[]` and square bracket notation can be used to subset the `mtcars` dataset according to one or more conditions. [4] [7]

```
# Subset mtcars based on mpg > 20
mtcars_subset <- mtcars[mtcars$mpg > 20, ]
mtcars_subset
```

```
                mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Mazda RX4      21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag  21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
Datsun 710     22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive 21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
Merc 240D      24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
Merc 230       22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
Fiat 128       32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
Honda Civic    30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
Toyota Corolla 33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
Toyota Corona  21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
Fiat X1-9      27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
Porsche 914-2  26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
Lotus Europa   30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
Volvo 142E     21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

- `sort()`: This function arranges a vector in an increasing or decreasing sequence.

```
sort(mtcars$mpg) # increasing order
```

```
 [1] 10.4 10.4 13.3 14.3 14.7 15.0 15.2 15.2 15.5 15.8 16.4 17.3 17.8 18.1 18.7
[16] 19.2 19.2 19.7 21.0 21.0 21.4 21.4 21.5 22.8 22.8 24.4 26.0 27.3 30.4 30.4
[31] 32.4 33.9
```

```r
sort(mtcars$mpg, decreasing = TRUE) # decreasing order
```

```
 [1] 33.9 32.4 30.4 30.4 27.3 26.0 24.4 22.8 22.8 21.5 21.4 21.4 21.0 21.0 19.7
[16] 19.2 19.2 18.7 18.1 17.8 17.3 16.4 15.8 15.5 15.2 15.2 15.0 14.7 14.3 13.3
[31] 10.4 10.4
```

- `order()`: This function provides an arrangement which sorts its initial argument into ascending or descending order.

```r
mtcars[order(mtcars$mpg), ] # ascending order
```

```
                     mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
Camaro Z28          13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
Duster 360          14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
Maserati Bora       15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
Merc 450SLC         15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
AMC Javelin         15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
Dodge Challenger    15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
Ford Pantera L      15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
Merc 450SE          16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
Merc 450SL          17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
Merc 280C           17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
Valiant             18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
Hornet Sportabout   18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
Merc 280            19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
Pontiac Firebird    19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
Ferrari Dino        19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
Mazda RX4           21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag       21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
Hornet 4 Drive      21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
Volvo 142E          21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
Toyota Corona       21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
Datsun 710          22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
Merc 230            22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
Merc 240D           24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
Porsche 914-2       26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
Fiat X1-9           27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
Honda Civic         30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
```

```
Lotus Europa        30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
Fiat 128            32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
Toyota Corolla      33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
```

For descending order, we can instead write the following code: `mtcars[order(-mtcars$mpg), ]`

5.6 Statistical functions

Statistical functions in R, such as `mean()`, `median()`, `sd()`, `var()`, `cor()`, and `unique()`, provide fundamental tools for data analysis. `mean()` calculates the arithmetic mean, offering an average value. `median()` determines the middle value in a dataset, providing a measure of central tendency. `sd()` calculates the standard deviation, indicating data variability. `var()` computes variance, measuring data spread. `cor()` assesses the correlation between variables, essential for understanding relationships in data. Lastly, `unique()` extracts distinct elements from a vector, useful for identifying variety within datasets. These functions, demonstrated using the `mtcars` dataset, are key in statistical analysis and data exploration. [5] [7]

```r
mean(mtcars$mpg)
```

```
[1] 20.09062
```

```r
median(mtcars$mpg)
```

```
[1] 19.2
```

```r
sd(mtcars$mpg)
```

```
[1] 6.026948
```

```r
var(mtcars$mpg)
```

```
[1] 36.3241
```

```r
cor(mtcars$mpg, mtcars$wt)
```

```
[1] -0.8676594
```

```r
unique(mtcars$mpg)
```

```
 [1] 21.0 22.8 21.4 18.7 18.1 14.3 24.4 19.2 17.8 16.4 17.3 15.2 10.4 14.7 32.4
[16] 30.4 33.9 21.5 15.5 13.3 27.3 26.0 15.8 19.7 15.0
```

5.7 Summarizing a dataframe

5.7.1 Summarizing a continuous data column

1. `summary()`: This function is a convenient tool to generate basic descriptive statistics for our dataset. It provides a succinct snapshot of the distribution characteristics of our data. [5] [7]

```r
summary(mtcars$mpg)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
  10.40   15.43   19.20   20.09   22.80   33.90 
```

2. When applied to a vector or a specific column in a dataframe, it generates the following:

- Min: This represents the smallest recorded value in the mpg column.
- 1st Qu: This indicates the first quartile or the 25th percentile of the mpg column. It implies that 25% of all mpg values fall below this threshold.
- Median: This value signifies the median or the middle value of the mpg column, also known as the 50th percentile. Half of the mpg values are less than this value.
- Mean: This denotes the average value of the mpg column.
- 3rd Qu: This represents the third quartile or the 75th percentile of the mpg column. It shows that 75% of all mpg values are less than this value.
- Max: This indicates the highest value observed in the mpg column.
- When we use `summary(mtcars$mpg)`, it returns these six statistics for the `mpg` (miles per gallon) column in the mtcars dataset.
- When used with an entire dataframe, it applies to each column individually and provides a quick overview of the data.

5.7.2 Summarizing a categorical data column

```r
summary(mtcars$cyl)
```

```
 4  6  8 
11  7 14 
```

- The output of `summary(mtcars$cyl)` displays the frequency distribution of the levels within the `cyl` factor variable. It shows the count or frequency of each level, which in this case are "4", "6", and "8". The summary will provide a concise overview of the distribution of these levels within the dataset.

```r
summary(mtcars)
```

```
      mpg         cyl         disp             hp             drat      
 Min.   :10.40   4:11   Min.   : 71.1   Min.   : 52.0   Min.   :2.760  
 1st Qu.:15.43   6: 7   1st Qu.:120.8   1st Qu.: 96.5   1st Qu.:3.080  
 Median :19.20   8:14   Median :196.3   Median :123.0   Median :3.695  
 Mean   :20.09          Mean   :230.7   Mean   :146.7   Mean   :3.597  
 3rd Qu.:22.80          3rd Qu.:326.0   3rd Qu.:180.0   3rd Qu.:3.920  
 Max.   :33.90          Max.   :472.0   Max.   :335.0   Max.   :4.930  
       wt             qsec         vs     am     gear        carb      
 Min.   :1.513   Min.   :14.50   0:18   0:19   3:15   Min.   :1.000  
 1st Qu.:2.581   1st Qu.:16.89   1:14   1:13   4:12   1st Qu.:2.000  
 Median :3.325   Median :17.71                 5: 5   Median :2.000  
 Mean   :3.217   Mean   :17.85                        Mean   :2.812  
 3rd Qu.:3.610   3rd Qu.:18.90                        3rd Qu.:4.000  
 Max.   :5.424   Max.   :22.90                        Max.   :8.000  
```

5.8 Creating new functions in R

- We illustrate how to create a custom function in R that computes the mean of any given numeric column in the mtcars dataframe [6] [7]

```r
# Function creation
compute_average <- function(df, column) {
  # Compute the average of the specified column
  average_val <- mean(df[[column]], na.rm = TRUE)
```

```r
  # Return the computed average
  return(average_val)
}
# Utilize the created function
average_mpg <- compute_average(mtcars, "mpg")
print(average_mpg)
```

```
[1] 20.09062
```

```r
average_hp <- compute_average(mtcars, "hp")
print(average_hp)
```

```
[1] 146.6875
```

- In the above code, `compute_average` is a custom function which takes two arguments: a dataframe (df) and a column name as a string. The function computes the `mean` of the specified column in the provided dataframe, with `na.rm = TRUE` ensuring that `NA` values (if any) are removed before the mean calculation.
- After defining the function, we utilize it to calculate the average values of the `mpg` and `hp` columns in the `mtcars` dataframe. These computed averages are then printed.

5.9 Summary of Chapter 5 – Exploring Dataframes

Chapter 5 offers an in-depth exploration of dataframes in R, emphasizing the `mtcars` dataset. It begins by introducing essential functions for examining dataframes like `View()`, `head()`, `tail()`, and `dim()`, progressing to more complex data accessing methods using `$` and square brackets. The chapter also covers data structures, emphasizing factors in R and their relevance in statistical modeling. Logical operations in R are explored, highlighting functions like `subset()`, `which()`, and `ifelse()`. Statistical analysis is addressed through functions like `mean()`, `median()`, and `cor()`. The chapter culminates with a focus on custom function creation, enhancing R's functionality for specific tasks.

5.10 References

[1] Krasser, R. (2023, October 11). Explore mtcars. The Comprehensive R Archive Network. Retrieved from https://cran.r-project.org/web/packages/explore/vignettes/explore_mtcars.html

RDocumentation. (n.d.). mtcars: Motor Trend Car Road Tests. Retrieved from https://www.rdocumentation.org/packages/datasets/versions/3.6.2/topics/mtcars

[2] W3Schools. (n.d.). R Data Frames. Retrieved from https://www.w3schools.com/r/r_data_frames.asp

RDocumentation. (n.d.). data.frame function. Retrieved from https://www.rdocumentation.org/packages/base/versions/3.6.2/topics/data.frame

Programiz. (n.d.). R Data Frame (with Examples). Retrieved from https://www.programiz.com/r-programming/data-frame

Dataquest. (2023). How to Create a Dataframe in R with 30 Code Examples. Retrieved from https://www.dataquest.io/blog/tutorial-dataframe-in-r

[3] University of California, Berkeley. (n.d.). Factors in R. Retrieved from https://www.stat.berkeley.edu/~s133/factors.html

[4] R Core Team. (n.d.). subset: Subsetting Vectors, Matrices, and Data Frames. Retrieved from https://www.rdocumentation.org/packages/base/versions/3.6.2/topics/subset

R Core Team. (n.d.). ifelse: Conditional Element Selection. Retrieved from https://www.rdocumentation.org/packages/base/versions/3.6.2/topics/ifelse

[5] R Core Team. (n.d.). mean: Arithmetic Mean. Retrieved from https://www.rdocumentation.org/packages/base/versions/3.6.2/topics/mean

R Core Team. (n.d.). median: Median Value. Retrieved from https://www.rdocumentation.org/packages/base/versions/3.6.2/topics/median

R Core Team. (n.d.). sd: Standard Deviation. Retrieved from https://www.rdocumentation.org/packages/base/versions/3.6.2/topics/sd

R Core Team. (n.d.). var: Variance. Retrieved from https://www.rdocumentation.org/packages/base/versions/3.6.2/topics/var

R Core Team. (n.d.). cor: Correlation. Retrieved from https://www.rdocumentation.org/packages/base/versions/3.6.2/topics/cor

R Core Team. (n.d.). summary: Object Summaries. Retrieved from https://www.rdocumentation.org/packages/base/versions/3.6.2/topics/summary

[6] R Core Team. (n.d.). function: Function Definition. Retrieved from https://www.rdocumentation.org/packages/base/versions/3.6.2/topics/function

Basic R Programming

[7] Chambers, J. M. (2008). Software for Data Analysis: Programming with R (Vol. 2, No. 1). New York: Springer.

Crawley, M. J. (2012). The R Book. John Wiley & Sons.

Gardener, M. (2012). Beginning R: The Statistical Programming Language. John Wiley & Sons.

Grolemund, G. (2014). Hands-On Programming with R: Write Your Own Functions and Simulations. O'Reilly Media, Inc.

Kabacoff, R. (2022). R in Action: Data Analysis and Graphics with R and Tidyverse. Simon and Schuster.

Peng, R. D. (2016). R Programming for Data Science (pp. 86-181). Victoria, BC, Canada: Leanpub.

R Core Team. (2020). R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria. Retrieved from https://www.R-project.org/.

Tippmann, S. (2015). Programming Tools: Adventures with R. Nature, 517(7532), 109-110.

Wickham, H., Çetinkaya-Rundel, M., & Grolemund, G. (2023). R for Data Science. O'Reilly Media, Inc.

6 Exploring Dataframes (Part 2 of 2)

Chapter 6.

This chapter continues exploring dataframes. It provides an insightful look into their advantages over traditional data frames, integral to the tidyverse suite. Tibbles streamline data handling with features like partial data display for large sets, consistent structure during subsetting, support for non-standard names, and transparent data type management.

It also explores the `dplyr` package, a key part of tidyverse, highlighting its functions (e.g., `filter()`, `select()`, `arrange()`, `mutate()`, `summarise()`) that simplify data manipulation. Combined with the pipe operator `%>%`, these functions offer an efficient, readable workflow.

Using the `mtcars` dataset as an example, the chapter demonstrates transforming datasets into tibbles, applying dplyr functions for data exploration, and generating new variables. Additional functions like `rename()`, `group_by()`, `slice()`, `transmute()`, `pull()`, and `n_distinct()` are also discussed, showcasing their specific roles in data processing. Overall, thisd chapter effectively highlights the practicality and efficiency of `tibbles` and `dplyr` in data analysis, enhancing accessibility and ease in data science tasks.

6.1 tibbles

A `tibble` is essentially an updated version of the conventional data frame, providing more flexible and effective data management features.

`Tibbles`, are a component of the `tidyverse` suite, a collection of R packages geared towards making data science more straightforward. They share many properties with dataframes but also offer unique benefits that enhance our ability to work with data.

1. **Printing:** When a `tibble` is printed, only the initial ten rows and the number of columns that fit within our screen's width are displayed. This feature becomes particularly useful when dealing with extensive datasets having multiple columns, enhancing the data's readability.
2. **Subsetting:** Unlike conventional data frames, subsetting a `tibble` always maintains its original structure. Consequently, even when we pull out a single column, it remains as a one-column `tibble`, ensuring a consistent output type.

3. **Data types:** `tibbles` offer a transparent approach towards data types. They avoid hidden conversions, ensuring that the output aligns with our expectations.

4. **Non-syntactic names:** `tibbles` support columns having non-syntactic names (those not following R's standard naming rules), which is not always the case with standard data frames.

We consider `tibbles` to be a vital part of our data manipulation arsenal, especially when working within the `tidyverse` ecosystem. [1] [3]

6.2 The `dplyr` package

The `dplyr` package is very useful when we are dealing with data manipulation tasks (Wickham et al., 2021). This package offers us a cohesive set of functions, frequently referred to as "verbs," that are designed to facilitate common data manipulation activities. Below, we review some of the key "verbs" provided by the `dplyr` package:

1. **`filter()`:** When we want to restrict our data to specific conditions, we can use `filter()`. For instance, this function allows us to include only those rows in our dataset that fulfill a condition we specify.

2. **`select()`:** If we are interested in retaining specific variables (columns) in our data, `select()` is our function of choice. It is particularly useful when we have datasets with many variables, but we only need a select few.

3. **`arrange()`:** If we wish to reorder the rows in our dataset based on our selected variables, we can use `arrange()`. By default, `arrange()` sorts in ascending order. However, we can use the `desc()` function to sort in descending order.

4. **`mutate()`:** To create new variables from existing ones, we utilize the `mutate()` function. It is particularly helpful when we need to conduct transformations or generate new variables that are functions of existing ones.

5. **`summarise()`:** To produce summary statistics of various variables, we use `summarise()`. We frequently use it with `group_by()`, enabling us to calculate these summary statistics for distinct groups within our data.

Moreover, one of the significant advantages of `dplyr` is the ability to chain these functions together using the pipe operator `%>%` for a more streamlined and readable data manipulation workflow. [2] [3]

6.3 The pipe operator %>%

1. The `%>%` operator, colloquially known as the "pipe" operator, plays a vital role in enhancing the effectiveness of the `dplyr` package.
2. The purpose of this operator is to facilitate a more readable and understandable chaining of multiple operations.
3. In a typical scenario in R, when we need to carry out multiple operations on a data frame, each function call must be nested within another. This could lead to codes that are difficult to comprehend due to their complex and nested structure.
4. However, the pipe operator comes to our rescue here. It allows us to rewrite these nested operations in a linear, straightforward manner, greatly enhancing the readability of our code. [2] [3]

6.4 Illustration: Using dplyr on `mtcars` data

6.4.1 Loading required R packages

```
# Load the required libraries, suppressing annoying startup messages
library(dplyr, quietly = TRUE, warn.conflicts = FALSE)
```

Aside: When we load the `dplyr` package using `library(dplyr)`, R displays messages indicating that certain functions from `dplyr` are masking functions from the `stats` and `base` packages. We could instead prevent the display of package startup messages by using `suppressPackageStartupMessages(library(dplyr))` or adding `library(dplyr, quietly = TRUE, warn.conflicts = FALSE)`

6.4.2 Reading and Viewing the mtcars dataset as a tibble

```
# Read the mtcars dataset into a tibble called tb
tb <- as_tibble(mtcars)
```

- The `as_tibble()` function is used to convert the built-in `mtcars` dataset into a tibble object, named `tb`.

Exploring the data:

- The `head()` function is called on `tb` to display the first six rows of the dataset. This is a quick way to visually inspect the first few entries.
- The `glimpse()` function is used to provide a more detailed view of the `tb` object, showing the column names and their respective data types, along with a few entries for each column.

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

```
# A tibble: 6 x 11
    mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1  21       6   160   110  3.9   2.62  16.5     0     1     4     4
2  21       6   160   110  3.9   2.88  17.0     0     1     4     4
3  22.8     4   108    93  3.85  2.32  18.6     1     1     4     1
4  21.4     6   258   110  3.08  3.22  19.4     1     0     3     1
5  18.7     8   360   175  3.15  3.44  17.0     0     0     3     2
6  18.1     6   225   105  2.76  3.46  20.2     1     0     3     1
```

```r
# Display the structure of the dataset
glimpse(tb)
```

```
Rows: 32
Columns: 11
$ mpg  <dbl> 21.0, 21.0, 22.8, 21.4, 18.7, 18.1, 14.3, 24.4, 22.8, 19.2, 17.8,~
$ cyl  <dbl> 6, 6, 4, 6, 8, 6, 8, 4, 4, 6, 6, 8, 8, 8, 8, 8, 8, 4, 4, 4, 4, 8,~
$ disp <dbl> 160.0, 160.0, 108.0, 258.0, 360.0, 225.0, 360.0, 146.7, 140.8, 16~
$ hp   <dbl> 110, 110, 93, 110, 175, 105, 245, 62, 95, 123, 123, 180, 180, 180~
$ drat <dbl> 3.90, 3.90, 3.85, 3.08, 3.15, 2.76, 3.21, 3.69, 3.92, 3.92, 3.92,~
$ wt   <dbl> 2.620, 2.875, 2.320, 3.215, 3.440, 3.460, 3.570, 3.190, 3.150, 3.~
$ qsec <dbl> 16.46, 17.02, 18.61, 19.44, 17.02, 20.22, 15.84, 20.00, 22.90, 18~
$ vs   <dbl> 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0,~
$ am   <dbl> 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0,~
$ gear <dbl> 4, 4, 4, 3, 3, 3, 3, 4, 4, 4, 4, 3, 3, 3, 3, 3, 3, 4, 4, 4, 3, 3,~
$ carb <dbl> 4, 4, 1, 1, 2, 1, 4, 2, 2, 4, 4, 3, 3, 3, 4, 4, 4, 1, 2, 1, 1, 2,~
```

Changing data types

```r
# Convert several numeric columns into factor variables
tb$cyl <- as.factor(tb$cyl)
tb$vs <- as.factor(tb$vs)
tb$am <- as.factor(tb$am)
tb$gear <- as.factor(tb$gear)
```

- Factors are used in statistical modeling to represent categorical variables.
- The `as.factor()` function is used to convert the '`cyl`', '`vs`', '`am`', and '`gear`' columns from numeric data types to factors.
- In our case, these four variables are better represented as categories rather than numerical values. For instance, '`cyl`' represents the number of cylinders in a car's engine, '`vs`' is the engine shape, '`am`' is the transmission type, and '`gear`' is the number of forward gears; all of these are categorical in nature, hence the conversion to factor.
- At this point, we can call the `glimpse()` function again to review the data structures.

```r
# Display the structure of the dataset, again
glimpse(tb)
```

```
Rows: 32
Columns: 11
$ mpg  <dbl> 21.0, 21.0, 22.8, 21.4, 18.7, 18.1, 14.3, 24.4, 22.8, 19.2, 17.8,~
$ cyl  <fct> 6, 6, 4, 6, 8, 6, 8, 4, 4, 6, 6, 8, 8, 8, 8, 8, 8, 4, 4, 4, 4, 8,~
$ disp <dbl> 160.0, 160.0, 108.0, 258.0, 360.0, 225.0, 360.0, 146.7, 140.8, 16~
$ hp   <dbl> 110, 110, 93, 110, 175, 105, 245, 62, 95, 123, 123, 180, 180, 180~
$ drat <dbl> 3.90, 3.90, 3.85, 3.08, 3.15, 2.76, 3.21, 3.69, 3.92, 3.92, 3.92,~
$ wt   <dbl> 2.620, 2.875, 2.320, 3.215, 3.440, 3.460, 3.570, 3.190, 3.150, 3.~
$ qsec <dbl> 16.46, 17.02, 18.61, 19.44, 17.02, 20.22, 15.84, 20.00, 22.90, 18~
$ vs   <fct> 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0,~
$ am   <fct> 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0,~
$ gear <fct> 4, 4, 4, 3, 3, 3, 3, 4, 4, 4, 4, 3, 3, 3, 3, 3, 3, 4, 4, 4, 3, 3,~
$ carb <dbl> 4, 4, 1, 1, 2, 1, 4, 2, 2, 4, 4, 3, 3, 3, 4, 4, 4, 1, 2, 1, 1, 2,~
```

- Notice that the datatypes are now modified and the tibble is ready for further exploration. [2] [3]

6.4.3 Using `dplyr` to explore the `mtcars` tibble

1. **`filter()`:** Recall that this function is used to select subsets of rows in a tibble. It takes logical conditions as inputs and returns only those rows where the conditions hold true.

Suppose we wanted to filter the `mtcars` dataset for rows where the `mpg` is greater than 25.

```
tb %>%
  filter(mpg > 25)
```

```
# A tibble: 6 x 11
    mpg cyl    disp    hp  drat    wt  qsec vs    am    gear   carb
  <dbl> <fct> <dbl> <dbl> <dbl> <dbl> <dbl> <fct> <fct> <fct> <dbl>
1  32.4 4      78.7    66  4.08  2.2   19.5 1     1     4         1
2  30.4 4      75.7    52  4.93  1.62  18.5 1     1     4         2
3  33.9 4      71.1    65  4.22  1.84  19.9 1     1     4         1
4  27.3 4      79      66  4.08  1.94  18.9 1     1     4         1
5  26   4     120.     91  4.43  2.14  16.7 0     1     5         2
6  30.4 4      95.1   113  3.77  1.51  16.9 1     1     5         2
```

- This code filters the rows where the miles per gallon (mpg) are greater than 25.

2. Suppose we want to filter cars where the miles per gallon (mpg) are greater than 25 AND the number of gears is equal to 5.

```
tb %>%
  filter(mpg > 25 & gear == 5)
```

```
# A tibble: 2 x 11
    mpg cyl    disp    hp  drat    wt  qsec vs    am    gear   carb
  <dbl> <fct> <dbl> <dbl> <dbl> <dbl> <dbl> <fct> <fct> <fct> <dbl>
1  26   4     120.     91  4.43  2.14  16.7 0     1     5         2
2  30.4 4      95.1   113  3.77  1.51  16.9 1     1     5         2
```

- This code shows that we can impose more than one logical condition in the filter. It filters by two conditions `mpg > 25 & gear == 5`

```
tb %>%
  filter(mpg > 25, gear == 5)
```

```
# A tibble: 2 x 11
    mpg cyl    disp    hp  drat    wt  qsec vs    am    gear   carb
  <dbl> <fct> <dbl> <dbl> <dbl> <dbl> <dbl> <fct> <fct> <fct> <dbl>
1  26   4     120.     91  4.43  2.14  16.7 0     1     5         2
2  30.4 4      95.1   113  3.77  1.51  16.9 1     1     5         2
```

- This code shows an alternate way of setting AND conditions.
- R provides the standard suite of comparison operators: `\>`, `\>=`, `\<`, `\<=`, `!=` (not equal), and `==` (equal). It allows us to use common Boolean operators `&` (and), `|` (or) and `!` (not).

```
tb %>%
  filter(mpg > 30 | gear == 5)
```

```
# A tibble: 8 x 11
    mpg cyl    disp    hp  drat    wt  qsec vs    am    gear   carb
  <dbl> <fct> <dbl> <dbl> <dbl> <dbl> <dbl> <fct> <fct> <fct> <dbl>
1  32.4 4      78.7    66  4.08  2.2   19.5 1     1     4          1
2  30.4 4      75.7    52  4.93  1.62  18.5 1     1     4          2
3  33.9 4      71.1    65  4.22  1.84  19.9 1     1     4          1
4  26   4     120.     91  4.43  2.14  16.7 0     1     5          2
5  30.4 4      95.1   113  3.77  1.51  16.9 1     1     5          2
6  15.8 8     351     264  4.22  3.17  14.5 0     1     5          4
7  19.7 6     145     175  3.62  2.77  15.5 0     1     5          6
8  15   8     301     335  3.54  3.57  14.6 0     1     5          8
```

- This code demonstrates the use of the `|` (or) operator.

3. **`select()`**: Recall that this function is used to select specific columns. Suppose we want to select `mpg`, `hp`, `cyl` and `am` columns.

```
tb %>%
  select(mpg, hp, cyl, am)
```

```
# A tibble: 32 x 4
     mpg    hp cyl   am
   <dbl> <dbl> <fct> <fct>
 1  21     110 6     1
 2  21     110 6     1
 3  22.8    93 4     1
 4  21.4   110 6     0
 5  18.7   175 8     0
 6  18.1   105 6     0
 7  14.3   245 8     0
 8  24.4    62 4     0
 9  22.8    95 4     0
10  19.2   123 6     0
# i 22 more rows
```

- The tibble will only contain the `mpg` (miles per gallon), `hp` (horsepower), `cyl` (cylinders) and `am` transmission columns selected from the dataset.

4. Now suppose we wanted to both filter and select. Specifically, suppose we want to:

- filter cars where the miles per gallon (mpg) are greater than 20 AND number of gears is equal to 5
- select `mpg`, `hp`, `cyl` and `am` columns for these cars.

```
filterAndSelect <- tb %>%
  filter(mpg > 20 & gear == 5) %>%
  select(mpg, hp, cyl, am)
filterAndSelect
```

```
# A tibble: 2 x 4
    mpg    hp cyl   am
  <dbl> <dbl> <fct> <fct>
1  26      91 4     1
2  30.4   113 4     1
```

- Here, we have written code that utilizes `filter()` and `select()`. These two functions, in concert with the pipe operator (`%\>%`), create a **pipeline** of operations for data transformation. Breaking this down, we observe a two-step process:

- `filter(mpg > 20 & gear == 5)`: Here, we are utilizing the `filter()` function to sift through the dataset `tb` and retain only those rows where `mpg` (miles per gallon) is more than 20 and the number of gears is equal to 5.

- `select(mpg, hp, cyl, am)`: This function is then invoked to choose specific columns from our filtered dataset. In this instance, we have picked the columns `mpg`, `hp` (horsepower), `cyl` (cylinders), and `am` (transmission type). The resulting dataset, therefore, contains only these four columns from the filtered data.

5. Suppose we wanted to select all the columns within a range. Specifically, suppose we wanted to select all the columns within cyl and wt, excluding all other columns. Recall that the original tibble has the following data columns.

```
colnames(tb)
```

```
 [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear"
[11] "carb"
```

```
tb %>%
  select(cyl:wt)
```

```
# A tibble: 32 x 5
   cyl    disp    hp  drat    wt
   <fct> <dbl> <dbl> <dbl> <dbl>
 1 6      160    110  3.9   2.62
 2 6      160    110  3.9   2.88
 3 4      108     93  3.85  2.32
 4 6      258    110  3.08  3.22
 5 8      360    175  3.15  3.44
 6 6      225    105  2.76  3.46
 7 8      360    245  3.21  3.57
 8 4      147.    62  3.69  3.19
 9 4      141.    95  3.92  3.15
10 6      168.   123  3.92  3.44
# i 22 more rows
```

- `select(cyl:wt)`: This code selects all columns in the `tb` dataframe starting from `cyl` up to and including `wt`.
- Only the five columns {cyl, disp, hp, drat, wt} get selected. This is a particularly useful feature when dealing with dataframes that have a large number of columns, and we are interested in a contiguous subset of those columns

6. Alternately, suppose instead that we wanted to select all columns except those within the range of `cyl` and `wt`.

```
tb %>%
  select(-cyl:wt)
```

```
# A tibble: 32 x 6
     mpg cyl    disp    hp  drat    wt
   <dbl> <fct> <dbl> <dbl> <dbl> <dbl>
 1  21   6      160    110  3.9   2.62
 2  21   6      160    110  3.9   2.88
 3  22.8 4      108     93  3.85  2.32
 4  21.4 6      258    110  3.08  3.22
 5  18.7 8      360    175  3.15  3.44
 6  18.1 6      225    105  2.76  3.46
 7  14.3 8      360    245  3.21  3.57
 8  24.4 4      147.    62  3.69  3.19
```

```
 9  22.8 4      141.    95  3.92  3.15
10  19.2 6      168.   123  3.92  3.44
# i 22 more rows
```

`select(-cyl:wt)`: The `-` sign preceding the cyl:wt range denotes exclusion. Consequently, this selects all columns in the `tb` dataframe, excluding those from `cyl` to `wt` inclusive.

7. **`arrange()`:** Recall that this function is used to reorder rows in a tibble by one or more variables. By default, it arranges rows in ascending order.

- Suppose we want to select only the `mpg` and `hp` columns, where `hp>200` and we want to sort the result in descending order of `mpg`.

```
tb %>%
  select(mpg, hp) %>%
  filter(hp>200) %>%
  arrange(desc(mpg))
```

```
# A tibble: 7 x 2
    mpg    hp
  <dbl> <dbl>
1  15.8   264
2  15     335
3  14.7   230
4  14.3   245
5  13.3   245
6  10.4   205
7  10.4   215
```

`tb %>% select(mpg, hp)`: The select function is used here to extract only the `mpg` and `hp` columns from the `tb` dataframe.

`filter(hp>200)`: The filter function is used to extract the cars where horsepower `hp>200`.

`arrange(desc(mpg))`: The arrange function is then used to order the rows in descending order (`desc`) based on the `mpg` column.

8. **Benefit from using %>%**: Suppose we wanted to subset the data as follows.

- Select cars with 6 cylinders (`cyl` == 6).
- Choose only the `mpg` (miles per gallon), `hp` (horsepower) and `wt` (weight) columns.
- Arrange in descending order by `mpg`.

Without using the pipe operator, we would have to nest the operations, as follows:

```
arrange(select(filter(tb, cyl == 6), mpg, hp, wt), desc(mpg))
```

```
# A tibble: 7 x 3
    mpg    hp    wt
  <dbl> <dbl> <dbl>
1  21.4   110  3.22
2  21     110  2.62
3  21     110  2.88
4  19.7   175  2.77
5  19.2   123  3.44
6  18.1   105  3.46
7  17.8   123  3.44
```

- Using the pipe operator, we could write the code more efficiently as follows:

```
tb %>%
  filter(cyl == 6) %>%
  select(mpg, hp, wt) %>%
  arrange(desc(mpg))
```

```
# A tibble: 7 x 3
    mpg    hp    wt
  <dbl> <dbl> <dbl>
1  21.4   110  3.22
2  21     110  2.62
3  21     110  2.88
4  19.7   175  2.77
5  19.2   123  3.44
6  18.1   105  3.46
7  17.8   123  3.44
```

- This way, the pipe operator makes the code more readable and the sequence of operations is easier to follow.

9. **`mutate()`:** Recall that this function is used to create new variables (columns) or modify existing ones.

- Suppose we want to create a new column named `efficiency`, defined as the ratio of `mpg` to `hp`.

```
mutated_data <- tb %>%
  mutate(efficiency = mpg / hp) %>%
  select(mpg, hp, efficiency, cyl, am) %>%
  filter(mpg>30)

mutated_data
```

```
# A tibble: 4 x 5
    mpg    hp efficiency cyl   am
  <dbl> <dbl>      <dbl> <fct> <fct>
1  32.4    66      0.491 4     1
2  30.4    52      0.585 4     1
3  33.9    65      0.522 4     1
4  30.4   113      0.269 4     1
```

- The resulting tibble contains a new column `efficiency`, which is the ratio of `mpg` to `hp`.
- Note that `mutate()` does not modify the original dataset, but creates a new object `mutated_data` with the results. If we want to modify the original dataset, we would need to save the result back to the original variable.

`transmute()`: This is a variation of **`mutate()`** which does not retain the old columns.

```
t1 <- tb %>%
  filter(mpg>30) %>%
  transmute(efficiency = mpg / hp)

t1
```

```
# A tibble: 4 x 1
  efficiency
       <dbl>
1      0.491
2      0.585
3      0.522
4      0.269
```

- Notice that `transmute()` retains only the newly created column.

10. **`summarise()`:**: Recall that this function is used to create summaries of data. It collapses a tibble to a single row. Suppose we want to calculate the mean of `mpg` in the `mtcars` dataset

```
tb %>%
  summarise(Mean_mpg = mean(mpg))
```

```
# A tibble: 1 x 1
  Mean_mpg
     <dbl>
1     20.1
```

- This code creates a tibble that contains a single row having the mean value of `mpg`.
- We can extend this code to print both the mean and standard deviation, by slightly extending the above code, as follows.

```
tb %>%
  summarise(Mean_mpg = mean(mpg),
            SD_mpg = sd(mpg)
            )
```

```
# A tibble: 1 x 2
  Mean_mpg SD_mpg
     <dbl>  <dbl>
1     20.1   6.03
```

11. To include additional statistical measures such as median, quartiles, minimum, and maximum in our summary data, we can use respective R functions within the `summarise()` function.

```
summary_data <- tb %>% summarise(
  N = n(),
  Mean = mean(mpg),
  SD = sd(mpg),
  Median = median(mpg),
  Q1 = quantile(mpg, 0.25),
  Q3 = quantile(mpg, 0.75),
  Min = min(mpg),
  Max = max(mpg)
)
summary_data
```

```
# A tibble: 1 x 8
      N  Mean    SD Median    Q1    Q3   Min   Max
  <int> <dbl> <dbl>  <dbl> <dbl> <dbl> <dbl> <dbl>
1    32  20.1  6.03   19.2  15.4  22.8  10.4  33.9
```

- We could convert this back into a standard dataframe and display it in a better formatted manner up to two decimal places, with the following code. [2] [3]

```
summary_data %>%
  as.data.frame() %>%
  round(2)
```

```
   N  Mean   SD Median    Q1   Q3  Min  Max
1 32 20.09 6.03   19.2 15.43 22.8 10.4 33.9
```

6.5 Additional functions in the dplyr package

1. **`rename()`:** The `rename()` function is utilized whenever we need to modify the names of some variables in our dataset. Without changing the structure of the original dataset, it allows us to give new names to chosen columns.
2. **`group_by()`:** The `group_by()` function comes into play when we need to implement operations on individual groups within our data. By categorizing our data based on one or multiple variables, we are able to apply distinct functions to each group separately.
3. **`slice()`:** To select rows by their indices, we use the `slice()` function. This is especially handy when we need specific rows, for example, the first 10 or last 10 rows, depending on a defined order.
4. **`transmute()`:** When we want to generate new variables from existing ones and keep only these new variables, we use the `transmute()` function. It is similar to `mutate()`, but it only keeps the newly created variables, making it a powerful tool when we're only interested in transformed or calculated variables.
5. **`pull()`:** The `pull()` function is used to extract a single variable as a vector from a dataframe. This function becomes very practical when we wish to isolate and work with a single variable outside its dataframe.
6. **`n_distinct()`:** To enumerate the unique values in a column or vector, we use the `n_distinct()` function. It's an essential function when we want to know the number of distinct elements within a specific categorical variable. [2] [3]

6.5.1 Using `dplyr` to explore the `mtcars` tibble more

1. **`rename()`:** Remember that this function is helpful in changing column names in our data. For instance, let us modify the name of the `mpg` column to `MPG` in the mtcars dataset.

```
tb %>%
  rename(MPG = mpg)
```

```
# A tibble: 32 x 11
     MPG cyl    disp    hp  drat    wt  qsec vs    am    gear   carb
   <dbl> <fct> <dbl> <dbl> <dbl> <dbl> <dbl> <fct> <fct> <fct> <dbl>
 1  21   6      160    110  3.9   2.62  16.5 0     1     4         4
 2  21   6      160    110  3.9   2.88  17.0 0     1     4         4
 3  22.8 4      108     93  3.85  2.32  18.6 1     1     4         1
 4  21.4 6      258    110  3.08  3.22  19.4 1     0     3         1
 5  18.7 8      360    175  3.15  3.44  17.0 0     0     3         2
 6  18.1 6      225    105  2.76  3.46  20.2 1     0     3         1
 7  14.3 8      360    245  3.21  3.57  15.8 0     0     3         4
 8  24.4 4      147.    62  3.69  3.19  20   1     0     4         2
 9  22.8 4      141.    95  3.92  3.15  22.9 1     0     4         2
10  19.2 6      168.   123  3.92  3.44  18.3 1     0     4         4
# i 22 more rows
```

2. **`group_by()`:** This function is key for performing operations within distinct groups of our data.

*For example, let us group the dataset by `cyl` (number of cylinders) and `am` transmission and find the mean and standard deviation for each sub-group.

```
tb %>%
  group_by(cyl, am) %>%
  summarize(MeanMPG = mean(mpg),
            StdDevMPG = sd(mpg))
```

```
`summarise()` has grouped output by 'cyl'. You can override using the `.groups`
argument.
```

```
# A tibble: 6 x 4
# Groups:   cyl [3]
  cyl   am    MeanMPG StdDevMPG
  <fct> <fct>   <dbl>     <dbl>
```

```
1 4      0        22.9     1.45
2 4      1        28.1     4.48
3 6      0        19.1     1.63
4 6      1        20.6     0.751
5 8      0        15.0     2.77
6 8      1        15.4     0.566
```

3. **`slice()`:** Recall that this function is beneficial when we wish to choose rows based on their positions. For example, let us slice from row 3 to row 7 of the dataset.

```
tb %>%
  slice(3:7)
```

```
# A tibble: 5 x 11
    mpg cyl    disp    hp  drat    wt  qsec vs    am    gear   carb
  <dbl> <fct> <dbl> <dbl> <dbl> <dbl> <dbl> <fct> <fct> <fct> <dbl>
1  22.8 4       108    93  3.85  2.32  18.6 1     1     4         1
2  21.4 6       258   110  3.08  3.22  19.4 1     0     3         1
3  18.7 8       360   175  3.15  3.44  17.0 0     0     3         2
4  18.1 6       225   105  2.76  3.46  20.2 1     0     3         1
5  14.3 8       360   245  3.21  3.57  15.8 0     0     3         4
```

In this `sliced_data` tibble, only the first three rows from the mtcars dataset are included.

4. **`pull()`**: Recall that this function is employed to remove a single variable from a dataframe as a vector. Let us isolate the `mpg` (miles per gallon) variable from the dataset.

```
tb %>%
  pull(mpg)
```

```
 [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2 10.4
[16] 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4 15.8 19.7
[31] 15.0 21.4
```

5. **`n_distinct()`**: Recall that this function is used to count the distinct values in a column or vector. Let us count the number of distinct values in the`cyl`(cylinders) column from the dataset.

```
tb %>%
  summarise(countDistinctCyl = n_distinct(cyl))
```

```
# A tibble: 1 x 1
  countDistinctCyl
             <int>
1                3
```

- This code shows the number of unique levels in the `cyl` column of the dataset.

6.6 Summary of Chapter 6 – Exploring Dataframes (Part 2 of 2)

This chapter provided an overview of the `tibble` data structure and the `dplyr` package in the R programming language. We started with an introduction to `tibble`, a data structure in R that is an updated version of data frames with enhanced features for flexible and effective data management. These benefits include more user-friendly printing, reliable subsetting behavior, transparent handling of data types, and support for non-syntactic column names.

Subsequently, we shifted focus to the `dplyr` package, which is a powerful tool for data manipulation in R. This package offers a cohesive set of functions, often referred to as "verbs", which allow for efficient and straightforward manipulation of data. The key "verbs" in `dplyr`—`filter()`, `select()`, `arrange()`, `mutate()`, and `summarise()`— have been explained and illustrated with examples. An integral component of the `dplyr` package, the pipe operator `%>%`, was also discussed. This operator allows for a more readable and understandable chaining of multiple operations in R, leading to cleaner and more straightforward code.

The chapter gives a comprehensive illustration of using `dplyr` on the `mtcars` dataset. This practical demonstration has involved applying `dplyr` functions to a dataset and explaining the process and results. In addition to the basics, the chapter has also touched upon additional `dplyr` functions such as `rename()`, `group_by()`, and `slice()`, enriching readers' understanding and competency in data manipulation using R.

Overall, this chapter has provided an in-depth understanding of `tibbles` and `dplyr`, their applications, and their importance in data manipulation and management in the R programming environment.

6.7 References

[1] Müller, K., & Wickham, H. (2021). tibble: Simple Data Frames. R Package Version 3.1.3. Retrieved from https://CRAN.R-project.org/package=tibble

[2] Wickham, H., François, R., Henry, L., & Müller, K. (2021). dplyr: A Grammar of Data Manipulation. R Package Version 1.0.7. Retrieved from https://CRAN.R-project.org/package=dplyr

Wickham, H. (2014). Tidy Data. Journal of Statistical Software, 59(10), 1-23. Retrieved from https://www.jstatsoft.org/article/view/v059i10

Grolemund, G., & Wickham, H. (2017). R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. O'Reilly Media, Inc.

[3] Wickham, H. (2016). ggplot2: Elegant Graphics for Data Analysis. Springer-Verlag New York. Retrieved from https://ggplot2.tidyverse.org

Wickham, H., & Grolemund, G. (2016). R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. O'Reilly Media.

Wickham, H. (2020). ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics (Version 3.3.2) [Computer Software]. Retrieved from https://CRAN.R-project.org/package=ggplot2

Wickham, H., et al. (2020). dplyr: A Grammar of Data Manipulation (Version 1.0.2) [Computer Software]. Retrieved from https://CRAN.R-project.org/package=dplyr

Wilkinson, L. (2005). The Grammar of Graphics (2nd ed.). Springer-Verlag.

Wickham, H., et al. (2020). tibble: Simple Data Frames (Version 3.0.3) [Computer Software]. Retrieved from https://CRAN.R-project.org/package=tibble

7 Univariate Categorical data

Chapter 7.

7.1 Exploring Univariate Categorical Data

This chapter explores how to summarize and visualize *univariate, categorical* data.

1. Categorical data is a type of data that can be divided into categories or groups. labels or categorical codes like "male" and "female," "red," "green," and "blue," or "A," "B," and "C" are frequently used to describe category data. This data type is commonly employed in research and statistics where we can categorize the data under various headings depending on their features, characteristics, or any other parameters [1]
2. There are several typical examples of categorical data.:
   - Gender (male, female)
   - Marital status (married, single, divorced)
   - Education level (high school, college, graduate school)
   - Occupation (teacher, doctor, engineer)
   - Hair color (brown, blonde, red, black)
   - Type of car (sedan, SUV, truck) [2]

7.2 Types of Categorical Data – Nominal, Ordinal Data

It is important to understand nominal and ordinal data.

1. '**Nominal data**': This data type is characterized by variables that have two or more categories without any kind of order or priority. This name-based categorization means that nominal data represents whether a variable belongs to a certain category or not, but it does not convey any qualitative value about the variable itself. For example, 'gender' is a nominal data type, as it categorizes data into 'male' or 'female', and neither category is intrinsically superior to the other. Nominal data is usually represented by text labels or categorical codes. [2]

2. ‘**Ordinal data**‘: Unlike nominal data, ordinal data categories have an **inherent order**. This order, however, does not provide any information about the exact differences between the categories. Common examples of ordinal data include the **Likert scale** in surveys, which ranges from “very dissatisfied” to “very satisfied,” or clothing sizes, which are typically “small,” “medium,” or “large.” While we know that “large” is bigger than “medium” and “medium” is bigger than “small,” we don’t know exactly how much bigger one is compared to the other.

   - Ordinal data is frequently utilised to describe groups or categories that can be rated or sorted, such as educational level {high school, college, graduate school}, or movie reviews {G, PG, PG-13, R, NC-17}. [2]

7.3 Factor variables in R

1. “**Factor**” is a term specifically used in R programming language to handle categorical data. Factors are the data objects which are used to categorize the data and store it as levels. They can store both strings and integers. They are useful in data analysis for statistical modeling. [3]

2. A factor variable has **levels**, which are the distinct categories of the variable. For instance, if we have a factor variable named “color”, the levels might be “red,” “blue,” and “green.” Each level corresponds to a category in the categorical data. Creation of a factor variable in R can be done as follows:

```
color <- c("red", "blue", "green", "red", "green", "yellow")
color_factor <- factor(color)
```

3. In this example, `color_factor` is a factor variable with four levels - “red”, “blue”, “yellow” and “green”.

```
levels(color_factor)
```

```
[1] "blue"   "green"  "red"    "yellow"
```

4. **Data**: Suppose we run the following code to prepare the `mtcars` data for subsequent analysis and save it in a tibble called `tb`.

```r
# Load the required libraries, suppressing annoying startup messages
library(dplyr, quietly = TRUE, warn.conflicts = FALSE)
library(tibble, quietly = TRUE, warn.conflicts = FALSE)

# Read the mtcars dataset into a tibble called tb
data(mtcars)
tb <- as_tibble(mtcars)

# Convert relevant columns into factor variables
tb$cyl <- as.factor(tb$cyl) # cyl = {4,6,8}, number of cylinders
tb$am <- as.factor(tb$am) # am = {0,1}, 0:automatic, 1: manual transmission
tb$vs <- as.factor(tb$vs) # vs = {0,1}, v-shaped engine, 0:no, 1:yes
tb$gear <- as.factor(tb$gear) # gear = {3,4,5}, number of gears

# Directly access the data columns of tb, without tb$mpg
attach(tb)
```

5. The above code reads the `mtcars` data into a tibble named `tb` and transforms certain variables from the tibble into factors (categorical variables), using the `as.factor()` function. Specifically, the `cyl` (cylinders), `vs` (engine shape), `am` (transmission type), and `gear` (number of gears) variables are transformed into factors. This change is useful for subsequent analyses that need to recognize these variables as categorical data, rather than numerical data. [3]

7.4 Analysis of a univariate Factor variable

1. Recall that in the `mtcars` dataset, `cyl` stands for the number of cylinders in the car's engine. By transforming `cyl` into a factor variable, we are recognizing it as a categorical variable, which means we're acknowledging that it consists of several distinct categories or levels. Each category or level corresponds to a specific number of cylinders a car might have. [3]

2. Notice that in order to convert `cyl` into a factor variable in R, we have used the following line of code, which modifies the `cyl` column in the mtcars data frame (or tibble), transforming it from a numerical variable into a factor variable.

```r
tb$cyl <- as.factor(tb$cyl)
```

3. As a univariate factor variable, `cyl` represents a single, categorical characteristic of each car in the dataset: the number of cylinders in the engine. In this context, "univariate" means that we're only considering one variable at a time, without reference to any other variables. [3]

4. In the case of `cyl`, the unique categories (or levels) would correspond to the different numbers of cylinders in the cars' engines. The following R code give us the levels of `cyl`.

```
levels(tb$cyl)
```

```
[1] "4" "6" "8"
```

This code could be alternately written as follows, giving the same result:

```
tb$cyl %>% levels()
```

5. The `levels()` function retrieves the levels of a factor variable. When it is applied to `tb$cyl` in the given context, it provides the unique categories of the `cyl` factor variable from the tb tibble, which are "4", "6", and "8". These values correspond to the distinct number of cylinders in the cars' engines that are represented in the `mtcars` dataset. [3]

7.5 Summarizing a univariate Factor Variable

1. **Frequency Table**: A simple way to summarize categorical or factor data is by using a frequency table. It represents the count i.e. frequency) of each category (level) in the factor variable. Essentially, a frequency table gives us a snapshot of the data distribution by indicating how many data points fall into each category [3]
2. We can create a frequency table using the `table()` function. For example, to create a frequency table for the `cyl` variable in the tb tibble, we could write the following code whose output shows the number of cars that fall into each `cyl` category (4, 6, or 8 cylinders).

```
table(tb$cyl)
```

```

 4  6  8 
11  7 14 
```

This code could be alternately written as follows, giving the same result:

```
tb$cyl %>% table()
```

Discussion:

- `tb$cyl`: This piece of code is specifying the `cyl` column in the `tb` tibble, where `cyl` represents the number of cylinders in the car's engine.
- `%>% table()`: The output of `tb$cyl` (i.e., the `cyl` column of `tb`) is then passed to the `table()` function via the pipe operator `%>%`. The `table()` function creates a frequency table of the `cyl` column, counting the number of occurrences of each level (i.e., each distinct number of cylinders).

3. **Proportions**: In contrast to the `table()` function, which generates a count of each category, the `prop.table()` function gives the proportions of each category. The subsequent code computes the proportions of each unique count of cylinders in the `cyl` column from the `tb` tibble.

```
tb$cyl %>%
  table() %>%
  prop.table()
```

```
.
      4       6       8
0.34375 0.21875 0.43750
```

`%>% prop.table()`: The output of `table()`, a frequency table, is passed to the `prop.table()` function. This function converts the frequency table into a proportion table, showing the proportion of the total for each level rather than the raw count.

- This code could be alternately written as follows, giving the same result.

```
p0 = prop.table(table(cyl))
p0
```

- The `prop.table()` function is used in this example to determine the percentage of each category in the `cyl` variable of the `mtcars` dataset.

4. **Percentages**: Suppose we wanted to express the proportions as percentages, rounded to two decimal places. We could calculate the percentages using the `round()` function, as follows:

```
tb$cyl %>%
  table() %>%
  prop.table() %>%
  `*`(100) %>%
  round(digits = 2) %>%
  paste0("%")
```

```
[1] "34.38%" "21.88%" "43.75%"
```

Discussion:

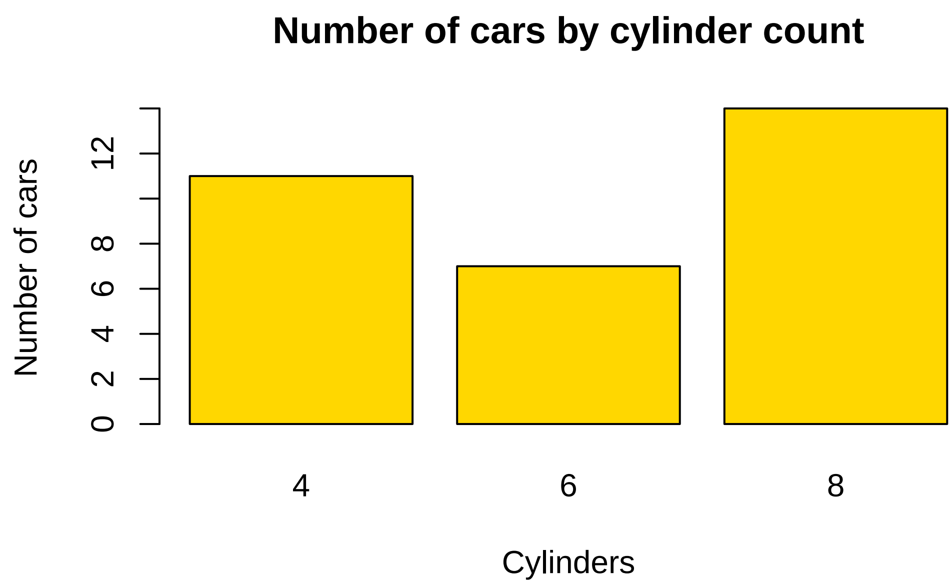
- `%>%*(100)`: Multiplies these proportions by 100, effectively converting them to percentage format.
- `%>% round(digits = 2)`: Rounds these percentages to two decimal places.
- `%>% paste0("%")`: Concatenates a `%` symbol to denote percentages .
- The final output is a table displaying the percentage of cars that have 4, 6, and 8 cylinders, respectively. Each percentage is rounded to two decimal places. The proportion of each category or group of categories, rounded to two decimal places, is presented. For instance, in this data, 43.75% of automobiles have 8 cylinders. [3]

7.6 Visualization of a univariate Factor variable

7.6.1 Barplot

- The `barplot()` function in base R can be used to create a simple bar plot.
- We usually feed it a frequency table, which can be created with the `table()` function. Here's an example of how to create a bar plot for the `cyl` variable:

```
# Create a table of the counts of cars by number of cylinders
cyl_table <- table(tb$cyl)

# Create a barplot of the table
barplot(cyl_table,
        main = "Number of cars by cylinder count",
        xlab = "Cylinders",
        ylab = "Number of cars",
        col = "gold")
```

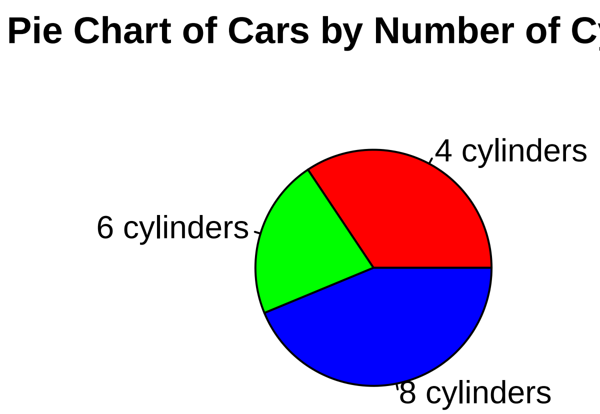


- In this code, `table(tb$cyl)` creates a frequency table for the `cyl` variable. The `xlab`, `ylab`, and `main` arguments are used to set the x-axis label, y-axis label, and the plot title, respectively. The color of the bars can be altered with the `col` option.

7.6.2 Pie chart

- Pie charts can be created with the `pie()` function in base R.
- Just like with `barplot()`, we typically provide a frequency table to `pie()`.
- Here's how to create a pie chart for the `cyl` variable:

```
# Count the number of cars with each number of cylinders
cyl_counts <- table(mtcars$cyl)

# Create a pie chart
pie(cyl_counts,
    main = "Pie Chart of Cars by Number of Cylinders",
    labels = c("4 cylinders", "6 cylinders", "8 cylinders"),
    col = c("red", "green", "blue"))
```

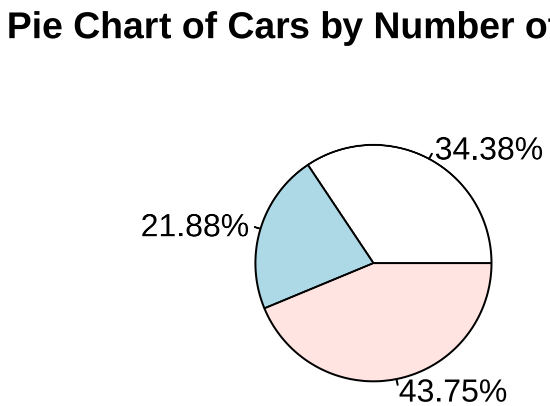


- Alternately, if we wanted to display the percentages, we could write the following code to create a labeled percentage table:

```
# Count the number of cars with each number of cylinders
cyl_counts <- table(mtcars$cyl)

# Calculate the percentages and round them to 2 decimal places
percentages <- round(prop.table(table(tb$cyl))*100, 2)

# Generate labels for the Pie Chart
mylabels <- paste(percentages, "%")

# Alternate code using pipe operator %>%
mylabels2 <-
  tb$cyl %>%
  table() %>%
  prop.table() %>%
  `*`(100) %>%
  round(digits = 2) %>%
  paste0("%")
```

- And then display a Pie Chart:

```
# Create a pie chart showing percentages
pie(cyl_counts,
    main = "Pie Chart of Cars by Number of Cylinders",
    labels = mylabels2)
```



Discussion:

- `percentages <- round(prop.table(table(tb$cyl))*100, 2)`: This line calculates the proportions of each unique number of cylinders, multiplies these proportions by 100 to convert them into percentages, and rounds these percentages to two decimal places. The resulting percentages are saved to the percentages variable.
- `mylabels <- paste(percentages, "%")`: This line creates labels for each slice of the pie chart. These labels are made up of the percentages calculated in the previous step, followed by a percentage sign (%). The labels are saved to a `mylabels` variable.
- The `pie()` function then creates a pie chart using the frequency counts (`cyl_counts`) and the labels (`mylabels`). The main argument is used to set the title of the pie chart.
- While both bar plots and pie charts can be used to visualize the same data, they each have their strengths and limitations. Bar plots are typically better for comparing absolute counts or proportions across levels, whereas pie charts may be more intuitive when demonstrating the part-whole relationship. [4]

7.7 Visualization of a univariate Factor variable using `ggplot2`

7.7.1 Overview of `ggplot2` package

- The `ggplot2` package, part of the tidyverse collection of R packages, is a powerful and flexible tool for creating a wide range of visualizations in R. It is built on the principles of the Grammar of Graphics, which provides a coherent system for describing and building graphs. [5]
- At the heart of `ggplot2` is the idea of mapping data to visual elements. This approach encourages us to think about the relationship between our data and the visual representation, making it easier to create complex and customized graphs.

Key elements of `ggplot2` include:

1. `Data`: The data frame that is to be visualized.
2. `Aesthetics`: These are mappings from data to visual elements (like position, color, and size).
3. `Geoms`: Geometric objects that represent the data (like points, lines, and bars).
4. `Scales`: These control how data values are translated to visual elements.
5. `Facets`: For creating multiple sub-plots, each showing a subset of the data.

In a `ggplot2` graph, these elements are combined in layers, enabling a high degree of customization. [5]

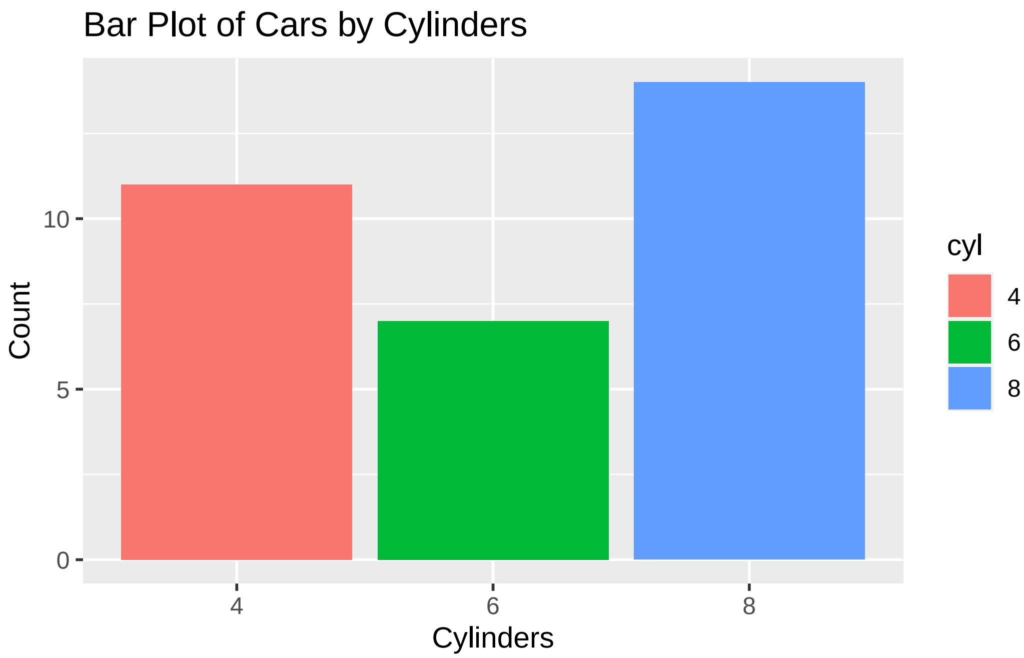
7.7.2 Bar Plot using `ggplot2` package

1. Here is an example of how to use `ggplot2` to create a bar plot of the `cyl` factor:

```
# Load the ggplot2 package for data visualization
library(ggplot2)
```

```
Attaching package: 'ggplot2'


The following object is masked from 'tb':

    mpg
```

```
# Create a bar plot using ggplot
ggplot(data = tb,  # Specify the data frame 'tb'
       aes(x = cyl)) +  # Define 'cyl' as the x-axis variable
  geom_bar(aes(fill = cyl)) +
  # Create a bar graph, 'fill' colors bars based on 'cyl' values
  labs(title = "Bar Plot of Cars by Cylinders",  # Add a title to the plot
       x = "Cylinders",  # Label for the x-axis
       y = "Count")  # Label for the y-axis
```



Discussion:

- `library(ggplot2)`: This line loads the `ggplot2` package into our R session. The ggplot2 package provides a powerful framework for creating diverse and sophisticated graphics in R.

- `ggplot(data = tb, aes(x = cyl)) +`: This line initiates the creation of a ggplot object. The `ggplot()` function takes as arguments the data frame to use (`tb` in this case) and an aesthetics mapping function `aes()`, which maps the `cyl` variable to the x-axis of the plot. The + sign indicates that more layers will be added to this initial ggplot object.

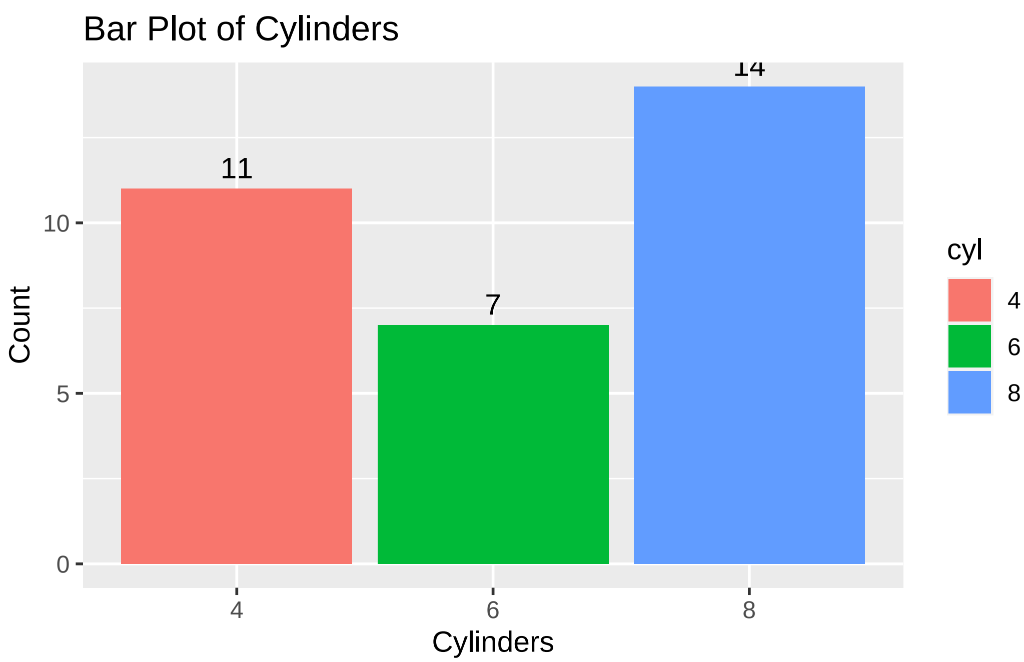
- `geom_bar() +`: This line adds a layer to the ggplot object to create a bar plot. The geom_bar() function by default will create a bar plot where the height of the bars corresponds to the count of each category in the data.

- `labs(title = "Bar Plot of Cylinders", x = "Cylinders", y = "Count")`: This line adds labels to the plot. The `labs()` function is used to set the title of the plot and the labels for the x and y axes. The title of the plot is set as "Bar Plot of Cylinders", the x-axis is labeled as "Cylinders", and the y-axis is labeled as "Count".

- To summarize, this code is loading the `ggplot2` package, initializing a ggplot object with data from the `tb` tibble and mapping the `cyl` variable to the x-axis, adding a bar plot layer to the ggplot object, and finally adding a title and labels to the x and y axes.

2. Suppose we wanted to add labels to the bars showing the count of each category. We can do this by using the `geom_text()` function in ggplot2. Here's how we can modify our code to include labels:

```
ggplot(data = tb, aes(x = cyl)) + # Initialize ggplot
  geom_bar(aes(fill = cyl)) + # Add bar geometries, filling bars
  geom_text(stat='count',    # Add text labels using the 'count' statistic
            aes(label=after_stat(count)), # 'label' to display the count
            vjust=-0.5) + # Adjust vertical position of labels
  labs(title = "Bar Plot of Cylinders", # Add a title to the plot
       x = "Cylinders",  # Label for the x-axis
       y = "Count")      # Label for the y-axis
```



Discussion:

- `geom_text(stat='count', aes(label=after_stat(count)), vjust=-0.5)`: This line adds a text layer to the plot.
- The `stat='count'` argument tells geom_text() to calculate the count of each category, and `aes(label=after_stat(count))` maps these counts to the text labels.
- The `vjust=-0.5` argument adjusts the vertical position of the labels to be slightly above the top of each bar.
- We can also customize the appearance of the text labels using the various arguments to `geom_text()`, such as `size`, `color`, `fontface`, and so on. [5]

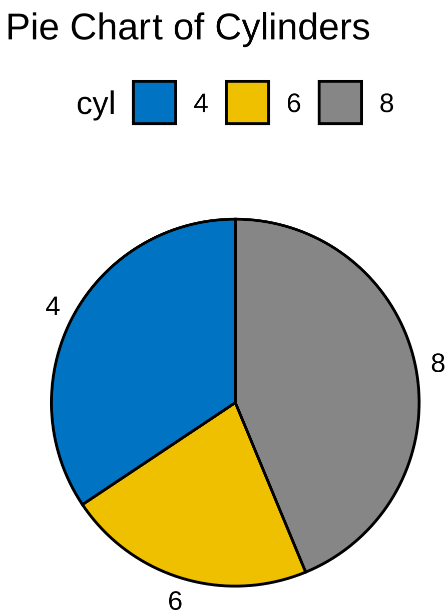
7.7.3 Pie chart using `ggpie`

1. *Aside*: Unfortunately, the `ggplot2` package doesn't directly support pie charts, because they are generally not as effective at displaying data as other types of plots.
2. The `ggpie()` function is part of the `ggpubr` package in R, a package that provides a set of functions to enhance the visual appeal and usability of `ggplot2` plots It specifically enables the creation of pie charts within the `ggplot2` framework. [6]
3. It accepts the following key arguments:

- `data`: The data frame containing the variables to be used in the pie chart.
- `x`: The variable in the data frame that determines the size (area) of the pie slices.
- `fill`: The variable that determines the fill color of the pie slices.
- `label`: The variable to use for the labels of the slices.
- Additional aesthetic details like the title of the chart, color palette, and other ggplot2 functionalities like adding layers, changing themes, etc., can also be utilized with `ggpie()`. [6]

3. Let us use `ggpie()` to create a boxplot of `cyl`.

```r
library(ggpubr)
# Compute counts of each cylinder type
cyl_counts <- as.data.frame(table(tb$cyl))
colnames(cyl_counts) <- c("cyl", "n")

# Create the pie chart
ggpie(data = cyl_counts,
      x = "n",
      fill = "cyl",
      label = "cyl",
      palette = "jco",
      title = "Pie Chart of Cylinders")
```

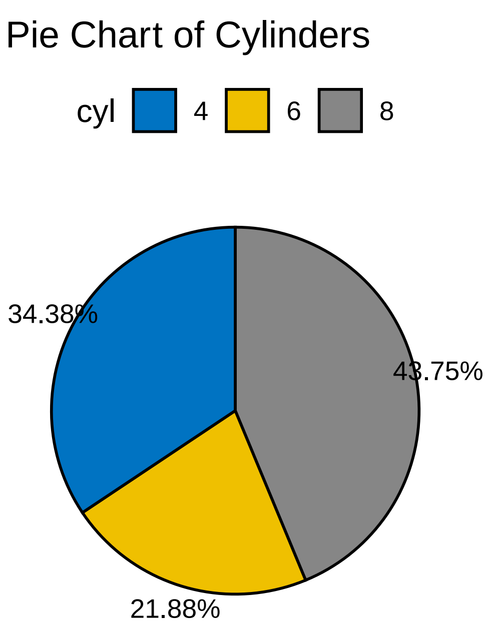


Discussion:

- The `library(ggpubr)` line loads the ggpubr package. This package contains the `ggpie` function which is used to create pie charts in `ggplot2`.
- The command `table(tb$cyl)` calculates the frequency of each type of `cyl` (cylinders) present in the tb dataset. `as.data.frame()` is then used to convert this table into a data frame, which is stored in the `cyl_counts` object. The column names of `cyl_counts` are then set to "cyl" and "n" using the `colnames()` function.
- The `ggpie()` function is used here to generate a pie chart. The arguments to `ggpie()` specify the data frame `(data = cyl_counts)`, the variable that determines the size of the pie slices `(x = "n")`, the variable that determines the fill color of the slices `(fill = "cyl")`, the variable used for the labels of the slices `(label = "cyl")`, the color palette `(palette = "jco")`, and the title of the chart `(title = "Pie Chart of Cylinders")`.
- This code results in a pie chart where each slice represents a different number of cylinders (`cyl`), the size of each slice is determined by the frequency of each number of cylinders (`n`), and the color of each slice is determined by the number of cylinders (`cyl`). The labels for each slice also represent the number of cylinders, and the color palette `jco` is used for the colors of the slices. [6]

4. If we wanted to instead display the percentages for each pie, we could modify this code as follows:

```
library(ggpubr)

# Compute counts and proportions of each cylinder type
cyl_counts <- as.data.frame(table(tb$cyl))
colnames(cyl_counts) <- c("cyl", "n")

# Calculate proportions
cyl_counts$prop <- cyl_counts$n / sum(cyl_counts$n)

# Create labels that display proportions as percentages
cyl_counts$labels <- paste0(round(cyl_counts$prop*100, 2), "%")

# Create the pie chart with proportions
ggpie(data = cyl_counts,
      x = "prop",
      fill = "cyl",
      label = "labels",
      palette = "jco",
      title = "Pie Chart of Cylinders")
```



Discussion:

- This version of the code calculates the proportion of each cylinder type by dividing the count of each type by the total count i.e., `cyl_counts$n / sum(cyl_counts$n)`.

- These proportions are then stored in a new column in the `cyl_counts` data frame.
- Labels are created that display these proportions as percentages, and these labels are used in the pie chart instead of the raw counts.
- The `x` argument in `ggpie()` is updated to use these proportions, so the size of each slice in the pie chart corresponds to the proportion of each cylinder type. [6]

7.8 Summary of Chapter 7 – Univariate Categorical data

The chapter presents an overview of categorical data, focusing on its two main types, nominal and ordinal. It explains how categorical data categorizes information into distinct groups, a key aspect in various analytical fields. Nominal data, which includes categories without a natural order like gender or hair color, and ordinal data, comprising ordered categories such as survey responses or clothing sizes, are thoroughly explored.

Emphasis is placed on the use of factor variables in R for effective management of categorical data, with practical application demonstrated using the `factor()` function. The chapter further delves into analyzing univariate factor variables, employing frequency tables for data summarization and utilizing proportions and percentages for a more nuanced understanding of data distribution.

Visualization techniques using R's `ggplot2` package are introduced, highlighting its capability for creating versatile visualizations, particularly bar plots. The chapter also addresses the limitations of `ggplot2` in creating pie charts and suggests using the `ggpie()` function from the `ggpubr` package as a workaround, allowing for the effective representation of data proportions in a pie chart format.

7.9 References

Categorical Data Analysis:

[1] Diez, D. M., Barr, C. D., & Çetinkaya-Rundel, M. (2012). OpenIntro Statistics (2nd ed.). OpenIntro.

[2] Agresti, A. (2018). An Introduction to Categorical Data Analysis (3rd ed.). Wiley.

Boston University School of Public Health. (2016). Analysis of Categorical Data. Retrieved from https://sphweb.bumc.bu.edu/otlt/MPH-Modules/BS/R/R6_CategoricalDataAnalysis/R6_CategoricalDataAnalysis2.html

Hair, J. F., Black, W. C., Babin, B. J., & Anderson, R. E. (2018). Multivariate Data Analysis (8th ed.). Cengage Learning.

Sheskin, D. J. (2011). Handbook of Parametric and Nonparametric Statistical Procedures (5th ed.). Chapman and Hall/CRC.

Tang, W., Tang, Y., & Song, X. (2012). Applied Categorical and Count Data Analysis. Chapman and Hall/CRC.

Basic R Programming:

[3] Crawley, M. J. (2007). The R Book. Wiley.

Friendly, M., & Meyer, D. (2016). Discrete Data Analysis with R: Visualization and Modeling Techniques for Categorical and Count Data. Chapman and Hall/CRC.

Kabacoff, R. I. (2015). R in Action: Data Analysis and Graphics with R (2nd ed.). Manning Publications.

Data Visualization:

[4] Chang, W. (2018). R Graphics Cookbook: Practical Recipes for Visualizing Data. O'Reilly Media.

Friendly, M. (2000). Visualizing Categorical Data. SAS Institute.

Healy, K., & Lenard, M. T. (2014). A Practical Guide to Creating Better Looking Plots in R. University of Oregon. Retrieved from https://escholarship.org/uc/item/07m6r

Healy, K. (2018). Data Visualization: A Practical Introduction. Princeton University Press.

Heiberger, R. M., & Robbins, N. B. (2014). Design of Diverging Stacked Bar Charts for Likert Scales and Other Applications. Journal of Statistical Software, 57(5), 1-32. doi: 10.18637/jss.v057.i05.

Lander, J. P. (2019). R for Everyone: Advanced Analytics and Graphics (2nd ed.). Addison-Wesley Data & Analytics Series.

Unwin, A. (2015). Graphical Data Analysis with R. CRC Press.

Wilke, C. O. (2019). Fundamentals of Data Visualization. O'Reilly Media.

ggplot2:

[5] Wickham, H. (2016). ggplot2: Elegant Graphics for Data Analysis. Springer-Verlag New York. ISBN 978-3-319-24277-4. Retrieved from https://ggplot2.tidyverse.org.

Wickham, H., & Grolemund, G. (2016). R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. O'Reilly Media.

Wilkinson, L. (2005). The Grammar of Graphics (2nd ed.). Springer-Verlag.

ggpubr:

[6] Kassambara, A. (2023). ggpubr: 'ggplot2' Based Publication Ready Plots. R Package Version 0.6.0. Retrieved from https://rpkgs.datanovia.com/ggpubr/.

8 Bivariate Categorical data (Part 1 of 2)

Chapter 8.

8.1 Exploring Bivariate Categorical Data

This chapter explores how to summarize and visualize *bivariate, categorical* data. It builds on the previous chapter, which taught how to summarize and visualize *univariate, categorical* data.

1. Bivariate categorical data analysis involves examining the **relationship between two categorical variables**. For instance, we might examine the relationship between a person's gender (male, female, or non-binary) and whether they own a car (yes or no). By exploring these two categorical variables together, we can discern potential correlations or associations.

2. As an extension, multivariate analysis involves the simultaneous observation and analysis of more than two variables. We will study multivariate data in the next chapter.

3. **Contingency Table**: A contingency table, also known as a **cross-tabulation** or **crosstab**, is a type of table in a matrix format that displays the frequency distribution of the variables. In the case of a univariate factor variable, a contingency table is essentially the same as a frequency table, as there's only one variable involved. In more complex analyses involving two or more variables, contingency tables provide a way to examine the interactions between the variables. [1]

4. **Data**: Suppose we run the following code to prepare the `mtcars` data for subsequent analysis and save it in a tibble called `tb`.

```
# Load the required libraries, suppressing annoying startup messages
library(dplyr, quietly = TRUE, warn.conflicts = FALSE)
library(tibble, quietly = TRUE, warn.conflicts = FALSE)

# Read the mtcars dataset into a tibble called tb
data(mtcars)
tb <- as_tibble(mtcars)
```

```r
# Convert relevant columns into factor variables
tb$cyl <- as.factor(tb$cyl) # cyl = {4,6,8}, number of cylinders
tb$am <- as.factor(tb$am) # am = {0,1}, 0:automatic, 1: manual transmission
tb$vs <- as.factor(tb$vs) # vs = {0,1}, v-shaped engine, 0:no, 1:yes
tb$gear <- as.factor(tb$gear) # gear = {3,4,5}, number of gears

# Directly access the data columns of tb, without tb$mpg
attach(tb)
```

8.1.1 Frequency Tables for Bivariate Categorical Data

1. As an illustration, let us investigate the bivariate relationship between the number of cylinders (`cyl`) and whether the car has an automatic or manual transmission, (`am=1` for manual, `am=0` for automatic). [2]
2. **`table()`**: We can use this function to generate a contingency table of these two variables.

```r
# Generate a frequency table for the 'cyl' (number of cylinders)
# and 'am' (transmission type) variables from the mtcars dataset.
table(cyl, am)
```

```
   am
cyl  0  1
  4  3  8
  6  4  3
  8 12  2
```

- In this code, a two-way frequency table of `am` and `cyl` is created using the `table()` function. The frequency of each grouping of categories is displayed in the table that results. As an illustration, there are 8 cars with a manual gearbox and 4 cylinders.
- **`addmargins()`**: The `addmargins()` function is used to add row and/or column totals to a table.

```r
# Generate a contingency table for 'cyl' and 'am' from mtcars,
# and add total counts for each row and column.
table(cyl, am) %>%
  addmargins()
```

```
     am
cyl    0  1 Sum
  4    3  8  11
  6    4  3   7
  8   12  2  14
  Sum 19 13  32
```

```r
# Create a contingency table for 'cyl' (cylinders) and '
# am' (transmission type) in mtcars, then use
# addmargins() with margin 1 to add row sums to the table.
table(cyl, am) %>%
  addmargins(1)
```

```
     am
cyl    0  1
  4    3  8
  6    4  3
  8   12  2
  Sum 19 13
```

- In this variation, the `1` in the function call indicates that we want to add **row totals**. So, this command adds the totals (Sum) of each row as a row in the contingency table.

```r
# Generate a contingency table for 'cyl' (cylinders) and 'am' (transmission type)
# in mtcars,and use addmargins(2) to add column sums to the table.
table(cyl, am) %>%
  addmargins(2)
```

```
   am
cyl  0  1 Sum
  4  3  8  11
  6  4  3   7
  8 12  2  14
```

- In this variation, the `2` specifies that we want to add **column totals**. Thus, it adds the totals (Sum) of each column to the contingency table.

3. **`xtabs()`**: This function provides a more versatile way to generate cross tabulations or contingency tables. It differs from the `table()` function by allowing the use of weights and formulas. [2]

   Here's an example of its use:

```r
# Use xtabs to create a cross-tabulation of the number of cylinders ('cyl')
# and transmission type ('am') in the 'tb' dataset.
xtabs(~ cyl + am,
      data = tb)
```

```
   am
cyl  0  1
  4  3  8
  6  4  3
  8 12  2
```

- In the code, we have used `xtabs()` to construct a cross-tabulation of `am` and `cyl`.
- The syntax `~ cyl + am` is interpreted as a formula, signifying that we aim to cross-tabulate these variables. The output is a table akin to what we obtain with `table()`, but with the added advantage of accommodating more intricate analyses.
- An important advantage of `xtabs()` over `table()` is its superior handling of missing values or `NA`; it doesn't automatically exclude them, which is beneficial when dealing with real-world data that often includes missing values. [2]

4. **`ftable()`**: This function is a powerful tool that offers an advanced way to create and display contingency tables. [2]

   Here's an example of its use:

```r
# Create a flat contingency table for the 'cyl' (number of cylinders)
# and 'am' (transmission type) columns in the 'tb' dataset.
ftable(tb$cyl, tb$am)
```

```
     0  1
         
4    3  8
6    4  3
8   12  2
```

- In this code, we've employed the `ftable()` function to create a contingency table of `am` and `cyl`. The output of this function is similar to what we get using `table()`, but it presents the information in a flat, compact layout, which can be particularly helpful especially when dealing with more than two variables.
- One key advantage of `ftable()` is that it creates contingency tables in a more readable format when dealing with more than two categorical variables, making it easier to visualize and understand complex multivariate relationships.

- Like `xtabs()`, `ftable()` also handles missing values or `NA` effectively, making it a reliable choice for real-world data that might contain missing values. [2]

5. We can also use `group_by()` and `summarize()` functions from package `dplyr` to generate contingency tables. [3]

```
# Load the 'dplyr' package
library(dplyr, quietly = TRUE, warn.conflicts = FALSE)

# Group data by the 'cyl' (cylinders) and 'am' (transmission type) columns;
# Then summarise the grouped data to count the frequency in each group
tb %>%
  group_by(cyl, am) %>%
  summarise(Frequency = n())
```

```
`summarise()` has grouped output by 'cyl'. You can override using the `.groups`
argument.
```

```
# A tibble: 6 x 3
# Groups:   cyl [3]
  cyl   am    Frequency
  <fct> <fct>     <int>
1 4     0             3
2 4     1             8
3 6     0             4
4 6     1             3
5 8     0            12
6 8     1             2
```

8.1.2 Proportions Table for Bivariate Categorical Data

6. **`prop.table()`**: This function is an advantageous tool to understand the relative proportions rather than raw frequencies. It converts a contingency table into a table of proportions. [2]

   Here is how we could utilize this function:

```
# Create a frequency table
freq <- table(cyl, am)

# Convert the frequency table to a proportion table
prop <- prop.table(freq)
```

```r
# Round the proportions in the table to three decimal places
round(prop, 3)
```

```
   am
cyl     0     1
  4 0.094 0.250
  6 0.125 0.094
  8 0.375 0.062
```

- In this code, we first generate a frequency table with the `table()` function, using `cyl` and `am` as our variables. Then, we employ the `prop.table()` function to convert this frequency table (`freq_table`) into a proportions table (`prop_table`).
- This resulting `prop_table` reveals the proportion of each combination of `cyl` and `am` categories relative to the total number of observations. This can provide insightful context, allowing us to see how each combination fits into the overall distribution. For instance, we could learn what proportion of cars in our dataset have 4 cylinders and a manual transmission.
- Here is a more efficient method of writing the above code using the pipe operator. [3]

```r
table(cyl, am) %>%
  prop.table() %>%  # Convert the frequency table to a table of proportions
  round(3)          # Round the values to 3 DP
```

```
   am
cyl     0     1
  4 0.094 0.250
  6 0.125 0.094
  8 0.375 0.062
```

7. We can alternately use package `dplyr` to showcase the frequency and proportions in tabular form, instead of a contingency table. [3]

```r
# Load dplyr for data manipulation
library(dplyr)

# Group 'tb' by 'cyl' and 'am', calculate group frequencies,
# and add proportions.
# .groups = "drop" prevents the creation of an additional grouping layer
tb %>%
  group_by(cyl, am) %>%
```

```
  summarise(Frequency = n(),
            .groups = "drop"
            ) %>%
  mutate(Proportion = Frequency / sum(Frequency))
```

```
# A tibble: 6 x 4
  cyl   am    Frequency Proportion
  <fct> <fct>     <int>      <dbl>
1 4     0             3     0.0938
2 4     1             8     0.25
3 6     0             4     0.125
4 6     1             3     0.0938
5 8     0            12     0.375
6 8     1             2     0.0625
```

- In this code, `group_by(cyl, am)` groups the data by `cyl` and `am`, `summarise(Frequency = n())` calculates the frequency for each group, `.groups = "drop"` drops the grouping structure.
- `mutate(Proportion = Frequency / sum(Frequency))` calculates the proportions by dividing each frequency by the total sum of frequencies.
- The `mutate()` function adds a new column to the dataframe, keeping the original data intact.
- **Percentages and Rounding**: If we wanted to display the proportions as percentages, we could round-off the proportion up to 4 decimal places, as follows: [3]

```
# Load the dplyr package for data manipulation tasks
library(dplyr)

# 1. Group by 'cyl' (cylinders) and 'am' (transmission type)
# 2. Summarise each group to count its frequency,
# and drop the group structure
# 3. Add a new column 'Percentage'
tb %>%
  group_by(cyl, am) %>%
  summarise(Frequency = n(), .groups = "drop") %>%
  mutate(Percentage = 100 * round(Frequency / sum(Frequency), 4))
```

```
# A tibble: 6 x 4
  cyl   am    Frequency Percentage
  <fct> <fct>     <int>      <dbl>
```

```
1 4      0              3       9.38
2 4      1              8      25
3 6      0              4      12.5
4 6      1              3       9.38
5 8      0             12      37.5
6 8      1              2       6.25
```

8.1.3 Margins in Proportions Tables

1. Different proportions provide various perspectives on the relationship between categorical variables in our dataset. We can calculate the i) Proportions for **Each Cell**; (ii) **Row-Wise* Proportions; (iii) Column-Wise** Proportions. This forms a crucial part of exploratory data analysis. [2]

2. **Proportions for Each Cell**: This calculates the ratio of each cell to the overall total.

```r
# Generate a table of proportions from the 'cyl' and 'am' variables,
# add total margins, round to 3 decimal places, and convert to percentages.
table(cyl, am) %>%
  prop.table() %>%
  addmargins() %>%
  round(3) %>%
  `*`(100)
```

```
     am
cyl       0     1   Sum
  4     9.4  25.0  34.4
  6    12.5   9.4  21.9
  8    37.5   6.2  43.8
  Sum  59.4  40.6 100.0
```

3. **Row-Wise Proportions**: Here, we compute the proportion of each cell relative to the total of its row.

```r
# Proportion table from 'cyl' and 'am', calculate row-wise proportions,
# add column margins, round to 3 DP, and convert to percentages.
table(cyl, am) %>%
  prop.table(1) %>%  # Row-wise proportions
  addmargins(2) %>%  # Add sums for each column
  round(3) %>%       # Round to three decimal places
  `*`(100)           # Convert to percentage
```

```
   am
cyl     0     1   Sum
  4  27.3  72.7 100.0
  6  57.1  42.9 100.0
  8  85.7  14.3 100.0
```

4. **Column-Wise Proportions**: Here, we determine the proportion of each cell relative to the total of its column.

```
# Proportion table from 'cyl' and 'am', calculate column-wise proportions,
# add row margins, round to 3 DP, and convert to percentages.
table(cyl, am) %>%
  prop.table(2) %>%  # Column-wise proportions
  addmargins(1) %>%  # Add sums for each row
  round(3) %>%       # Round to three decimal places
  `*`(100)           # Convert to percentage
```

```
     am
cyl       0     1
  4    15.8  61.5
  6    21.1  23.1
  8    63.2  15.4
  Sum 100.0 100.0
```

8.2 Visualizing Bivariate Categorical Data

1. **Grouped Barplots** and **Stacked Barplots** serve as powerful tools for representing and understanding bivariate categorical data, where both variables are categorical in nature.

2. Grouped Barplots, often referred to as **side-by-side bar plots**, illustrate the relationship between two categorical variables by placing bars corresponding to one category of a variable next to each other, differentiated by color or pattern. This layout facilitates a direct comparison between categories of the second variable. Grouped bar plots are particularly effective when we are interested in comparing the distribution of a categorical variable across different groups

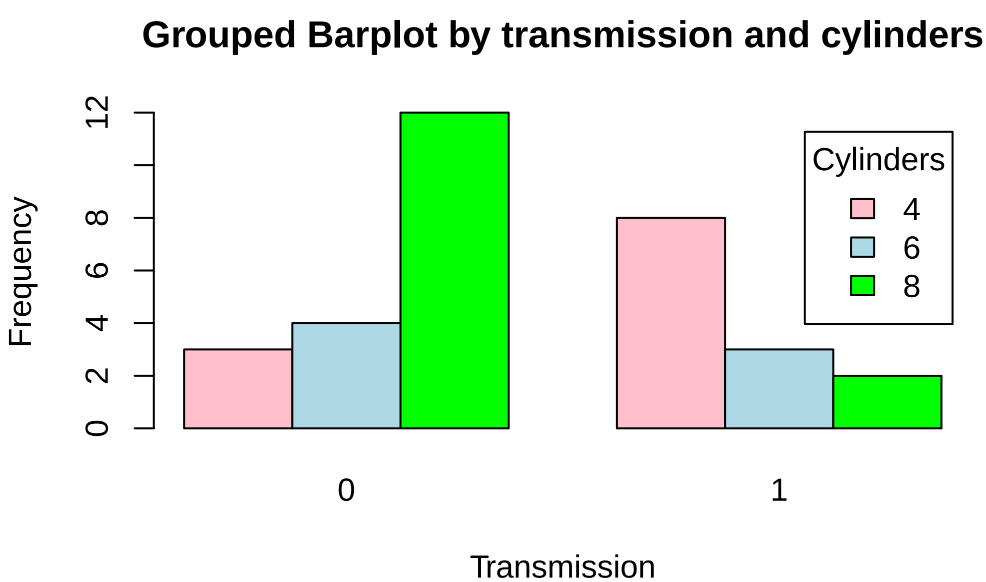
3. On the other hand, **stacked bar plots** present a similar relationship between two categorical variables, but rather than aligning bars side by side, they stack bars on top of one another. This results in a single bar for each category of one variable, with the length of different segments in each bar corresponding to the counts or proportions of the categories of the other variable. Stacked bar plots are advantageous when we're interested in the total size of groups as well as the distribution of a variable across groups. [1]

4. **Grouped Barplot**

```
# Frequency table with counts grouped by the number of cylinders ('cyl')
# and transmission type ('am')
freq <- table(cyl, am)
freq  # Display the created frequency table
```

```
   am
cyl  0  1
  4  3  8
  6  4  3
  8 12  2
```

```
# Generate a grouped bar plot using the frequency table
barplot(freq,
        beside = TRUE,  # Place bars for different groups side by side
        col = c("pink", "lightblue", "green"),  # Assign colors to each group
        xlab = "Transmission",  # Label for the x-axis, transmission type
        ylab = "Frequency",  # Label for the y-axis, the count/frequency
        main = "Grouped Barplot by transmission and cylinders",
        legend.text = rownames(freq),  # Add a legend using row names
        args.legend = list(title = "Cylinders"))  # Set the title
```



5. **Discussion**:

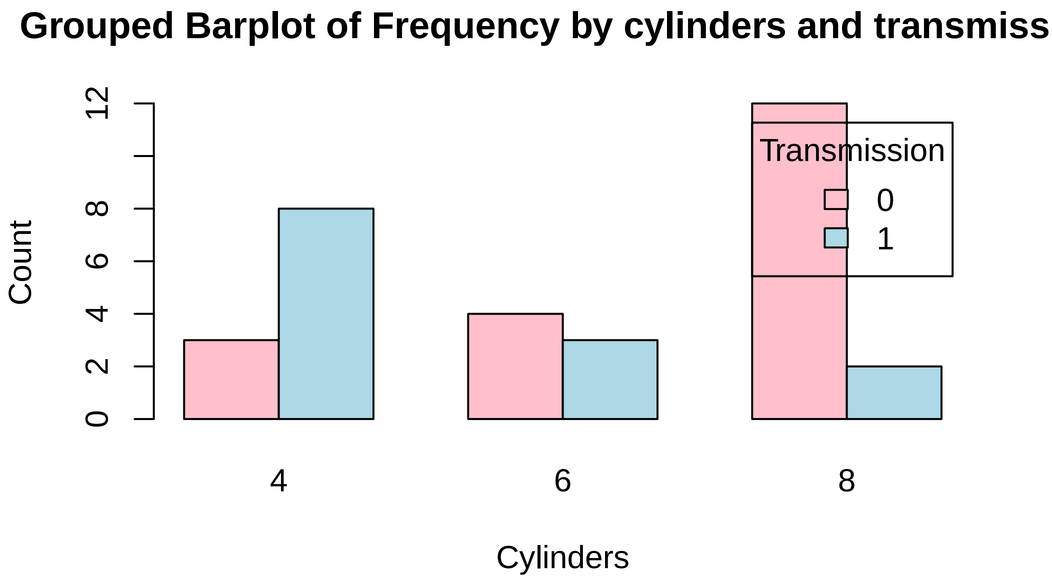
- `freq`: This is the dataset being visualized, which we anticipate to be a contingency table of `am` and `cyl` variables.
- `beside` = TRUE: This argument is specifying that the bars should be positioned next to each other, which means that for each level of `am`, there will be a distinct bar for each level of `cyl`.
- `col = c("pink", "lightblue", "green")`: Here, we are setting the colors of the bars to pink, light blue, and green.
- `xlab = "Transmission"` and `ylab = "Frequency"`: These arguments set the labels for the x and y-axes, respectively.
- `main` = "Grouped Barplot of Frequency by transmission and cylinders": This argument assigns a title to the plot.
- `legend.text = rownames(freq)`: This creates a legend for the plot, using the row names of freq as the legend text.
- `args.legend = list(title = "Cylinders")`: This sets the title of the legend to "Cylinders". [3]

6. Consider this alternate barplot.

```
# Table with counts grouped by transmission ('am') and cylinders ('cyl')
freqInverted <- table(tb$am, tb$cyl)
freqInverted  # Display the created frequency table
```

```
     4  6  8
  0  3  4 12
  1  8  3  2
```

```
# Generate a bar plot using the frequency table
barplot(freqInverted,
        beside = TRUE,  # Place bars for cylinder counts side by side
        col = c("pink", "lightblue"),  # Assign colors
        xlab = "Cylinders",  # Label for the x-axis,
        ylab = "Count",  # Label for the y-axis,
        main = "Grouped Barplot of Frequency by cylinders and transmission",
        legend.text = rownames(freqInverted),  # Add a legend
        args.legend = list(title = "Transmission"))
```



5. **Discussion**: The most significant differences from the previous Grouped Barplot and this one are as follows. [3]

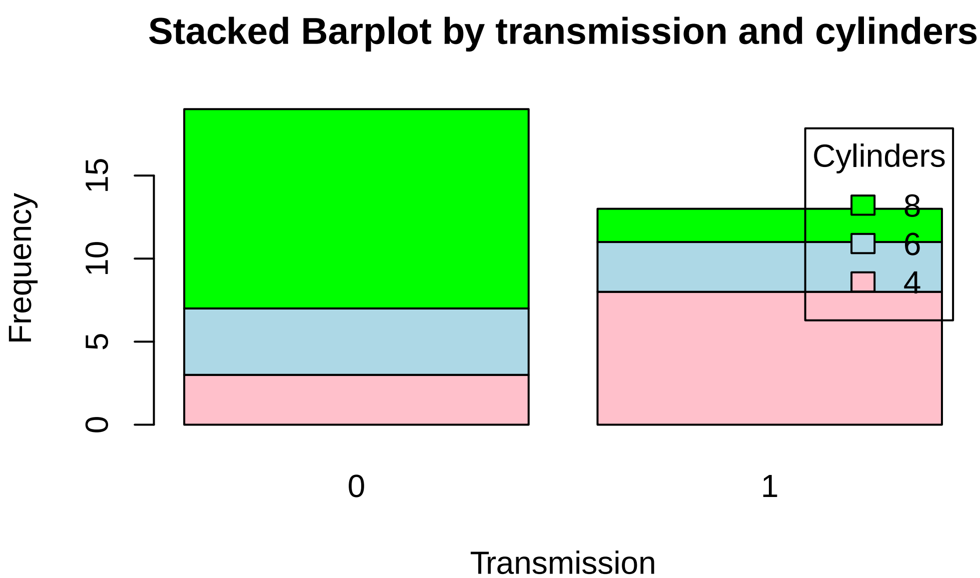
- `freqInverted`: The contingency table's axes have been swapped or inverted. Hence, the table's rows now correspond to the am variable (transmission), and its columns correspond to the `cyl` variable (cylinders).
- `xlab = "Cylinders"` and `ylab = "Count"`: These arguments set the labels for the x and y-axes, respectively. This is a departure from the previous plot where the x-axis represented 'Transmission'. In this case, the x-axis corresponds to 'Cylinders'.
- `legend.text = rownames(freqInverted)` and `args.legend = list(title = "Transmission")`: In the legend, the roles of 'Transmission' and 'Cylinders' are reversed compared to the previous plot.
- To put it succinctly, the main distinction between the two plots is the swapping of the roles of the `cyl` and `am` variables. In the second plot, 'Cylinders' is on the x-axis, which was occupied by 'Transmission' in the first plot. This perspective shift helps to understand the data in a different light, adding another dimension to our exploratory data analysis. [3]

5. **Stacked Barplot**

```
# Generate a frequency table by cylinders ('cyl') and transmission ('am')
freq <- table(tb$cyl, tb$am)
freq  # Display the frequency table
```

```
   0  1
4  3  8
6  4  3
8 12  2
```

```r
# Create a Stacked bar plot using the frequency table
barplot(freq,
        beside = FALSE,  # Bars stacked on top of each other
        col = c("pink", "lightblue", "green"),  # Assign colors
        xlab = "Transmission",  # Label for the x-axis
        ylab = "Frequency",  # Label for the y-axis
        main = "Stacked Barplot by transmission and cylinders",
        legend.text = rownames(freq),  # Add a legend
        args.legend = list(title = "Cylinders"))  # title of the legend
```



There are a few key differences between this Stacked Barplot and the original Grouped Barplot:

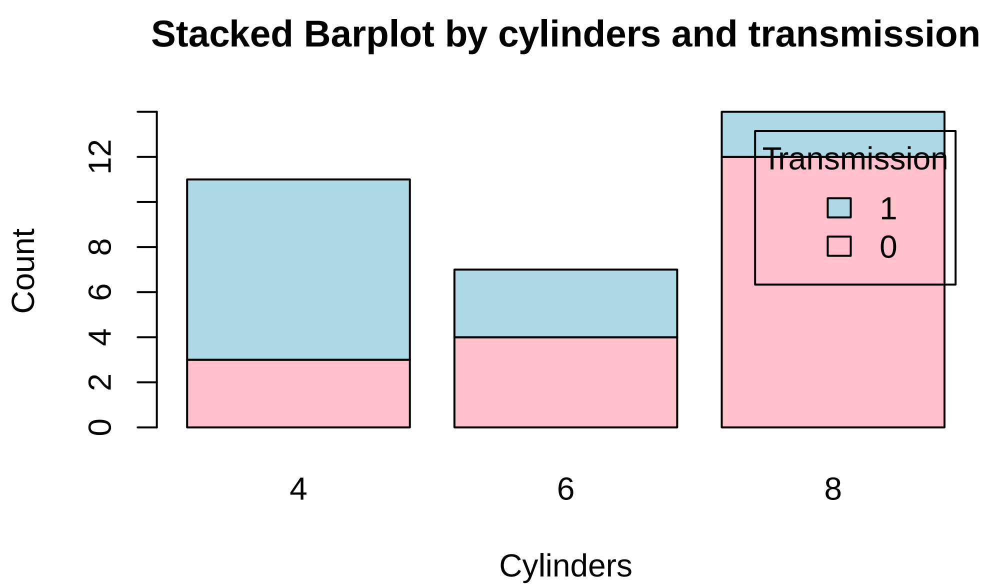
- `beside = FALSE`: In the original code, `beside = TRUE` was used to generate a grouped bar plot, where each set of bars corresponding to each transmission type (automatic or manual) were displayed side by side. However, with `beside = FALSE`, we obtain a stacked bar plot. In this plot, the bars corresponding to each cylinder category (4, 6, or 8 cylinders) are stacked on top of one another for each transmission type.

- `main = "Stacked Barplot of Frequency by transmission and cylinders"`: The title of the plot also reflects this change, mentioning now that it's a stacked bar plot instead of a grouped bar plot.
- Finally, here is a Stacked Barplot, corresponding to the second Grouped Barplot discussed above. [3]

```
# Generate a frequency table by transmission ('am') and cylinders ('cyl')
freqInverted <- table(tb$am, tb$cyl)
freqInverted  # Display the frequency table
```

```
    4  6  8
 0  3  4 12
 1  8  3  2
```

```
# Create a Stacked bar plot using the frequency table
barplot(freqInverted,
        beside = FALSE,  # Configure bars to be stacked
        col = c("pink", "lightblue"),  # Assign colors
        xlab = "Cylinders",  # Set the x-axis label to 'Cylinders'
        ylab = "Count",  # Set the y-axis label to 'Count'
        main = "Stacked Barplot by cylinders and transmission",
        legend.text = rownames(freqInverted),  # Add a legend
        args.legend = list(title = "Transmission"))
```



6. The grouped bar plot helps in comparing the number of cylinders across transmission types side by side, while the stacked bar plot gives an overall comparison in terms of total number of cars, with the frequency of each cylinder type stacked on top of the other. The choice between a stacked and a grouped bar plot would depend on the specific aspects of the data one would want to highlight. Taken together, grouped and stacked bar plots offer visually appealing and intuitive methods for presenting bivariate categorical data, allowing us to understand and analyze relationships between categorical variables in a meaningful way. [3]

8.2.1 Using `ggplot2`

Grouped Barplot using `ggplot2`

- We demonstrate how to create a Grouped Barplot of Car Count by transmission and cylinders

1. Set up the data

```
# Convert the table of cylinder and transmission counts to a data frame
freq <- table(tb$cyl, tb$am)
df <- as.data.frame.table(freq)

# Rename columns for clarity
names(df) <- c("Cylinders", "Transmission", "Frequency")

# Convert 'Cylinders' to a factor for categorical analysis
df$Cylinders <- factor(df$Cylinders)

# Convert 'Transmission' to a factor
# and label the categories as 'Automatic' and 'Manual'
df$Transmission <- factor(df$Transmission,
                          labels = c("Automatic", "Manual"))
```

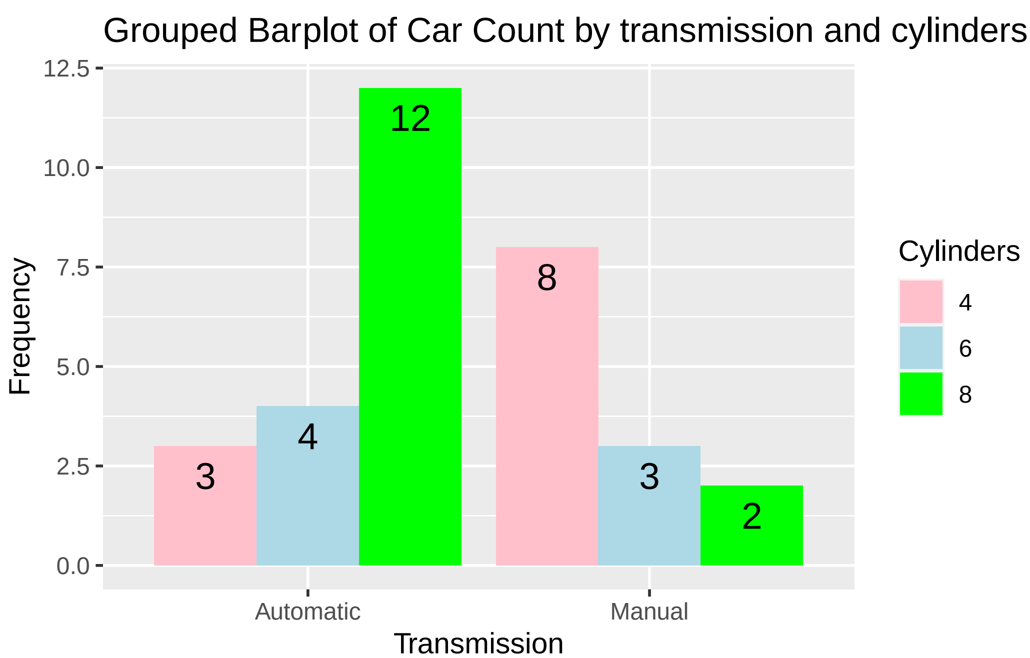
2. Create a Grouped Barplot using `ggplot2` [4]

```
library(ggplot2)
```

```

Attaching package: 'ggplot2'
```

```
The following object is masked from 'tb':

    mpg
```

```r
# Create a Grouped Barplot
ggplot(df,
       aes(x = Transmission,
           y = Frequency,
           fill = Cylinders)) +  # Set up the aesthetics
  geom_bar(stat = "identity",
           # create bars, 'identity' to use direct values from data
           position = position_dodge()) +
           # Dodge position to place bars side by side
  geom_text(aes(label=Frequency),
            # Add text labels on top of bars for frequency values
            vjust=1.6,  # Adjust vertical position of labels
            color="black",  # Set label color
            position = position_dodge(0.9),
            # Dodge position for the labels to align with the bars
            size=5) +  # Set size of the text labels
  labs(x = "Transmission", y = "Frequency",  # Set labels for x and y axes
       fill = "Cylinders",  # Legend title
       title = "Grouped Barplot of Car Count by transmission and cylinders") +
  scale_fill_manual(values = c("pink", "lightblue", "green"))
```



```
# Manually set colors for each cylinder category
```

3. **Discussion**:

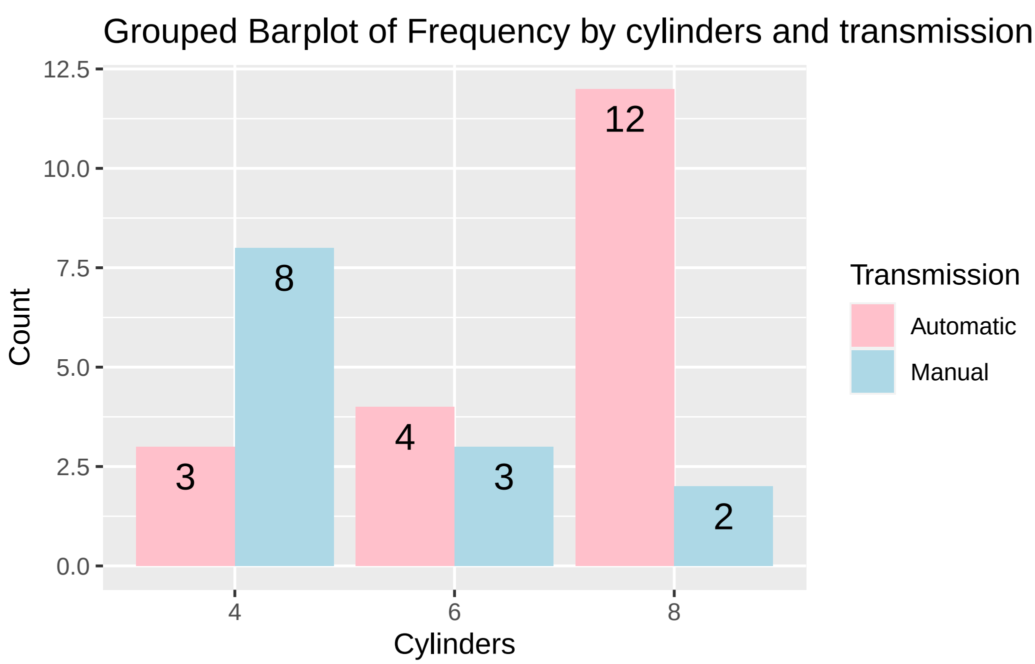
- We first convert the frequency table into a data frame that `ggplot2` can use. This involves converting the table to a data frame and then reshaping it to long format.
- We then use the `ggplot()` function to initiate the plot and specify the aesthetic mappings. Here, we map 'Transmission' to the x-axis, 'Frequency' to the y-axis, and 'Cylinders' to the fill aesthetic which controls the color of the bars.
- `geom_bar()` with `stat = "identity"` is used to add the bar geometry to the plot.
- `position = position_dodge()` ensures the bars are placed side by side, which creates the grouped effect.
- `labs()` is used to add labels to the plot.
- `scale_fill_manual()` allows us to manually specify the colors for the bars.
- We demonstrate how to create a Grouped Barplot of Car Count by cylinders and transmission

4. Set up the data

```
# Frequency table by transmission ('am') and number of cylinders ('cyl')
freqInverted <- table(tb$am, tb$cyl)

# Convert the frequency table to a data frame for better data manipulation
df_inverted <- as.data.frame.table(freqInverted)

# Rename columns to be more descriptive
names(df_inverted) <- c("Transmission", "Cylinders", "Count")

# Convert 'Transmission' to a factor with meaningful labels for clarity
df_inverted$Transmission <- factor(df_inverted$Transmission,
                                   labels = c("Automatic", "Manual"))

# Convert 'Cylinders' to a factor to treat it as a categorical variable
df_inverted$Cylinders <- factor(df_inverted$Cylinders)
```

5. Create a Grouped Barplot using `ggplot2` [4]

```r
# Create the Grouped Barplot using ggplot2
ggplot(df_inverted,
       aes(x = Cylinders,
           y = Count,
           fill = Transmission)) +  # Define aesthetics
  geom_bar(stat = "identity",  # Create bar plot
           position = position_dodge()) +  # Dodge position for grouped bars
  geom_text(aes(label = Count),  # Add text labels (Count) on each bar
            vjust = 1.6,  # Vertically adjust text to sit above the bars
            color = "black",  # Set text color to black for readability
            position = position_dodge(0.9),  # Adjust text position
            size = 5) +  # Set text size
  labs(x = "Cylinders",
       y = "Count",
       fill = "Transmission",  # Label
       title = "Grouped Barplot of Frequency by cylinders and transmission") +
  scale_fill_manual(values = c("pink", "lightblue"))
```




```r
  # Manually set the colors for the 'Transmission' categories
```

6. **Discussion**:

- The `geom_text()` function is used in the same way as in the previous example to add frequency labels to the bars.

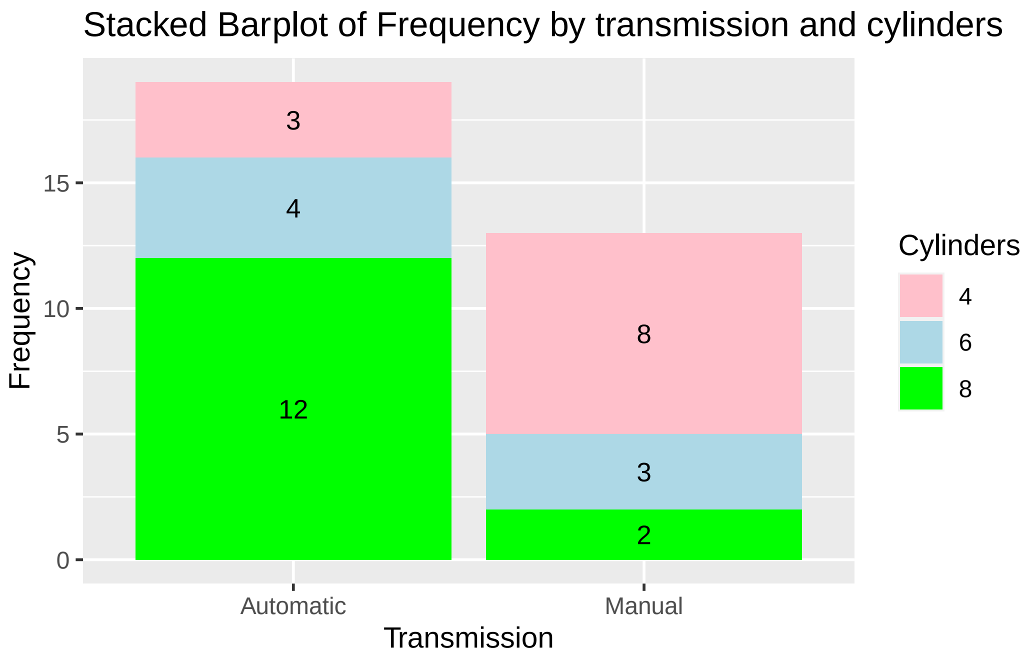
- The `scale_fill_manual()` function is used to specify the colors for the different transmission types.

Stacked Barplots using ggplot2

- We demonstrate how to create a Stacked Barplot of Car Count by cylinders and transmission
- To create a stacked bar plot with `ggplot2`, we apply the `geom_bar()` function but without the dodge positioning this time, since we want the bars stacked. [4]

7. Create a Stacked Barplot using `ggplot2`

```r
library(ggplot2)

# Create the Stacked Barplot using ggplot2
ggplot(df, aes(x = Transmission,
               y = Frequency,
               fill = Cylinders)) +  # Set up plot aesthetics
  geom_bar(stat = "identity") +  # Create stacked bars
  geom_text(aes(label = Frequency),  # Add frequency labels
            position = position_stack(vjust = 0.5),  # Vertically center
            color = "black",  # Set label color to black for contrast
            size = 3.5) +  # Adjust text size for visibility
  labs(x = "Transmission", y = "Frequency",  # Set x and y-axis labels
       fill = "Cylinders",  # Legend title for color fill
       title = "Stacked Barplot of Frequency by transmission and cylinders") +
  scale_fill_manual(values = c("pink", "lightblue", "green"))
```




```
# Manually assign colors to different cylinder counts
```

8. **Discussion**:

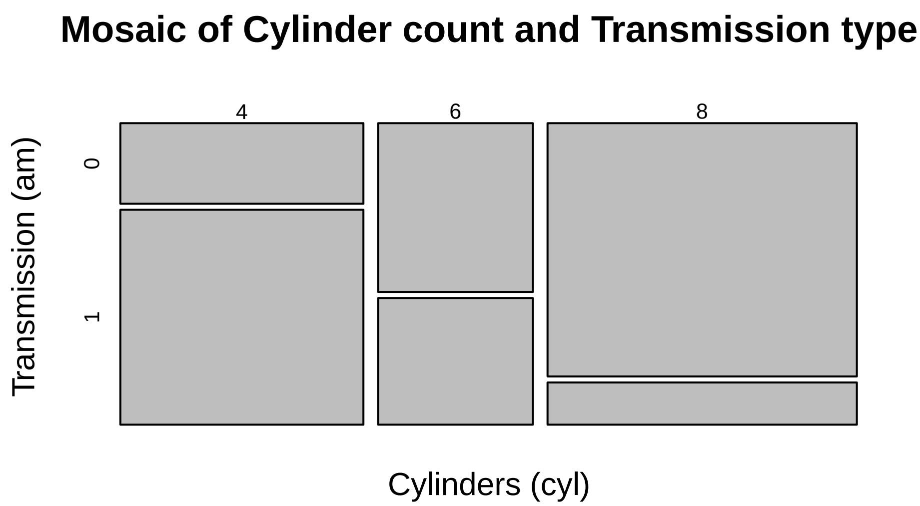
- The `geom_bar(stat = "identity")` is used to create a stacked bar plot.
- The `scale_fill_manual()` function is used to specify the colors for the different cylinder categories. Again, we changed the order of the Transmission and Cylinders columns to suit this specific plot.
- The `geom_text()` function is used to add the labels to the stacked bars. We use position_stack() to position the labels in the middle of the stacked sections.
- The `vjust` argument inside position_stack() controls the vertical positioning of the labels, and 0.5 puts them in the middle.
- As before, `scale_fill_manual()` allows us to specify the colors for the different cylinder categories.[4]

8.2.2 Mosaic Plots for Bivariate Categorical Data

1. A mosaic plot is a graphical method for visualizing data from two or more qualitative variables / categorical data. It is a form of area plot that can provide a visual representation of the frequency or proportion of the different categories within the variables. [5]

2. The following code generates a mosaic plot from a contingency table of two variables: `cyl` (cylinders) and `am` (transmission).

```
# Create a mosaic plot
mosaicplot(table(tb$cyl, tb$am),
           main = "Mosaic of Cylinder count and Transmission type",
           xlab = "Cylinders (cyl)",  # Label for the x-axis
           ylab = "Transmission (am)")  # Label for the y-axis
```



2. In a mosaic plot, we interpret two categorical variables on two axes. [5]

- The **width** of each section on one axis signifies the proportion of that particular category in our dataset.
- Conversely, the **height** of a section on the other axis illustrates the proportion of that category, contingent on the specific category from the first variable.
- Therefore, the **area** of each rectangle directly corresponds to the frequency or proportion of observations falling within that specific combination of categories.

3. By combining both the height and width of the rectangles, a mosaic plot gives us a visual representation of the joint distribution of the two categorical variables. It helps us to identify patterns, associations, and dependencies between the two variables. [5]

4. **Discussion**:

- The `table()` function is used to create a contingency table of the `cyl` and `am` columns of the `tb` tibble. This contingency table represents the counts of all combinations of `cyl` and `am` in the data.

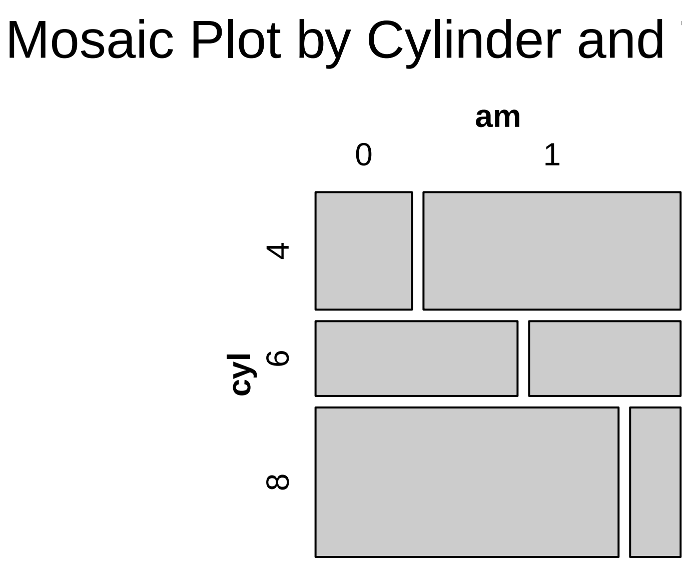
- The `mosaicplot()` function then creates a mosaic plot from this contingency table. The `main`, `xlab`, and `ylab` arguments are used to set the main title, x-axis label, and y-axis label of the plot, respectively.
- This code gives a mosaic plot that visualizes the distribution of the number of cylinders by the type of transmission. Each block's width in the plot would be proportional to the number of cylinders, and the height would be proportional to the transmission type.

5. They can be used to identify breaks in independence or test hypotheses regarding the connections between the variables. They are especially helpful for examining interactions between two or more categorical variables.
6. The `vcd` package, short for Visualizing Categorical Data, provides alternative visual and analytical methods for categorical data. A mosaic plot is created from the variables `cyl` (cylinders) and `am` (transmission type) using the `mosaic()` function. [5]

```
# Load the vcd package for categorical data visualization
library(vcd)
```

```
Loading required package: grid
```

```
# Create a mosaic plot of cylinder count (cyl) vs. transmission (am)
vcd::mosaic(~ cyl + am,
       data = tb,
       main = "Mosaic Plot by Cylinder and Transmission")
```

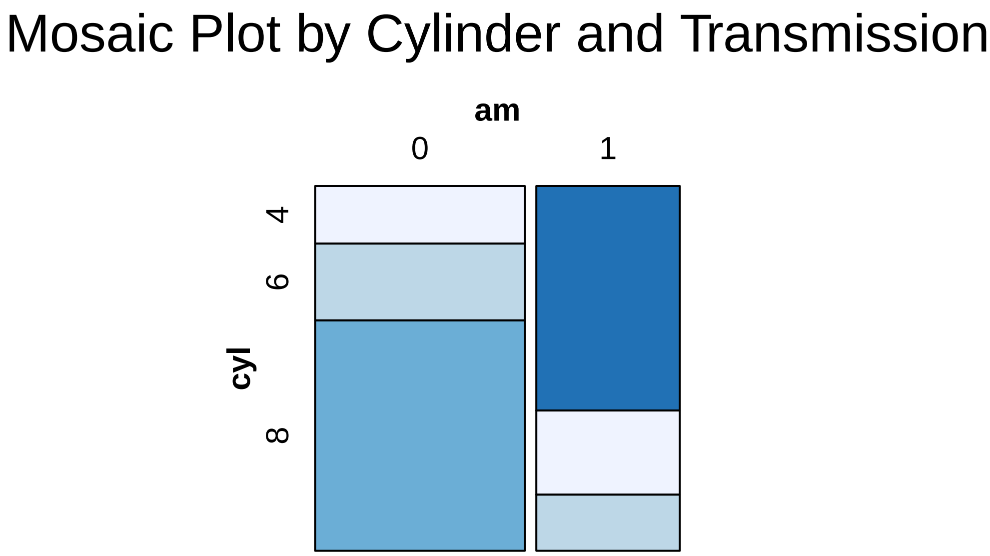


7. **Discussion**:

- `library(vcd)` is used to import the `vcd` package which includes the `mosaic()` function that is superior to the base R `mosaicplot()` in its handling of labeling and legends.
- `mosaic(\~ cyl + am, data = tb, main = "Mosaic of Cylinder count and Transmission type")` creates a mosaic plot.
- In this command, the formula `~ cyl + am` tells R to plot `cyl` against `am`. The argument `data = tb` specifies that the variables for the plot come from the `tb` data frame. The `main` argument sets the title of the plot.

8. We can customize the shading color of the mosaic plot by setting the color using the `col` argument inside the `mosaic()` function. To specify shades of blue, we can use the `RColorBrewer` package and its `brewer.pal()` function to generate a color palette:

```
# Load necessary packages
library(vcd)
library(RColorBrewer)

# Define the color palette using RColorBrewer
cols <- brewer.pal(4, "Blues")
# Select a palette of 4 shades from the 'Blues' spectrum

# Create a mosaic plot
vcd::mosaic(~ cyl + am,
       data = tb,
       main = "Mosaic Plot by Cylinder and Transmission",
       shade = TRUE,  # Apply shading to highlight differences
       highlighting = "cyl",  # Highlight based on cylinder count
       highlighting_fill = cols)  # Use the defined colors
```

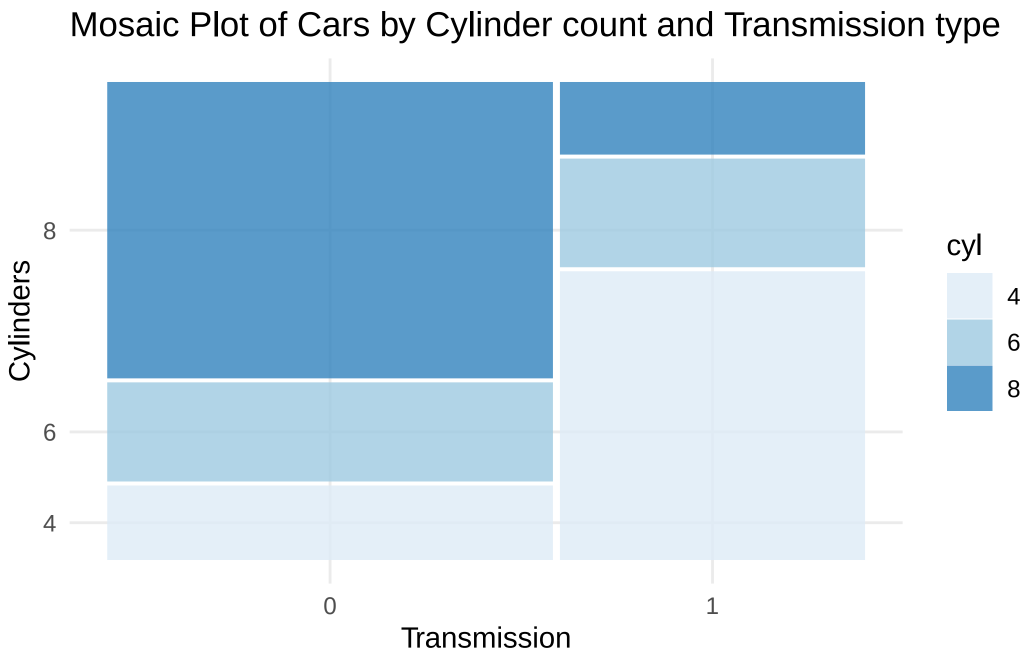


9. **Discussion**:

- `brewer.pal(4, "Blues")` creates a color palette of 4 different shades of blue.
- `highlighting = "cyl"` means that the shading will differentiate the categories in the cyl variable.
- `highlighting_fill = cols` applies the cols color palette to the shading.

10. We can also recreate the previous mosaic plot using `ggplot2` and `ggmosaic` packages. Here is how to do it:

```
# Load the packages
library(ggplot2)
library(ggmosaic, quietly = TRUE, warn.conflicts = FALSE)  # Load ggmosaic

# Create the mosaic plot using ggplot2 and ggmosaic
ggplot(data = tb) +
  geom_mosaic(aes(x = product(cyl, am),
                  fill = cyl)) +
  # Create mosaic plot with 'cyl' and 'am' and fill based on 'cyl'
  theme_minimal() +  # Use a minimal theme for a cleaner look
  labs(title = "Mosaic Plot of Cars by Cylinder count and Transmission type",
       x = "Transmission",  # Label for the x-axis
       y = "Cylinders") +  # Label for the y-axis
  scale_fill_brewer(palette = "Blues")
```




```
# Use a color scale from the 'Blues' palette for the fill
```

11. **Discussion**:

- `geom_mosaic(aes(x = product(cyl, am), fill = cyl))` creates the mosaic plot with `cyl` and `am` as the categorical variables.
- The `fill = cyl` part means that the color fill will differentiate the categories in the `cyl` variable.
- `theme_minimal()` applies a minimal theme to the plot.
- `labs()` is used to specify the labels for the plot, including the `title`, x-axis label, and y-axis label.
- `scale_fill_brewer(palette = "Blues")` specifies the color palette to be used for the fill color, in this case, a palette of blues.

8.3 Summary of Chapter 8 – Bivariate Categorical data (Part 1 of 2)

In this chapter, we delve into an extensive exploration of various methods for visualizing bivariate categorical data using R, which include but are not limited to grouped bar plots, stacked bar plots, and mosaic plots.

An explanation is provided to distinguish between grouped and stacked bar plots, further supported by the inclusion of coding examples and variations in parameters. Both base R functions and the more advanced `ggplot2` library serve as instrumental tools to depict these techniques, furnishing readers with a comprehensive understanding of their practical usage. The `ggplot2` library further elevates the possibilities by offering enhanced customization capabilities and superior control over aesthetics. Detailed coding examples are presented again to explain both the grouped and stacked bar plots within this library.

The chapter also introduces mosaic plots as another way of visualizing bivariate categorical data and discusses how mosaic plots can provide a visual representation of the frequency or proportion of different categories within variables. The use of the base R `mosaicplot()` function is exemplified, followed by the demonstration of the `mosaic()` function in the `vcd` package and the `ggmosaic` package, bringing full circle the spectrum of visualizations covered for bivariate categorical data.

8.4 References

Categorical Data Analysis:

[1] Agresti, A. (2018). An Introduction to Categorical Data Analysis (3rd ed.). Wiley.

Boston University School of Public Health. (2016). Analysis of Categorical Data. Retrieved from https://sphweb.bumc.bu.edu/otlt/MPH-Modules/BS/R/R6_CategoricalDataAnalysis/R6_CategoricalDataAnalysis2.html

Hair, J. F., Black, W. C., Babin, B. J., & Anderson, R. E. (2018). Multivariate Data Analysis (8th ed.). Cengage Learning.

Sheskin, D. J. (2011). Handbook of Parametric and Nonparametric Statistical Procedures (5th ed.). Chapman and Hall/CRC.

Tang, W., Tang, Y., & Song, X. (2012). Applied Categorical and Count Data Analysis. Chapman and Hall/CRC.

Basic R Programming:

[2] Chambers, J. M. (2008). Software for Data Analysis: Programming with R (Vol. 2, No. 1). Springer.

Crawley, M. J. (2012). The R Book. John Wiley & Sons.

Gardener, M. (2012). Beginning R: The Statistical Programming Language. John Wiley & Sons.

Grolemund, G. (2014). Hands-On Programming with R: Write Your Own Functions and Simulations. O'Reilly Media, Inc.

Kabacoff, R. (2022). R in Action: Data Analysis and Graphics with R and Tidyverse. Simon and Schuster.

Peng, R. D. (2016). R Programming for Data Science (pp. 86-181). Leanpub.

R Core Team. (2020). R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing. Retrieved from https://www.R-project.org/.

Tippmann, S. (2015). Programming Tools: Adventures with R. Nature, 517(7532), 109-110.

Wickham, H., Çetinkaya-Rundel, M., & Grolemund, G. (2023). R for Data Science. O'Reilly Media, Inc.

Data Visualization:

[3] Chang, W. (2018). R Graphics Cookbook: Practical Recipes for Visualizing Data. O'Reilly Media.

Friendly, M. (2000). Visualizing Categorical Data. SAS Institute.

Healy, K., & Lenard, M. T. (2014). A Practical Guide to Creating Better Looking Plots in R. University of Oregon. Retrieved from https://escholarship.org/uc/item/07m6r

Healy, K. (2018). Data Visualization: A Practical Introduction. Princeton University Press.

Heiberger, R. M., & Robbins, N. B. (2014). Design of Diverging Stacked Bar Charts for Likert Scales and Other Applications. Journal of Statistical Software, 57(5), 1-32. doi: 10.18637/jss.v057.i05.

Lander, J. P. (2019). R for Everyone: Advanced Analytics and Graphics (2nd ed.). Addison-Wesley Data & Analytics Series.

Unwin, A. (2015). Graphical Data Analysis with R. CRC Press.

Wilke, C. O. (2019). Fundamentals of Data Visualization. O'Reilly Media.

ggplot2:

[4] Wickham, H. (2016). ggplot2: Elegant Graphics for Data Analysis. Springer-Verlag New York. ISBN 978-3-319-24277-4. Retrieved from https://ggplot2.tidyverse.org.

Wickham, H., & Grolemund, G. (2016). R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. O'Reilly Media.

Wilkinson, L. (2005). The Grammar of Graphics (2nd ed.). Springer-Verlag.

Mosaic Plot:

[5] Friendly, M. (1994). Mosaic Displays for Multi-Way Contingency Tables. Journal of the American Statistical Association, 89(425), 190-200.

Hartigan, J. A., & Kleiner, B. (1981). Mosaics for Contingency Tables. In Computer Science and Statistics: Proceedings of the 13th Symposium on the Interface (pp. 268-273).

Meyer, D., Zeileis, A., & Hornik, K. (2020). vcd: Visualizing Categorical Data. R Package Version 1.4-8. Retrieved from https://CRAN.R-project.org/package=vcd

9 Bivariate Categorical data (Part 2 of 2)

Chapter 9.

9.1 Exploring Multivariate Categorical Data

This chapter explores how to summarize and visualize *multivariate, categorical* data.

1. Multivariate categorical variables allow us to analyze relationships between **three or more** categorical variables respectively.
2. This form of analysis can help **reveal complex interactions and dependencies** between multiple variables that cannot be detected in bivariate analyses. Both bivariate and multivariate analyses are essential in statistical and data analysis as they allow us to uncover relationships and patterns in data. [1]
3. **Data**: Suppose we run the following code to prepare the `mtcars` data for subsequent analysis and save it in a tibble called `tb`. [2]

```
# Load the required libraries, suppressing annoying startup messages
library(dplyr, quietly = TRUE, warn.conflicts = FALSE)
library(tibble, quietly = TRUE, warn.conflicts = FALSE)
# Read the mtcars dataset into a tibble called tb
data(mtcars)
tb <- as_tibble(mtcars)

# Convert relevant columns into factor variables
tb$cyl <- as.factor(tb$cyl) # cyl = {4,6,8}, number of cylinders
tb$am <- as.factor(tb$am) # am = {0,1}, 0:automatic, 1: manual transmission
tb$vs <- as.factor(tb$vs) # vs = {0,1}, v-shaped engine, 0:no, 1:yes
tb$gear <- as.factor(tb$gear) # gear = {3,4,5}, number of gears

# Directly access the data columns of tb, without tb$mpg
attach(tb)
```

9.2 Three Way Relationships

1. A Three-Dimensional Contingency Table, often referred to as a **3-way contingency table**, is a statistical tool that helps analyze the relationship between three categorical variables. It builds upon the concept of a standard two-dimensional contingency table, which shows the distribution of two categorical variables, by adding a third dimension to the analysis.

2. Imagine a grid-like structure with three axes representing the three variables. The **rows** correspond to the categories of the first variable, the **columns** represent the categories of the second variable, and the **layers** (sheets) represent the categories of the third variable. Each cell within the table contains the frequency or count of observations that belong to a specific combination of the three variables.

3. When dealing with a three-way relationship, our focus is on three categorical variables and how they **interact** with each other. Such interactions can be manifest in the form of changes in the relationship between two variables based on the values of the third variable. Alternatively, we might seek to comprehend how all three variables collectively influence the observed data.

4. Graphically, three-way relationships can be represented in various forms, such as **three-dimensional contingency tables**, **side-by-side mosaic plots**, or even **three-dimensional bar plots**. However, it's crucial to note that these visual representations can become complicated and challenging to decipher as the number of categories within each variable rises. [1].

9.2.1 Three-Dimensional Contingency Tables

1. The R language, versatile as it is, provides multiple functions for creating contingency tables for multivariate categorical data. In this case, we're focusing on the `table()`, `xtabs()`, and `ftable()` functions for forming a three-way contingency table. [2]

2. We can create a three-way contingency table of `cyl`, `gear`, and `am` using the `table()` function.

```
# Create a contingency table showing the frequencies of car counts
# grouped by cylinder number ('cyl'), number of gears ('gear'),
# and transmission type ('am').
table(cyl,
      gear,
      am)
```

```
, , am = 0

   gear
cyl  3  4  5
  4  1  2  0
  6  2  2  0
  8 12  0  0

, , am = 1

   gear
cyl  3  4  5
  4  0  6  2
  6  0  2  1
  8  0  0  2
```

- When we run this code, the output is a **three-dimensional contingency table** showing the frequencies of all combinations of the three variables. Each cell in the resulting table represents the number of observations for a particular combination of `cyl`, `gear`, and `am` categories.
- Notice that we are segmenting the tables based on the 3rd argument given the table function, which is the transmission `am`.

3. We could alternately run the following code and instead segment the tables based on the cylinders `cyl`. [2]

```
# Create a contingency table with counts of cars grouped by
# transmission ('am'), number of gears ('gear'), and cylinder ('cyl')
table(am,
      gear,
      cyl)
```

```
, , cyl = 4

   gear
am   3  4  5
  0  1  2  0
  1  0  6  2

, , cyl = 6

   gear
```

```
am   3  4  5
  0  2  2  0
  1  0  2  1

, , cyl = 8

   gear
am   3  4  5
  0 12  0  0
  1  0  0  2
```

4. `xtabs()`: We can also create a three-way contingency table of `cyl`, `gear`, and `am` using the `xtabs()` function. [2]

```r
# Use xtabs to create a cross-tabulation of cylinder count ('cyl'),
# gear count ('gear'), and transmission type ('am')
xtabs(~ cyl + gear + am,
      data = tb)
```

```
, , am = 0

   gear
cyl  3  4  5
  4  1  2  0
  6  2  2  0
  8 12  0  0

, , am = 1

   gear
cyl  3  4  5
  4  0  6  2
  6  0  2  1
  8  0  0  2
```

- Here, the formula `~ cyl + gear + am` defines the three variables we are interested in.

5. `ftable()`: The `ftable()` function is employed to generate a 'flat' contingency table, which is essentially a higher-dimensional contingency table displayed in a two-dimensional format. We can also create a three-way contingency table of `gear`, `cyl`, and `am` using the following code:

```r
# Create a flat contingency table using 'ftable' to display the relationship between
# gear count ('gear'), cylinder count ('cyl'), and transmission type ('am')
ftable(gear + cyl ~ am,
       data = tb)
```

```
   gear  3         4         5
   cyl   4  6  8  4  6  8  4  6  8
am
0        1  2 12  2  2  0  0  0  0
1        0  0  0  6  2  0  2  1  2
```

- In this code, `ftable(gear + cyl \~ am, data = tb)`, we arrange the `gear` and `cyl` variables in the rows and the `am` variable in the columns.
- The `~` operator separates the variables that will be displayed in rows (on the left) and columns (on the right) in the resulting table.
- The `+` operator denotes that both `gear` and `cyl` will be included in the row labels. [2]

```r
# Generate a flat contingency table displaying the relationship between
# gear ('gear') and a combination of cylinder ('cyl') and transmission ('am')
ftable(gear ~ cyl + am,
       data = tb)
```

```
       gear  3  4  5
cyl am
4   0        1  2  0
    1        0  6  2
6   0        2  2  0
    1        0  2  1
8   0       12  0  0
    1        0  0  2
```

- This variation of code, `ftable(gear ~ cyl + am, data = tb)`, it is structured slightly differently. Here, the `gear` variable forms the row and both `cyl` and `am` variables form the columns of the flat contingency table.
- In both scenarios, a `ftable` provides a more compact view of the three-way relationship among the `gear`, `cyl`, and `am` variables. However, the orientation of the variables in the rows and columns changes, providing different views of the data and potentially making certain patterns more evident depending on the question we're trying to answer.
- The exact choice between `ftable(gear + cyl ~ am, data = tb)` and `ftable(gear ~ cyl + am, data = tb)` will depend on what specific relationships we are most interested in exploring in our data. [2]

9.2.2 Visualization using a Faceted Bar Plot in `ggplot`

1. A Three-Dimensional Bar Plot is a generalization of a conventional two-dimensional bar graph, expanded into a third dimension. Rather than using bars at specific x coordinates in a two-dimensional plane, we utilize a grid of bars on the x-y plane, extending upwards in the z direction to indicate the data's magnitude. [4]

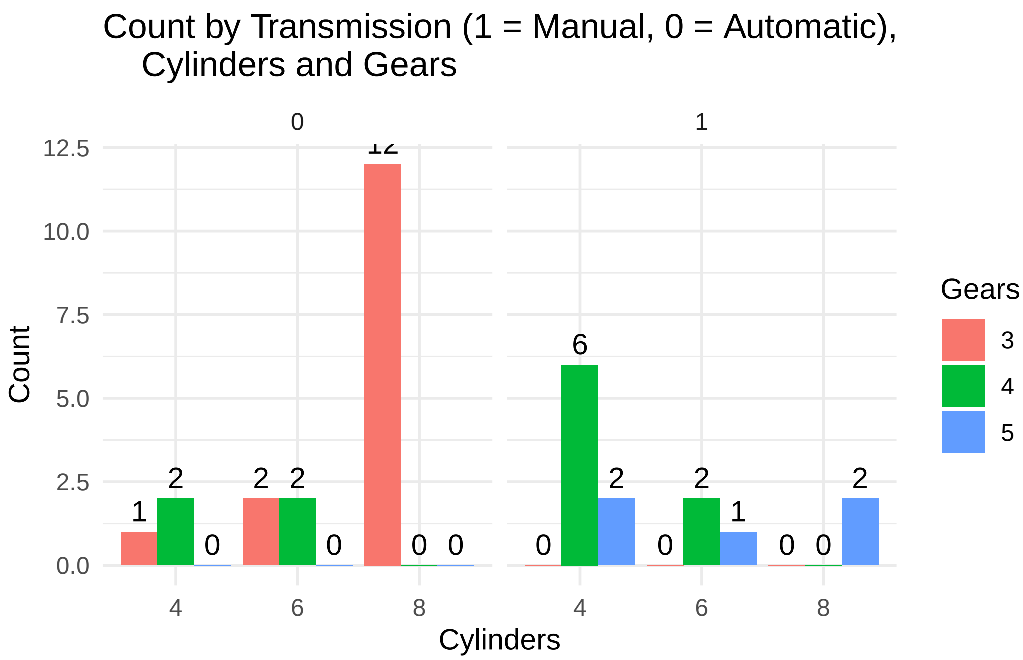
2. Here is some sample code:

```
# Load necessary package
library(ggplot2)
```

```
Attaching package: 'ggplot2'


The following object is masked from 'tb':

    mpg
```

```
# Create a table with count of each combination
count_df <- table(tb$cyl,
                  tb$am,
                  tb$gear)
# Convert table to a data frame for plotting
count_df <- as.data.frame.table(count_df)

# Rename columns for clarity
names(count_df) <- c("cyl", "am", "gear", "count")

# Create the plot using ggplot
ggplot(count_df,
       aes(x=cyl,
           y=count,
           fill=gear)) +
  geom_bar(stat="identity",
           position="dodge") +  # Use dodge position for side-by-side bars
  facet_grid(~am) +  # Create a facet for each transmission type
  labs(
    title = "Count by Transmission (1 = Manual, 0 = Automatic),
    Cylinders and Gears",
    x = "Cylinders",
    y = "Count",
```

```
  fill = "Gears"
) +
# Add count labels to the bars
geom_text(aes(label = count),
          position = position_dodge(width = 0.9),
          vjust = -0.5) +  # Vertically adjust text to sit above bars
theme_minimal()  # Use a minimal theme for a clean look
```



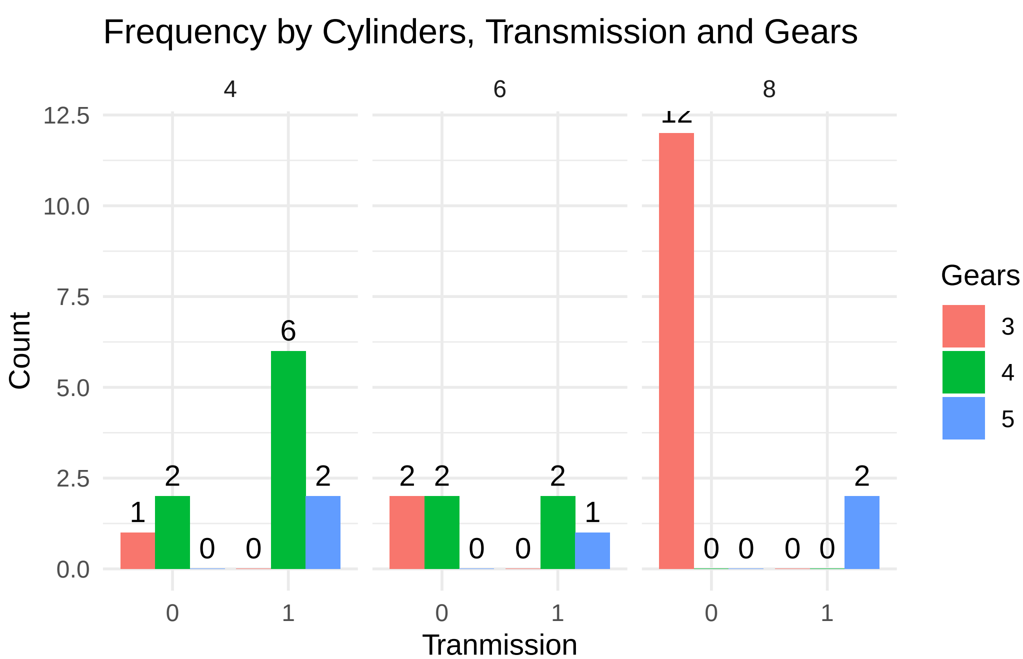
- This code creates a faceted bar plot to visually represent the frequency of combinations of three categorical variables: `cyl` (Cylinders), `am` (Transmission), and `gear` (Gears).
- The line `count_df <- table(tb$cyl, tb$am, tb$gear)` generates a contingency table of the frequencies at each level of the three categorical variables (`cyl`, `am`, `gear`), using the `table()` function.
- Next, `count_df <- as.data.frame.table(count_df)` is used to convert the generated contingency table into a data frame, which can be more conveniently manipulated and visualized using `ggplot2`. [7]
- The names of the data frame's columns are then reassigned using `names(count_df) <- c("cyl", "am", "gear", "count")`. The 'count' column represents the frequency of each combination of the levels of the `cyl`, `am`, and `gear` variables.

- The plot is created using the `ggplot()` function, which initializes a ggplot object. The aesthetic mapping `aes(x=cyl, y=count, fill=gear)` specifies that the x-axis represents `cyl`, the y-axis represents `count`, and the color fill of the bars is based on `gear`.
- The `geom_bar(stat="identity", position="dodge")` function call adds a layer to the plot that depicts the data as a bar chart. The argument `stat="identity"` informs ggplot that the heights of the bars are given in the data (i.e., in the `count` variable), and `position="dodge"` causes bars associated with different levels of `gear` to be drawn side-by-side.
- The `facet_grid(~am)` function call adds facets to the plot based on the `am` variable, creating a separate subplot for each level of `am`.
- The `labs()` function call specifies the labels for the plot, including the title and the x-, y-, and fill-axis labels. The `theme_minimal()` call is used to apply a minimalist aesthetic theme to the plot.
- The `aes(label = count)` inside `geom_text()` tells ggplot to use the `count` column in the data frame as the labels for each bar. The `position_dodge(width = 0.9)` argument ensures that the labels are properly aligned with the corresponding bars in the grouped plot, and `vjust = -0.5` adjusts the vertical position of the labels to make them appear just above the bars.
- This code thus provides a clear and insightful visualization of the frequency of each combination of `cyl`, `am`, and `gear`. [4]

3. Here is some alternate sample code:

```
# Load necessary package
library(ggplot2)

# Create a table with count of each combination
count_df <- table(tb$cyl, tb$am, tb$gear)

# Convert table to a data frame for plotting
count_df <- as.data.frame.table(count_df)

# Rename columns for better understanding
names(count_df) <- c("cyl", "am", "gear", "count")

# Create the plot using ggplot2
ggplot(count_df, aes(x=am, y=count, fill=gear)) +
  geom_bar(stat="identity",
           position="dodge") +
  # Use dodge to display separate bars for each gear type
```

```r
facet_grid(~cyl) +
# Facet by cylinder count for separate plots for each cylinder type
labs(
  title = "Frequency by Cylinders, Transmission and Gears",
  x = "Tranmission",  # X-axis label
  y = "Count",  # Y-axis label
  fill = "Gears"  # Legend title
) +
# Add count labels to each bar for clarity
geom_text(aes(label = count),
          position = position_dodge(width = 0.9),
          vjust = -0.5) +  # Adjust the position of the text labels
theme_minimal()  # Use a minimal theme for a clean appearance
```



- This code is similar to the previous one; both create a faceted bar plot to display the frequencies of the levels of three categorical variables. The main difference between the two lies in the aesthetic mappings and the facet specification in the `ggplot()` function call.
- In this new code, the x-axis mapping in `aes()` is changed from `cyl` (Cylinders) to `am` (Transmission). Hence, the x-axis of the bar plot will now depict the Transmission type instead of Cylinders.

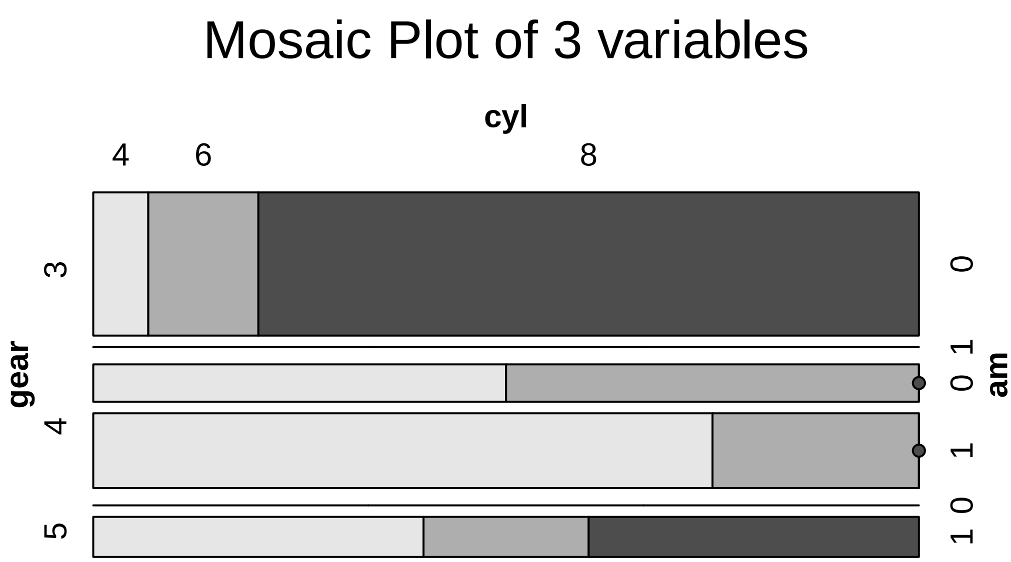
- Similarly, the `facet_grid()` function, which was previously applied to `am`, is now applied to `cyl`. This means that the plot will now be faceted by the Cylinders variable. Each facet (or subplot) will correspond to a different number of Cylinders (4, 6, or 8). [4]

9.2.3 Visualization using a Mosaic Plot

```
# Load necessary packages
library(vcd)
```

```
Loading required package: grid
```

```
# Create a mosaic plot
vcd::mosaic(~gear+cyl+am,
       data = tb,
       main = "Mosaic Plot of 3 variables",  # Set the title
       shade = TRUE,  # Apply shading to highlight differences
       highlighting = "cyl" )  # Highlight based on cylinders
```



1. The provided R code generates a mosaic plot using the `vcd` package, specifically focusing on the `gear`, `cyl`, and `am` variables. A mosaic plot is a visual representation of the frequencies or proportions of combinations of categorical variables.

- `vcd::mosaic(~gear+cyl+am, data = tb, main = "Mosaic of Gears, Cylinders and Transmission type", shade = TRUE, highlighting = "cyl" )` generates the mosaic plot. The `~gear+cyl+am` formula indicates that the mosaic plot should visualize the `gear`, `cyl`, and `am` variables.

- `data = tb` specifies the dataset to be used, which is `tb` in this case.
- `main = "Mosaic of Gears, Cylinders and Transmission type"` sets the main title of the plot.
- `shade = TRUE` means that shading is applied to the cells in the plot. The shading can help to differentiate the cells visually based on the residuals from a model of independence.
- `highlighting = "cyl"` means that the `cyl` variable's levels will be distinctly colored. This highlighting helps to visually emphasize the differences among `cyl` categories in the plot. [5]

2. The generated mosaic plot provides an effective visual exploration of the joint distribution of `gear`, `cyl`, and `am` variables in the `tb` dataset, highlighting the `cyl` variable.

9.3 Four-Way Relationships

- Studying a four-way relationship between four categorical variables, involves looking at their interactions to see how they work together. [1]

9.3.1 Four-Dimensional Contingency Tables

1. Recall that `am`, `cyl`, `gear`, and `vs` are all categorical / factor variables.

- `am` indicates the type of transmission (0 = automatic, 1 = manual).
- `cyl` represents the number of cylinders in the car's engine.
- `gear` is the number of forward gears in the car.
- `vs` indicates the engine shape (0 = V-shaped, 1 = straight).

2. Suppose we wish to create a 4-way Contingency Table. Recall that the `ftable()` function provides an easy way to summarize categorical data in a "flat" table, which can make the data easier to understand and interpret. [2]

```
# Create a flat contingency table with transmission ('am') and
# cylinder ('cyl') against gear ('gear') and engine shape ('vs')
ftable(am + cyl ~ gear + vs,
       data = tb)
```

```
          am   0         1
          cyl  4  6  8  4  6  8
gear vs
3    0         0  0 12  0  0  0
     1         1  2  0  0  0  0
4    0         0  0  0  0  2  0
     1         2  2  0  6  0  0
5    0         0  0  0  1  1  2
     1         0  0  0  1  0  0
```

3. **Discussion**:

- In the formula `am + cyl ~ gear + vs`, the tilde (`~`) separates the rows and columns of the table.
- The variables to the left of the tilde (`am` and `cyl`) form the rows, and the variables to the right of the tilde (`gear` and `vs`) form the columns.
- The plus sign (+) between variables indicates that we want to consider all combinations of these variables.
- The resulting table shows the counts of each combination of `am` and `cyl` for each combination of gear and vs.

4. We can personalize the 4-way contingency table in several ways. Consider this alternate code. [2]

```
# Create a flat contingency table showing the relationship between
# transmission ('am'), cylinder ('cyl'), and engine shape ('vs')
# against the number of gears ('gear')
ftable(am + cyl + vs ~ gear,
       data = tb)
```

```
     am   0                 1
     cyl  4     6     8     4     6     8
     vs   0  1  0  1  0  1  0  1  0  1  0  1
gear
3         0  1  0  2 12  0  0  0  0  0  0  0
4         0  2  0  2  0  0  0  6  2  0  0  0
5         0  0  0  0  0  0  1  1  1  0  2  0
```

5. **Discussion**:

- This code `ftable(am + cyl + vs ~ gear, data = tb)` also uses the `ftable` function in R to generate a contingency table. However, there is a significant difference in the configuration of variables compared to the previous code.

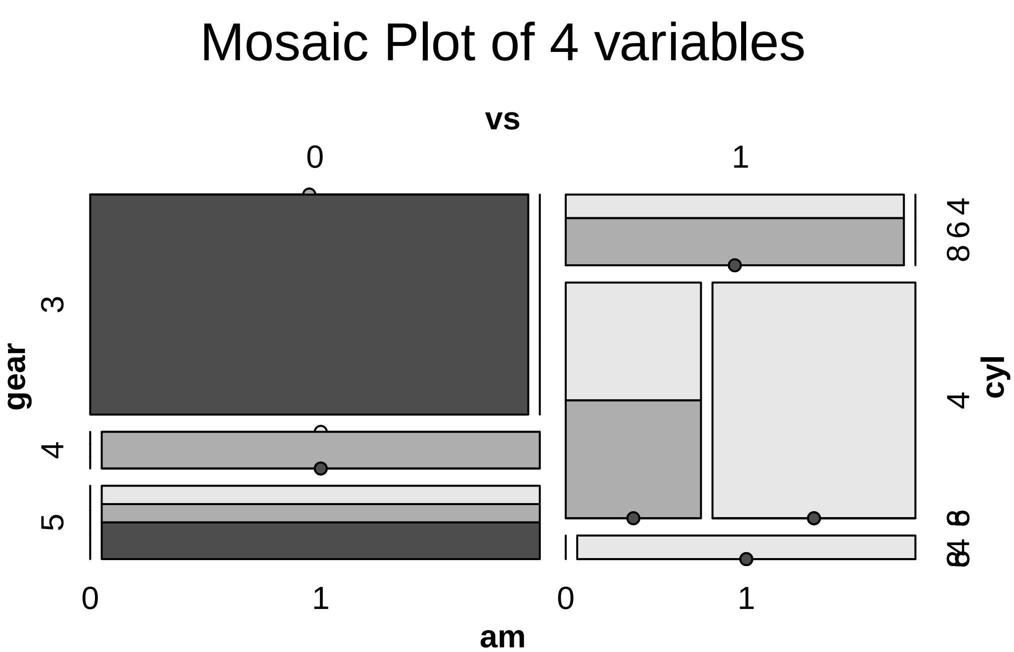
- In the previous code, `am + cyl ~ gear + vs`, the row variables (`am` and `cyl`) were cross-tabulated against the combinations of column variables (`gear` and `vs`).
- In this new code, we have `am + cyl + vs ~ gear`, which means that we now have three row variables (`am`, `cyl`, and `vs`) and one column variable (`gear`). This will result in a table showing the counts of each combination of `am`, `cyl`, and `vs` for each level of `gear`.
- In short, the new code has added `vs` as a third row variable and removed it from the column variables, thus changing the structure and the interpretation of the resulting table. [3]

9.3.2 Visualization using a Mosaic Plot

1. We can visualize four dimensional Contingency Tables using a Mosaic Plot. This is an extension to our previous discussions about two-way and three-way contingency tables. [5]

   Consider the following code:

```
# Create a mosaic plot of cyl vs, gear, am
vcd::mosaic(~ cyl + vs + gear + am,
            data = tb,
            main = "Mosaic Plot of 4 variables",  # Title
            shade = TRUE,  # Apply shading
            highlighting = "cyl" )  # Highlight based on cylinder
```



2. **Discussion**:

- `~ cyl + vs + gear + am` is a formula that indicates the categorical variables to be plotted. Each variable will be represented as a dimension in the plot, and their combined proportions will make up the whole plot.
- The `data = tb` argument informs the dataframe.
- The `main = "Mosaic Plot of 4 variables"` line sets the main title of the plot.
- `shade = TRUE` means that the cells in the mosaic plot will be shaded. The shading adds an extra visual element, which can make it easier to compare proportions.
- Finally, `highlighting = "cyl"` means that the cells in the plot will be highlighted based on the levels of the `cyl` variable. This can help to visually differentiate the categories of this variable. [5]

9.4 Summary of Chapter 9 – Bivariate Categorical data (Part 2 of 2)

In this chapter, we explore the world of multivariate categorical variables. Specifically, we focus on three-dimensional analyses that unveil complex relationships and dependencies between numerous variables. We utilize the R programming language, demonstrating a range of functions for constructing three-dimensional contingency tables. We also delve into segmenting these tables based on various variables, offering unique perspectives on the data.

Further, we expand on visualizing this data, discussing the creation of three-dimensional bar plots and mosaic plots. These powerful visual tools assist in data interpretation by representing the frequency of combinations of multiple variables. We don't limit ourselves to three-dimensional analysis; we advance into examining four-way relationships between categorical variables. Here, we utilize four-dimensional contingency tables and mosaic plots for analysis and visualization.

In essence, this chapter provides a robust understanding of multivariate categorical data handling and visualization, preparing readers to navigate and analyze complex datasets effectively. Overall, these three chapters together provide a comprehensive understanding of handling and visualizing multivariate categorical data.

9.5 References

Categorical Data Analysis:

[1] Agresti, A. (2018). An Introduction to Categorical Data Analysis (3rd ed.). Wiley.

Boston University School of Public Health. (2016). Analysis of Categorical Data. Retrieved from https://sphweb.bumc.bu.edu/otlt/MPH-Modules/BS/R/R6_CategoricalDataAnalysis/R6_CategoricalDataAnalysis2.html

Hair, J. F., Black, W. C., Babin, B. J., & Anderson, R. E. (2018). Multivariate Data Analysis (8th ed.). Cengage Learning.

Sheskin, D. J. (2011). Handbook of Parametric and Nonparametric Statistical Procedures (5th ed.). Chapman and Hall/CRC.

Tang, W., Tang, Y., & Song, X. (2012). Applied Categorical and Count Data Analysis. Chapman and Hall/CRC.

R Programming:

[2] Crawley, M. J. (2007). The R Book. Wiley.

Friendly, M., & Meyer, D. (2016). Discrete Data Analysis with R: Visualization and Modeling Techniques for Categorical and Count Data. Chapman and Hall/CRC.

Kabacoff, R. I. (2015). R in Action: Data Analysis and Graphics with R (2nd ed.). Manning Publications.

R Core Team. (2020). ftable: Flat Contingency Tables. In R Documentation. Retrieved from https://www.rdocumentation.org/packages/base/versions/3.6.2/topics/ftable

Data Visualization:

[3] Chang, W. (2018). R Graphics Cookbook: Practical Recipes for Visualizing Data. O'Reilly Media.

Friendly, M. (2000). Visualizing Categorical Data. SAS Institute.

Healy, K., & Lenard, M. T. (2014). A Practical Guide to Creating Better Looking Plots in R. University of Oregon. Retrieved from https://escholarship.org/uc/item/07m6r

Healy, K. (2018). Data Visualization: A Practical Introduction. Princeton University Press.

Heiberger, R. M., & Robbins, N. B. (2014). Design of Diverging Stacked Bar Charts for Likert Scales and Other Applications. Journal of Statistical Software, 57(5), 1-32. doi: 10.18637/jss.v057.i05.

Lander, J. P. (2019). R for Everyone: Advanced Analytics and Graphics (2nd ed.). Addison-Wesley Data & Analytics Series.

Unwin, A. (2015). Graphical Data Analysis with R. CRC Press.

Wilke, C. O. (2019). Fundamentals of Data Visualization. O'Reilly Media.

ggplot2:

[4] Wickham, H. (2016). ggplot2: Elegant Graphics for Data Analysis. Springer-Verlag New York. ISBN 978-3-319-24277-4. Retrieved from https://ggplot2.tidyverse.org.

Wickham, H., & Grolemund, G. (2016). R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. O'Reilly Media.

Wilkinson, L. (2005). The Grammar of Graphics (2nd ed.). Springer-Verlag.

Mosaic Plot:

[5] Hartigan, J. A., & Kleiner, B. (1981). Mosaics for Contingency Tables. In Computer Science and Statistics: Proceedings of the 13th Symposium on the Interface (pp. 268-273).

Meyer, D., Zeileis, A., & Hornik, K. (2020). vcd: Visualizing Categorical Data. R Package Version 1.4-8. Retrieved from https://CRAN.R-project.org/package=vcd

Friendly, M. (1994). Mosaic Displays for Multi-Way Contingency Tables. Journal of the American Statistical Association, 89(425), 190-200.

10 Univariate Continuous data (Part 1 of 2)

Chapter 10.

10.1 Exploring Univariate Continuous Data

This chapter explores how to summarize and visualize *univariate, continuous* data.

1. **Univariate** continuous data refers to data coming from **one feature or variable**, which could take on an infinite number of possible values, typically within an interval [1].
2. For instance, in the `mtcars` dataset in R, variables like `mpg` (miles per gallon), `wt` (weight), and `hp` (horsepower) epitomize continuous data. They are not limited to specific, separate numbers and can encompass any value, including decimal points, within their respective ranges.
3. **Data**: Suppose we run the following code to prepare the `mtcars` data for subsequent analysis and save it in a tibble called `tb`. [1]

```
# Load the required libraries, suppressing annoying startup messages
library(dplyr, quietly = TRUE, warn.conflicts = FALSE)
library(tibble, quietly = TRUE, warn.conflicts = FALSE)

# Read the mtcars dataset into a tibble called tb
data(mtcars)
tb <- as_tibble(mtcars)

# Convert relevant columns into factor variables
tb$cyl <- as.factor(tb$cyl) # cyl = {4,6,8}, number of cylinders
tb$am <- as.factor(tb$am) # am = {0,1}, 0:automatic, 1: manual transmission
tb$vs <- as.factor(tb$vs) # vs = {0,1}, v-shaped engine, 0:no, 1:yes
tb$gear <- as.factor(tb$gear) # gear = {3,4,5}, number of gears

# Directly access the data columns of tb, without tb$mpg
attach(tb)
```

10.1.1 Measures of Central Tendency

1. In our journey of understanding data, we often turn to certain statistical tools, among which, the measures of central tendency play a pivotal role. These measures provide a way to summarize our data with a single value that represents the "center" or the "average" of our data distribution. [2]
2. Primarily, there are three measures of central tendency that we often rely on: the mean, median, and mode. [2]
3. As an illustration, here is R code to determine the `mean` and `median` of the `wt` (weight) for all vehicles: [3]

```
# Mean of wt
mean(wt)
```

```
[1] 3.21725
```

```
# Median of wt
median(wt)
```

```
[1] 3.325
```

10.1.2 Measures of Variability

1. In our exploration of continuous data, we also consider measures of variability. These statistical measures provide insight into the spread or dispersion of our data points. To further illustrate the concepts we've discussed, we'll apply these measures of variability to the `mpg` column from the `mtcars` dataset. [3]
2. **Range**: This is the difference between the highest and the lowest value in our data set. However, while `range` is easy to calculate and understand, it is sensitive to outliers, so we must interpret it carefully. The `range()` function in R provides the minimum and maximum `mpg`. [3]

```
# Range of mpg
range(mpg)
```

```
[1] 10.4 33.9
```

3. **Min and Max**: We can off course measure the minimum and maximum values, using the following simple code. [3]

```
# Minimum mpg
min(mpg)
```

```
[1] 10.4
```

```
# Maximum mpg
max(tb$mpg)
```

```
[1] 33.9
```

4. **Variance**: It is calculated as the average of the squared deviations from the mean. Larger variances suggest that the data points are more spread out around the mean. One limitation of the variance is that its units are the square of the original data's units, which can make interpretation difficult. We use the var() function to compute the variance. [3]

```
# Variance of mpg
var(tb$mpg)
```

```
[1] 36.3241
```

5. **Standard Deviation:** This is simply the square root of the variance. Because it is in the same units as the original data, it is often easier to interpret than the variance. A larger standard deviation indicates a greater spread of data around the mean. [3]

```
# Standard Deviation of mpg
sd(tb$mpg)
```

```
[1] 6.026948
```

6. **Interquartile Range (IQR)**: It is another measure of dispersion, especially useful when we have skewed data or outliers. It represents the range within which the central 50% of our data falls. This measure is less sensitive to extreme values than the range, variance, or standard deviation. To find the interquartile range (IQR), which provides the spread of the middle 50% of the `mpg` values, we use the `IQR()` function. [3]

```
# Inter-Quartile Range of mpg
IQR(tb$mpg)
```

```
[1] 7.375
```

7. **Skewness** and **Kurtosis**:

- **Skewness** is a measure of the asymmetry of our data. Positive skewness indicates a distribution with a long right tail, while negative skewness indicates a distribution with a long left tail.
- **Kurtosis**, on the other hand, measures the "tailedness" of the distribution. A distribution with high kurtosis exhibits a distinct peak and heavy tails, while low kurtosis corresponds to a flatter shape.
- These two measures can be computed using the `skewness()` and `kurtosis()` functions from the `moments` package.

```
# Load moments package
suppressPackageStartupMessages(library(moments))

# Skewness of 'wt' in the mtcars dataframe
skewness(tb$wt)
```

```
[1] 0.4437855
```

```
# Kurtosis of 'wt' in the mtcars dataframe
kurtosis(tb$wt)
```

```
[1] 3.172471
```

8. Overall, these measures of variability help us quantify the dispersion and shape of our data, offering a more complete picture when combined with measures of central tendency. [3]

10.2 Summarizing Univariate Continuous Data

1. Our primary objective in summarizing data is to gain an initial overview or snapshot of the data set we're dealing with. This fundamental analysis provides us a sense of the data's central tendency, spread, and distribution shape, which in turn guides our decision-making process for subsequent stages of data analysis.
2. In R, the `summary()` function offers a succinct summary of the selected data object. When applied to a numeric vector such as `mpg` from the `mtcars` dataset, it yields the minimum and maximum values, the first quartile (25th percentile), the median (50th percentile), the third quartile (75th percentile), and the mean. [3]

```
# A summary of 'mpg'
summary(tb$mpg)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
  10.40   15.43   19.20   20.09   22.80   33.90 
```

3. The `describe()` function, part of the `psych` package, goes a step further by providing a more comprehensive summary of the data. It includes additional statistics like the number of valid (non-missing) observations, the standard deviation, and metrics of skewness and kurtosis [4].

```
suppressPackageStartupMessages(library(psych))

# A summary of 'mpg' using describe()
describe(mpg)
```

```
   vars  n  mean   sd median trimmed  mad  min  max range skew kurtosis   se
X1    1 32 20.09 6.03   19.2    19.7 5.41 10.4 33.9  23.5 0.61    -0.37 1.07
```

4. Specific columns from `describe(tb$mpg)`

```
# Select specific columns from describe(mpg)
columns = c("n","mean","sd","median","min","max","skew","kurtosis")
describe(mpg)[, columns]
```

```
    n  mean   sd median  min  max skew kurtosis
X1 32 20.09 6.03   19.2 10.4 33.9 0.61    -0.37
```

10.2.1 Summarizing an entire dataframe or tibble

1. The function `summary()` in R can also be employed to summarize the entirety of a dataframe or tibble in a comprehensive manner. [3].

```
# A summary of the tibble tb
summary(tb)
```

```
      mpg         cyl         disp              hp             drat      
 Min.   :10.40   4:11   Min.   : 71.1   Min.   : 52.0   Min.   :2.760  
 1st Qu.:15.43   6: 7   1st Qu.:120.8   1st Qu.: 96.5   1st Qu.:3.080  
 Median :19.20   8:14   Median :196.3   Median :123.0   Median :3.695  
 Mean   :20.09          Mean   :230.7   Mean   :146.7   Mean   :3.597  
 3rd Qu.:22.80          3rd Qu.:326.0   3rd Qu.:180.0   3rd Qu.:3.920  
 Max.   :33.90          Max.   :472.0   Max.   :335.0   Max.   :4.930  
       wt             qsec         vs     am     gear        carb      
 Min.   :1.513   Min.   :14.50   0:18   0:19   3:15   Min.   :1.000  
 1st Qu.:2.581   1st Qu.:16.89   1:14   1:13   4:12   1st Qu.:2.000  
 Median :3.325   Median :17.71                 5: 5   Median :2.000  
 Mean   :3.217   Mean   :17.85                        Mean   :2.812  
 3rd Qu.:3.610   3rd Qu.:18.90                        3rd Qu.:4.000  
 Max.   :5.424   Max.   :22.90                        Max.   :8.000  
```

2. **Discussion**

- For numeric columns, `summary()` delivers a six-number summary that includes minimum, first quartile (Q1 or 25th percentile), median (Q2 or 50th percentile), mean, third quartile (Q3 or 75th percentile), and maximum. This gives a broad understanding of the central tendency and dispersion of the data within each numeric column.
- For categorical (factor) columns, `summary()` generates the counts of each category level. The output of this code is essentially a comprehensive snapshot of the `tb` tibble, enabling us to quickly understand the nature of our data. [3]

3. To obtain a more detailed statistical summary of an entire dataframe or tibble, we can employ the `describe()` function from the `psych` package [4].

```
# Select specific columns from describe(mpg)
columns = c("n","mean","sd","median","min","max","skew","kurtosis")
describe(tb)[, columns]
```

```
      n   mean     sd median   min    max  skew kurtosis
mpg   32  20.09   6.03  19.20 10.40  33.90  0.61    -0.37
cyl*  32   2.09   0.89   2.00  1.00   3.00 -0.17    -1.76
disp  32 230.72 123.94 196.30 71.10 472.00  0.38    -1.21
hp    32 146.69  68.56 123.00 52.00 335.00  0.73    -0.14
drat  32   3.60   0.53   3.70  2.76   4.93  0.27    -0.71
wt    32   3.22   0.98   3.33  1.51   5.42  0.42    -0.02
qsec  32  17.85   1.79  17.71 14.50  22.90  0.37     0.34
vs*   32   1.44   0.50   1.00  1.00   2.00  0.24    -2.00
am*   32   1.41   0.50   1.00  1.00   2.00  0.36    -1.92
```

```
gear* 32   1.69   0.74   2.00  1.00   3.00  0.53    -1.07
carb  32   2.81   1.62   2.00  1.00   8.00  1.05     1.26
```

4. **Discussion:**

- The `describe()` function analyzes each column in the provided tibble individually and outputs a range of useful statistics. For numeric columns, it offers count, mean, standard deviation, trimmed mean, minimum and maximum values, range, skewness, and kurtosis among others.
- For non-numeric or factor columns, the `describe()` function still provides a count of elements but defaults to `NA` for the rest of the statistics, as these metrics are not applicable.

10.3 Visualizing Univariate Continuous Data

- In our journey to explore and understand univariate continuous data, visualizations act as our valuable companions. Visual graphics provide us with an instant and clear understanding of the underlying data patterns and distributions that may otherwise be challenging to discern from raw numerical data.
- Let's take a closer look at some of the most effective ways of visualizing univariate continuous data, including

(i) Bee Swarm plots;

(ii) Stem-and-Leaf plots

(iii) Histograms;

(iv) PDF and CDF Density plots;

(v) Box plots;

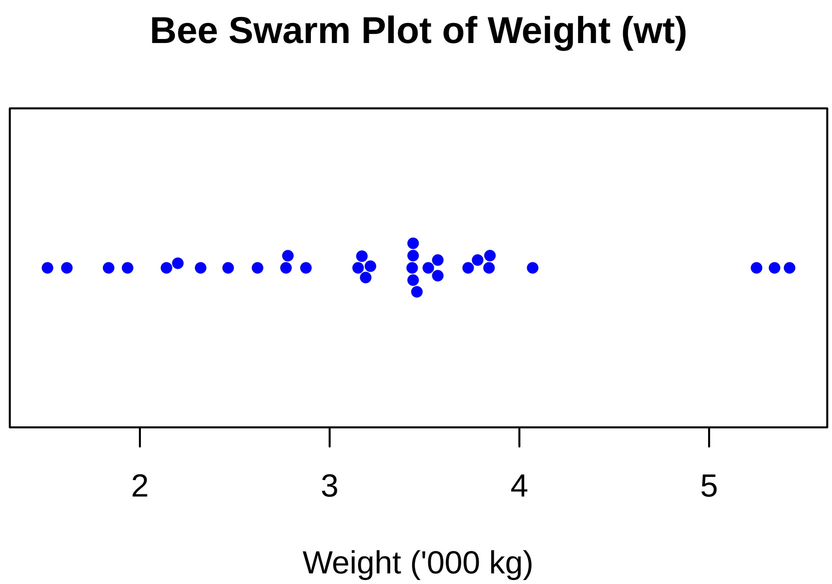
(vi) Violin plots;

(vii) Quantile-Quantile (Q-Q) Plots

10.3.1 Bee Swarm Plot

1. A Bee Swarm plot is a *one-dimensional scatter plot* that reduces overlap and provides a better representation of the distribution of individual data points. This type of plot provides a more detailed view of the data, particularly for smaller data sets. [5]

```r
# Load the beeswarm package
library(beeswarm)

# Create a bee swarm plot of wt column
beeswarm(tb$wt,
         main="Bee Swarm Plot of Weight (wt)",
         xlab = "Weight ('000 kg)",
         pch=16,  # type of points
         cex=0.8, # size of the points
         col="blue",
         horizontal = TRUE)
```



2. **Discussion**

- In the above code, we load the `beeswarm` package using the `library()` function.
- We then create a bee swarm plot of the `wt` column using the `beeswarm()` function.
- The `main` argument is used to specify the title of the plot.
- The `pch` argument is used to set the type of points to be plotted, and the `cex` argument is used to set the size of the points.
- The `col` argument is used to set the color of the points.
- The resulting plot displays the individual `wt` values in the dataset as points on a horizontal axis, with no overlap between points. This provides a visual representation of the distribution of the data, as well as any outliers or gaps in the data.

- `horizontal = TRUE`: Renders it horizontally rather than the default vertical orientation.

10.3.2 Stem-and-Leaf Plot

1. Stem-and-leaf plots serve as an efficient tool for visualizing the distribution of data, particularly when working with small to medium-sized datasets. The method involves breaking down each data point into a "stem" and a "leaf", with the "stem" representing the primary digit(s) and the "leaf" embodying the subsequent digit(s) [7]
2. We can utilize the `stem()` function in R to devise stem-and-leaf plots. Here's how we can apply it to the `wt` column:

```
stem(wt)
```

```

  The decimal point is at the |

  1 | 5689
  2 | 123
  2 | 56889
  3 | 22224444
  3 | 55667888
  4 | 1
  4 |
  5 | 334
```

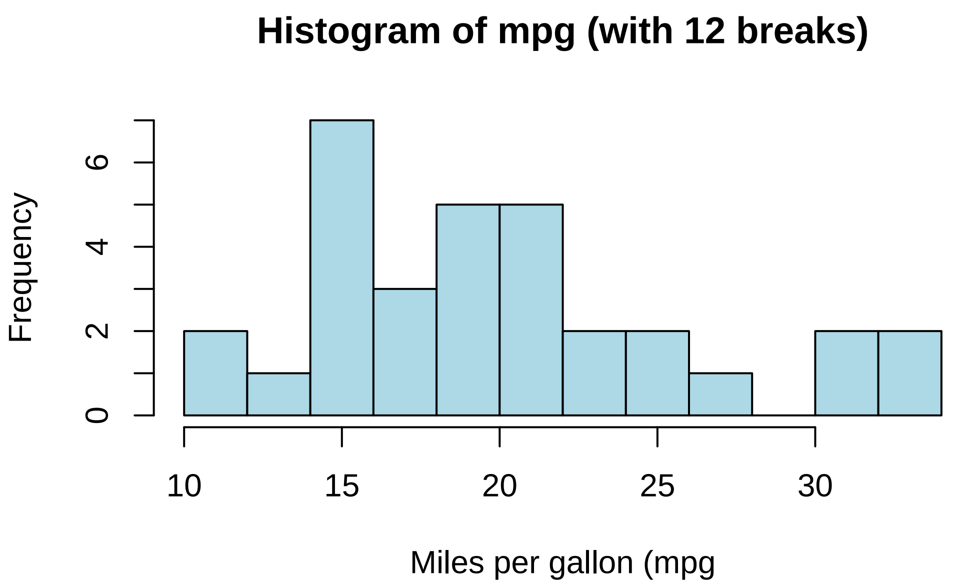
3. In the resulting plot, the vertical bar ("|") symbolizes the decimal point's location.
4. This visual representation enables us to swiftly assess the data's distribution, the center, and the spread, in a fashion similar to a histogram. However, unlike a histogram, a stem-and-leaf plot retains the original data to a certain degree, providing more granular detail.

10.3.3 Histogram

1. A histogram is a graphical representation showcasing the *frequency* of discrete or grouped data points within a dataset.
2. It splits the data into equal-width bins, with the height of each bar matching the frequency of data points in each respective bin.

3. It serves as a valuable tool for demonstrating the distribution shape of the data. In R, we can construct a histogram using the `hist()` function and control its appearance. The final histogram visually depicts the frequency of `mpg` values in the dataset, where each bar represents the count of observations within a specific range of values. [6]

```
# Create a histogram of mpg column with bins of equal width
hist(tb$mpg,
     breaks = 12, # This creates 12 bins of equal width
     main="Histogram of mpg (with 12 breaks)",
     xlab="Miles per gallon (mpg",
     col="lightblue",
     border="black")
```

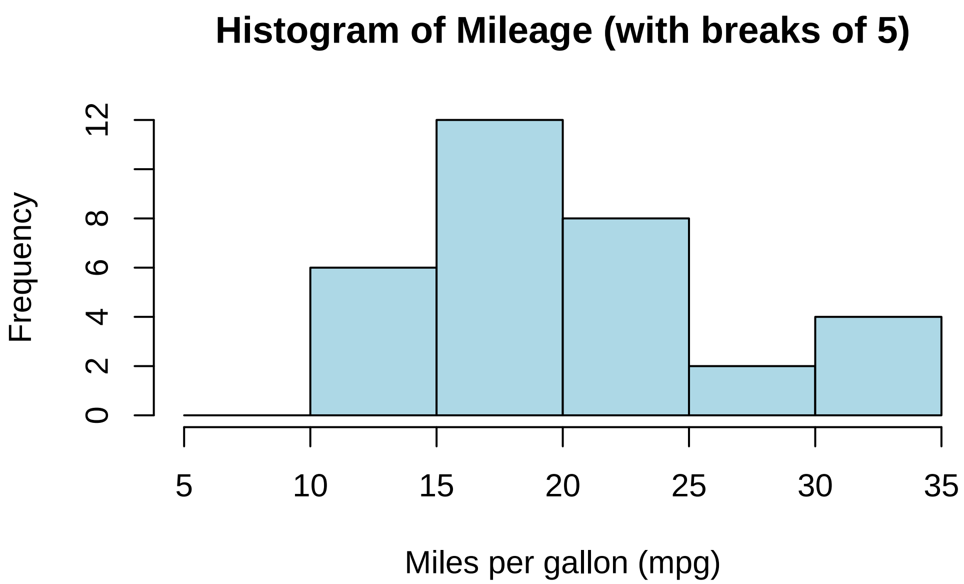


4. **Discussion:**

- This code generates a histogram of `mpg` using the `hist()` function. The `main` argument denotes the plot's title, while the `xlab` argument labels the x-axis.
- We use the `col` argument to specify the color of the histogram bars, and the `border` argument to determine the color of the bar borders.
- We can control the *number of bins* or the ranges of the bins in a histogram using the `breaks` argument inside the `hist()` function:

5. We can alternately specify the *ranges of the bins*:

```r
# Create a histogram of mpg column with specific bin ranges
hist(tb$mpg,
     breaks = seq(5, 35, by = 5), # bins with ranges 10-15, 15-20, etc.
     main="Histogram of Mileage (with breaks of 5)",
     xlab="Miles per gallon (mpg)",
     col="lightblue",
     border="black")
```



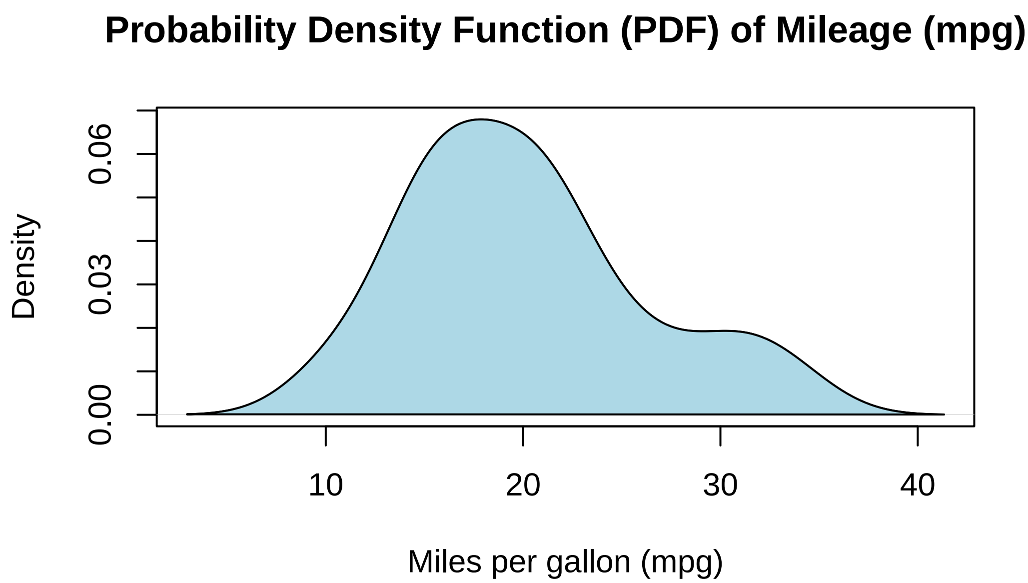
6. **Discussion:**

- The breaks argument uses the `seq()` function to create a sequence of break points from 5 to 35, with a step of 5.
- This results in bins with ranges 5-10, 10-15, 15-20, 20-25, 25-30, and 30-35. [7]

10.3.4 Probability Density Function (PDF) plot

1. Smoothed approximations of histograms are often represented by density plots, as they assist in offering an estimation of the underlying continuous probability distribution of a given dataset. [3]
2. Compared to histograms, these plots often present superior accuracy and aesthetic appeal, and they eliminate the need for arbitrary bin selection. A density plot shares several similarities with a histogram. However, instead of presenting the frequency of individual values, it conveys the probability density of the dataset. [3]

```r
# Calculate density
dens <- density(mtcars$mpg)
# Create a density plot
plot(dens,
     main = "Probability Density Function (PDF) of Mileage (mpg)",
     xlab = "Miles per gallon (mpg)",
)
# Add a polygon to fill under the density curve
polygon(dens, col = "lightblue", border = "black")
```



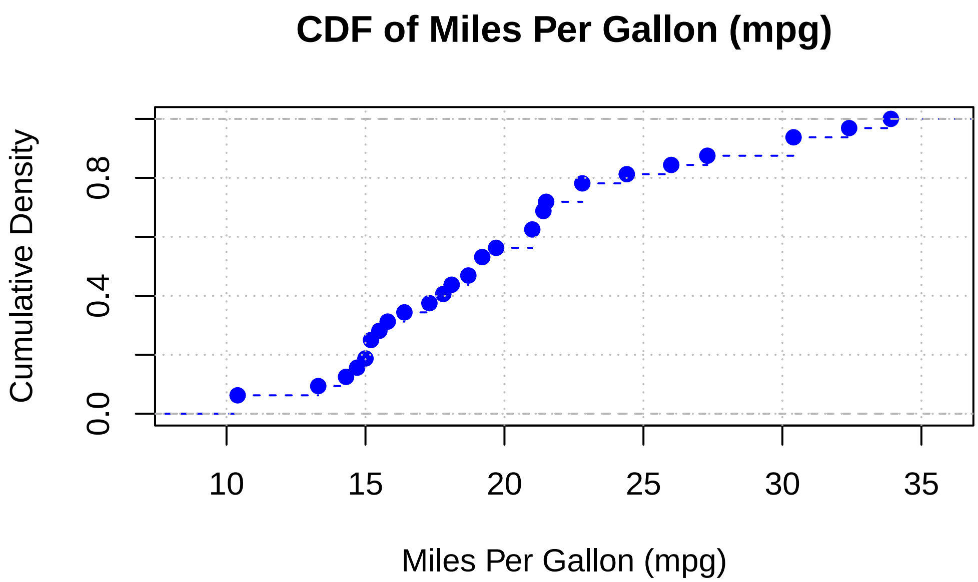
3. **Discussion:**

- We use the `density()` function to generate a PDF plot for the `mpg` column.
- Here, we utilize the `plot()` function to graph the resulting density object.
- The `main` argument stipulates the title of the plot, while the `xlab` argument designates the label for the x-axis.
- Through the `polygon()` function, we determine the shaded color.
- The final plot displays the probability density of `mpg` values, using the curve to signify the data distribution. [3]

10.3.5 Cumulative Distribution Function (CDF) Plot

1. CDF plots visualize the fraction of data points that are less than or equal to a specified value on the x-axis [3].
2. They facilitate easy representation of the median, percentiles, and spread.
3. In R, we can employ the `ecdf()` function to generate a CDF plot.

```
# Create a CDF plot of mpg column
plot(ecdf(tb$mpg),
     main = "CDF of Miles Per Gallon (mpg)",
     xlab = "Miles Per Gallon (mpg)",
     ylab = "Cumulative Density",
     col = "blue",        # Line color
     lty = 2,
     )
grid(col = "gray", lty = "dotted")  # Add a grid to the plot
```



4. **Discussion:**

- `ecdf(tb$mpg)`: The function `ecdf()` computes the empirical cumulative distribution function (CDF) for the mpg column from the `tb` data frame.
- `plot(ecdf(tb$mpg))`: Plots the CDF of the mpg column.
- `main, xlab, ylab`: Set the title and axis labels.

- `col = "blue"`: Colors the plot line blue; `lty = 2`: Uses a dashed line.
- `grid(col = "gray", lty = "dotted")`: Adds a gray, dotted grid to the plot.

5. Computing and Inversing CDF Values

- The following code demonstrates how to determine the cumulative distribution function (CDF) value for a given `mpg` using `ecdf()` and how to find the `mpg` value corresponding to a specific CDF with `quantile()`.
- Suppose we want to identify the CDF at `mpg = 20`, here is how we do it:

```
# Generate the empirical cumulative distribution function
ecdf_func <- ecdf(tb$mpg)

# Derive the CDF for mpg = 20
ecdf_func(20)
```

```
[1] 0.5625
```

- If we're interested in knowing the `mpg` value that corresponds to a certain CDF value, the `quantile()` function comes to our aid. For instance, we can obtain the `mpg` value associated with a CDF of `0.6` as follows:

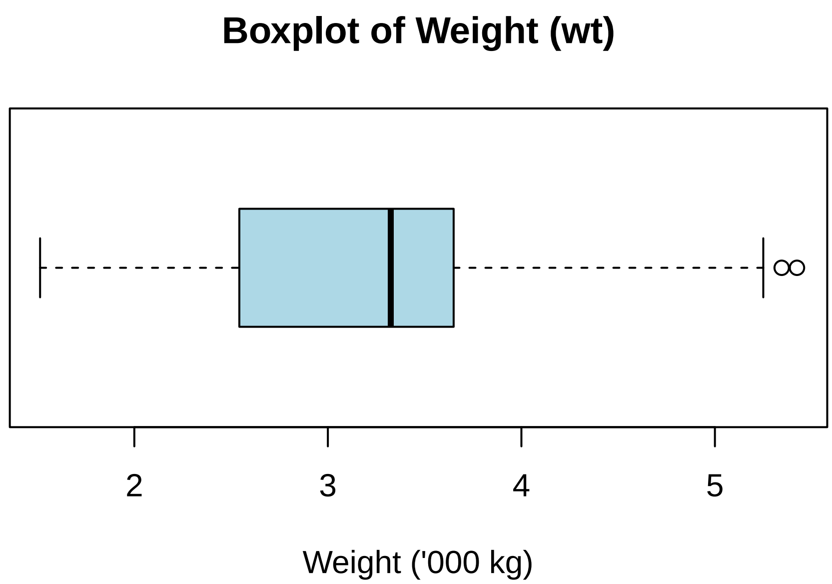
```
# Discover the mpg corresponding to CDF = 0.6
quantile(tb$mpg, 0.6)
```

```
60%
 21
```

10.3.6 Box Plot

1. Box-and-whisker plots, commonly known as box plots, are crucial graphical instruments for illustrating a distribution's center, spread, and potential outliers [7].
2. Here is sample code to generate a boxplot of `wt` (Weight) of the cars. [3]

```
boxplot(tb$wt,
        main = "Boxplot of Weight (wt)",
        ylab = "",
        xlab = "Weight ('000 kg)",
        col = "lightblue",
        horizontal = TRUE
       )
```



3. The box plot's construction involves the use of an *interquartile range (IQR)* represented by a box, which contains the middle 50% of the dataset, from Q1 to Q3.
4. The box's internal line signifies the median, while the "whiskers" reach out to the smallest and largest observations within a distance of 1.5 times the IQR.
5. The whiskers extend to the *minimum and maximum non-outlier values*, or *1.5 times the interquartile range beyond the quartiles*, whichever is shorter.
6. Any points outside of the whiskers are considered outliers and are plotted individually.
7. Discussion:

- The above R code creates a horizontal light blue boxplot for the `wt` column of the `tb` data frame, titled "Boxplot of Weight (wt)", with the x-axis labeled "Weight ('000 kg)" and no y-axis label.
- `horizontal = TRUE`: Renders the boxplot horizontally rather than the default vertical orientation.

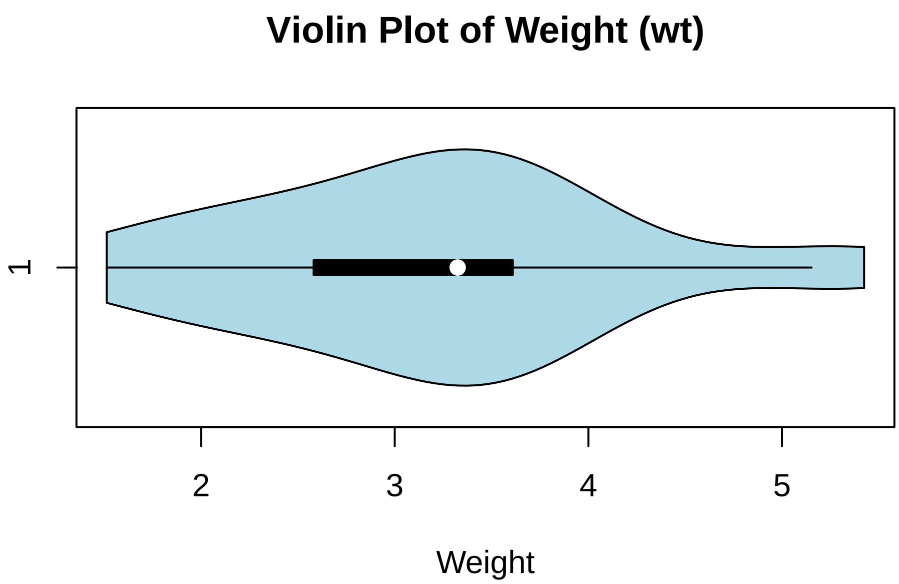
10.3.7 Violin Plot

1. Violin plots are a compelling tool to merge the benefits of box plots and kernel density plots and enable us to depict a detailed view of data distribution.
2. These plots exhibit the *probability density* at different values, where the plot's *breadth* represents the density or frequency of data points. More extensive areas denote a higher aggregation of data points Akin to a box plot, a violin plot provides a visual display of

the entire data distribution via a *kernel density estimate*, as opposed to just presenting the quartiles [8].

3. The `vioplot()` function, part of the `vioplot` package in R, allows us to create a violin plot.

```
suppressPackageStartupMessages(library(vioplot))
# Constructing a violin plot for the wt
vioplot(tb$wt,
        main="Violin Plot of Weight (wt)",
        col = "lightblue",
        horizontal = TRUE,
        xlab="Weight"
        )
```



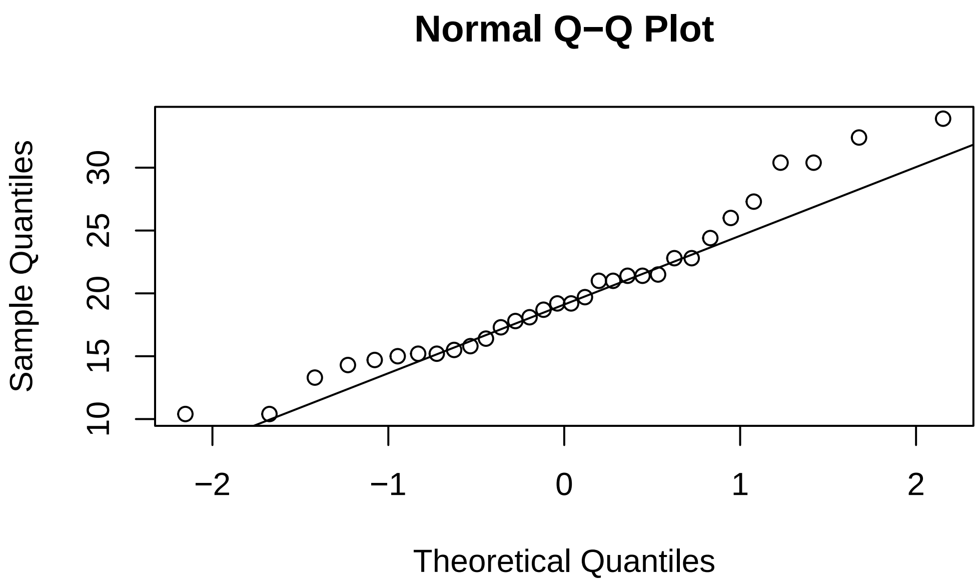
4. **Discussion**:

- In this code, the `vioplot()` function crafts a violin plot for the `wt` variable. We use the `main` argument to assign the plot's title and the `xlab` argument to designate the axis label.
- The resulting plot unveils the entire `wt` data distribution, with a kernel density estimate indicating the concentration of data points at different sections.
- The plot also visualizes the boxplot and the median, quartiles, and any outliers present in the data.
- `horizontal = TRUE`: Renders the violin plot horizontally rather than the default vertical orientation.

10.3.8 Quantile-Quantile (Q-Q) Plot

1. Quantile-Quantile plots, commonly referred to as Q-Q plots, are a visual tool we use to check if data follows a particular distribution, like a normal distribution.
2. Suppose we order a data column from the smallest to the biggest value, and each data point gets a *score* based on its position. This is what we call a *quantile*. Now, imagine a perfectly normal distribution doing the same thing. In a Q-Q plot, we compare our data's scores to the scores from the ideal normal distribution.
3. If our data aligns with the normal distribution, the points in the Q-Q plot will form a straight line. But if our data doesn't follow the normal distribution, the points will stray from the line. This way, the Q-Q plot gives us an intuitive, visual way to decide if our data is normally distributed or not [3].
4. In R, we can use the `qqnorm()` function to create the plot and the `qqline()` function to add the reference line. If the points lie close to the reference line, it's a good indication that our data is normally distributed.

```
# Generate a Q-Q plot for 'mpg' column
qqnorm(tb$mpg)
# Add a reference line to the plot
qqline(tb$mpg)
```



5. This approach isn't limited to normal distributions. We can compare our data with other distributions too, which makes Q-Q plots a versatile tool for understanding our data.

10.4 Summary of Chapter 10 – Univariate Continuous data (Part 1 of 2)

This chapter examines continuous univariate data, focusing on single variables in the 'mtcars' dataset using R's dplyr and ggplot packages. We employ R's inherent functions and the 'modeest' package to compute the mean, median, and mode, alongside variability measures like range, variance, and standard deviation.

We use R's `summary()` and the `psych` package's `describe()` functions to create succinct and detailed overviews of our data, providing insights into its central tendency, spread, and distribution shape. These functions can also summarize an entire dataframe or tibble, setting the stage for future analysis.

Visualisations are key to understanding data patterns and distributions. We use bee swarm plots, box plots, violin plots, histograms, and density plots. Bee swarm plots, using the `beeswarm()` function, show all data points and their distributions. Stem-and-leaf plots, created using the `stem()` function, provide a quick evaluation of the distribution.

Histograms, constructed with the `hist()` function, and density plots, using the `density()` function, display data frequency and smoothed approximations respectively. Cumulative Distribution Function (CDF) plots, via the `ecdf()` function, show the proportion of data points equal to or less than specific values.

Box plots, made with the `boxplot` function, highlight the distribution's center, spread, and outliers. Violin plots, via the `vioplot()` function, merge box plots and kernel density plots to display data density. Lastly, Q-Q plots, created using `qqnorm()` and `qqline()`, verify if data follows a normal distribution.

To summarize, this chapter presents key R functions and techniques for visualizing continuous univariate data, providing valuable insights into data patterns and distributions.

10.5 References

R Programming:

[1] Crawley, M. J. (2007). The R Book. Wiley.

Kabacoff, R. I. (2015). R in Action: Data Analysis and Graphics with R (2nd ed.). Manning Publications.

R Core Team. (2020). R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria. Retrieved from https://www.R-project.org/.

Statistics:

[2] Bogaert, P. (2021). “A Comparison of Kernel Density Estimators.” Computational Statistics & Data Analysis, 77, 402-413.

Downey, A. B. (2014). Think Stats: Exploratory Data Analysis. O’Reilly Media.

Gravetter, F. J., & Wallnau, L. B. (2016). Statistics for the Behavioral Sciences. Cengage Learning.

Kleiber, C., & Zeileis, A. (2008). Applied Econometrics with R. Springer Science & Business Media.

Moore, D. S., McCabe, G. P., & Craig, B. A. (2012). Introduction to the Practice of Statistics. Freeman.

Triola, M. (2017). Elementary Statistics. Pearson.

Tukey, J. W. (1977). Exploratory Data Analysis. Addison-Wesley.

Thode Jr, H. C. (2002). Testing for Normality. CRC Press.

Venables, W. N., & Ripley, B. D. (2002). Modern Applied Statistics with S (4th ed.). Springer.

Statistics using R:

[3] Field, A., Miles, J., & Field, Z. (2012). Discovering Statistics Using R. Sage Publications.

R Core Team. (2020). R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria. Retrieved from https://www.R-project.org/.

Hyndman, R. J., & Fan, Y. (1996). Sample Quantiles in Statistical Packages. The American Statistician, 50(4), 361-365.

psych:

[4] Revelle, W. (2020). psych: Procedures for Psychological, Psychometric, and Personality Research. Northwestern University, Evanston, Illinois. R Package Version 2.0.12. Retrieved from https://CRAN.R-project.org/package=psych.

Beeswarm:

[5] Ellis, K. (2011). Beeswarm: The Bee Swarm Plot, an Alternative to Stripchart. R Package Version 0.2.3.

Histograms:

[6] Scott, D. W. (1979). On Optimal and Data-Based Histograms. Biometrika, 66(3), 605-610.

Wand, M. P., & Jones, M. C. (1995). Kernel Smoothing. Chapman and Hall/CRC.

Boxplots:

[7] McGill, R., Tukey, J. W., & Larsen, W. A. (1978). Variations of Box Plots. The American Statistician, 32(1), 12-16.

Violin Plots:

[8] Hintze, J. L., & Nelson, R. D. (1998). Violin Plots: A Box Plot-Density Trace Synergism. The American Statistician, 52(2), 181-184.

11 Univariate Continuous data (Part 2 of 2)

Chapter 11.

11.1 Exploring Univariate Continuous Data using `ggplot2` and `ggpubr`

This chapter demonstrates the use of the popular **ggplot2** and **ggpubr** packages to further explore *univariate, continuous* data.

1. **ggplot2**: In the `ggplot2` package for instance, the function `geom_boxplot()` produces box plots, `geom_violin()` creates violin plots, and `geom_histogram()` and `geom_density()` generate histograms and density plots, respectively. The related **ggbeeswarm** package can be used for creating bee swarm plots. [1]

2. **ggpubr**: The `ggpubr` package in R augments `ggplot2` by offering tools for creating publication-ready plots. It enables simplified plotting with easy-to-use functions like gghistogram(), ggdensity(), ggboxplot(), ggviolin, and makes it easy to merge multiple plots with `ggarrange()`, and provides specialized themes for a polished look. Essentially, **ggpubr** merges `ggplot2`'s extensive customization with the ease of creating visually appealing and informative plots. [2]

3. We load the necessary packages, including `ggplot2`, `dplyr` and **ggthemes** packages. The package **ggthemes** allows us to use a variety of themes. [3]

```r
# Load the required libraries, suppressing annoying startup messages
library(dplyr, quietly = TRUE, warn.conflicts = FALSE)
library(tibble, quietly = TRUE, warn.conflicts = FALSE)
library(ggplot2, quietly = TRUE, warn.conflicts = FALSE)
library(ggpubr, quietly = TRUE, warn.conflicts = FALSE)

library(rmarkdown, quietly = TRUE, warn.conflicts = FALSE)
library(knitr, quietly = TRUE, warn.conflicts = FALSE)
library(kableExtra, quietly = TRUE, warn.conflicts = FALSE)

library(ggthemes)
```

5. **Data**: Suppose we run the following code to prepare the `mtcars` data for subsequent analysis and save it in a tibble called `tb`.

```
# Read the mtcars dataset into a tibble called tb
data(mtcars)
tb <- as_tibble(mtcars)

# Convert relevant columns into factor variables
tb$cyl <- as.factor(tb$cyl) # cyl = {4,6,8}, number of cylinders
tb$am <- as.factor(tb$am) # am = {0,1}, 0:automatic, 1: manual transmission
tb$vs <- as.factor(tb$vs) # vs = {0,1}, v-shaped engine, 0:no, 1:yes
tb$gear <- as.factor(tb$gear) # gear = {3,4,5}, number of gears
```

5. Let's take a closer look at some of the most effective ways of Visualizing Univariate Continuous Data using `ggplot2` and related packages, including

- Bee Swarm plots using `ggbeeswarm`
- Histograms using `ggplot2` and `ggpubr`
- PDF and CDF Density plots using `ggplot2` and `ggpubr`
- Bar plots using `ggplot2`
- Box plots using `ggplot2` and `ggpubr`
- Violin plots using `ggplot2` and `ggpubr`
- Quantile-Quantile (Q-Q) Plots using `ggplot2`

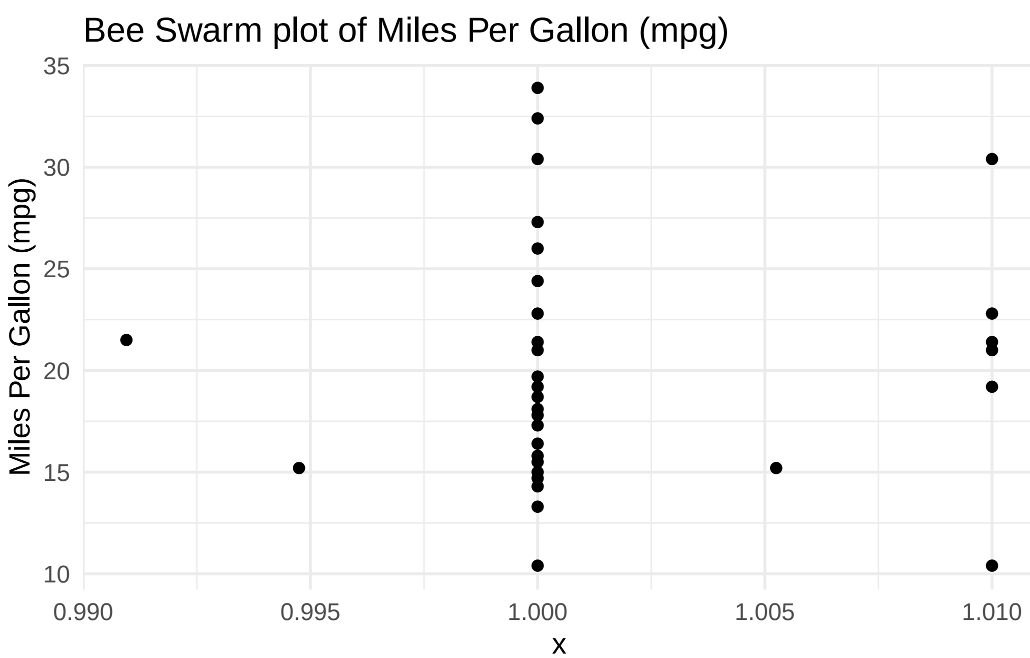
Note that it is inconvenient to create Stem-and-Leaf plots using `ggplot2`.

11.1.1 Bee Swarm plot using `ggbeeswarm`

1. The bee swarm plot is an alternative to the box plot, where each point is plotted in a manner that avoids overlap.
2. We use the `ggbeeswarm` package on the `mpg` column of the `tb` tibble.

```
# Load necessary libraries
library(ggplot2)
library(ggbeeswarm) # Provides the geom_beeswarm() function

# Create a bee swarm plot for visualizing the distribution of mpg
ggplot(tb,
       aes(x = 1,  # Dummy x-axis value
```

```
         y = mpg)) +  # mpg values on the y-axis
  geom_beeswarm() +  # Use beeswarm plot  points
  labs(title = "Bee Swarm plot of Miles Per Gallon (mpg)",
       y = "Miles Per Gallon (mpg)") +  # Label for the y-axis
  theme_minimal()  # theme for a clean and clear presentation
```



3. Discussion:

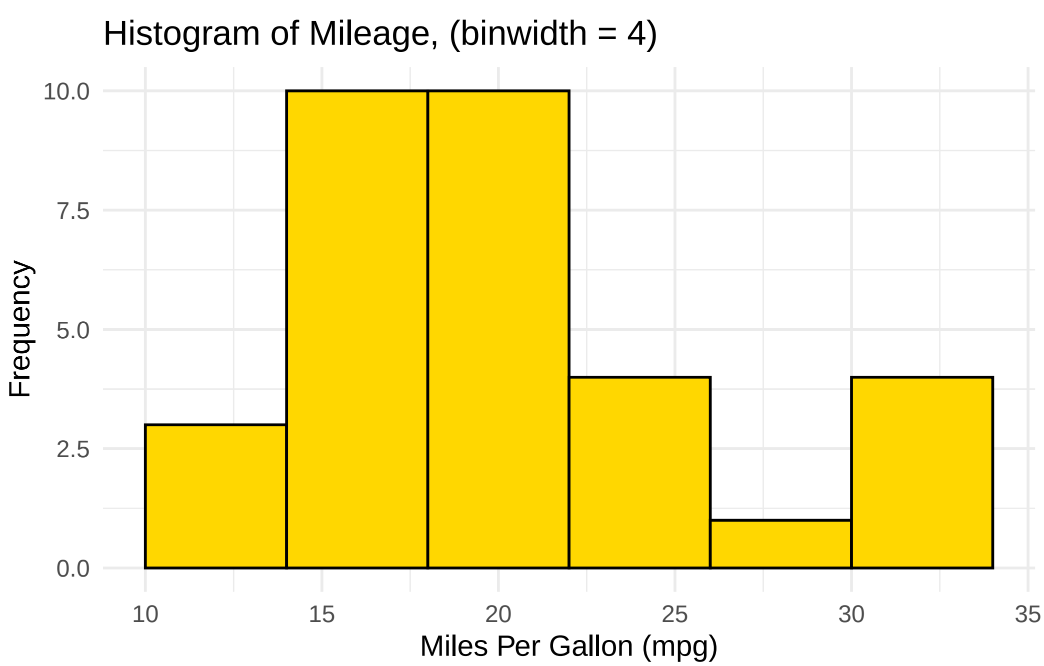
- Initially, we declare our dataset and the aesthetic mappings, defining how variables in the data are visually represented. For the bee swarm plot, we only need a y aesthetic, which is `mpg`. We set the x aesthetic to 1 as a placeholder, because bee swarm plots require an x aesthetic, but we only have one variable.
- Following that, we append a bee swarm plot using the `geom_beeswarm()` function.
- We use the `labs()` function to label the plot.
- We then adopt a minimalist theme by using `theme_minimal()` to give our plot a sleek and simple look. [4]

11.2 Histogram using `ggplot2`

11.2.1 Histogram with binwidth

1. The following code creates a histogram using the `ggplot2` package. Here, we pre-specify the bin width and the resulting number of bins in the histogram depend on the range of the data. [1]

```
# histogram using ggplot2 to visualize the distribution of mpg
ggplot(tb,
       aes(x = mpg)) +  # Set mpg as the x-axis variable
  geom_histogram(binwidth = 4,  # Define the binwidth for the histogram
                 fill = "gold",  # Set the fill color of the bins to gold
                 color = "black") +  # Set the border color of the bin
  theme_minimal() +  # Apply a minimal theme for a cleaner look
  labs(title = "Histogram of Mileage, (binwidth = 4)",
       x = "Miles Per Gallon (mpg)",  # Label for the x-axis
       y = "Frequency")  # Label for the y-axis
```



2. Discussion:

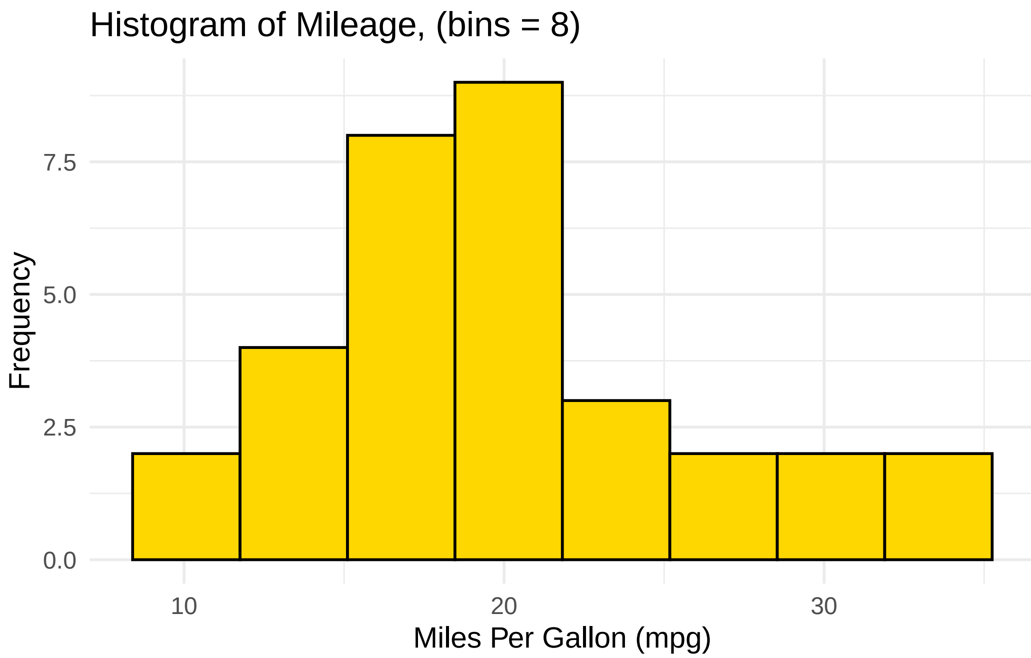
- The code `ggplot(tb, aes(x = mpg))` initializes a plot using the `tb` data frame, mapping the `mpg` column to the x-axis.

- The histogram is created with `geom_histogram()`, using an adjustable `binwidth = 4`. Given this bin width, the resulting number of bins in the histogram depend on the range of `mpg`.
- The `binwidth` argument specifies the width of the bins in the histogram, and we have chosen `4` as an arbitrary width.
- We use `fill` and `color` to set the bar colors to be gold with a black border.
- A clean appearance is achieved with `theme_minimal()`, and titles and labels are added using `labs()`. [1]

11.2.2 Histogram with bins

3. We could alternately set the number of bins in the histogram, instead of specifying the bin width. In this case, the bin-width gets calculated depending on the range of the data and the specified number of bins.

```
# Create a histogram of mpg from the tb dataset
ggplot(tb, aes(x = mpg)) +
  geom_histogram(bins = 8, fill = "gold", color = "black") +
  # Set histogram bins to 8 with gold fill and black border
  theme_minimal() + # Use a minimalistic theme
  labs(title = "Histogram of Mileage, (bins = 8)",
       x = "Miles Per Gallon (mpg)",
       y = "Frequency") # Add title and axis labels
```



4. Discussion:

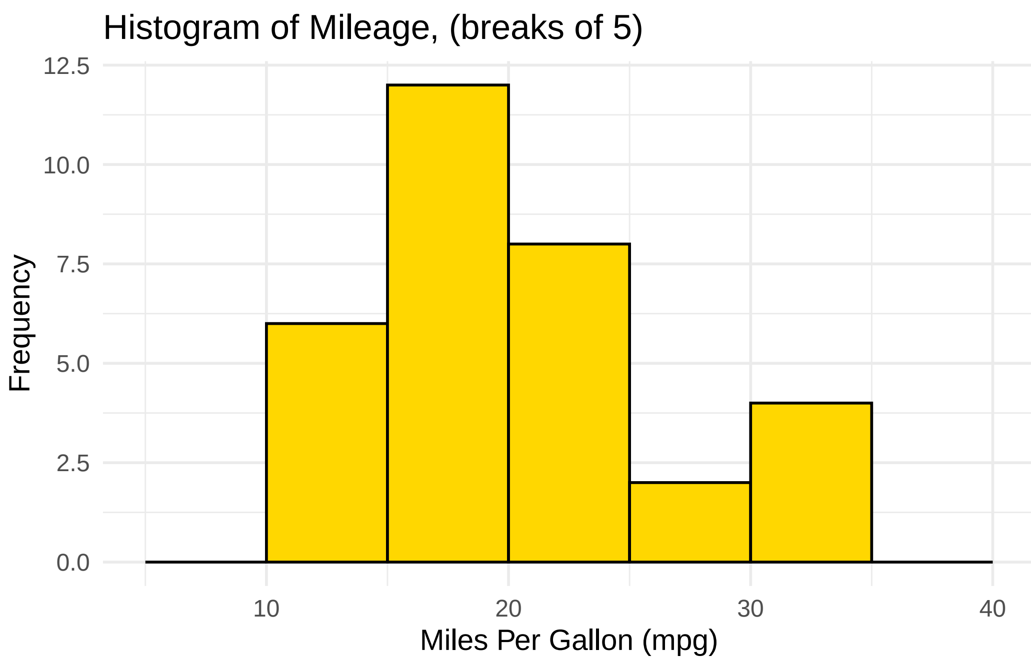
- We instruct R to create a histogram having 8 bins of equal width, by setting `bins = 8` in `geom_histogram()`
- The width of each bin is adjusted by dividing the range of `mpg` by the number of specified bins. [1]

11.2.3 Histogram with bin range

5. Alternately, we can specify custom bin ranges in a histogram. In this this approach, we supply a vector of breakpoints which defines the range of each bin. For example, the following code defines histogram bins with ranges of 5-10, 10-15, 15-20, 20-25, 25-30, 30-35, 35-40, for the `mpg` variable

```
# Plot a histogram of mpg values from the tb dataset using ggplot2
ggplot(tb, aes(x = mpg)) +
  geom_histogram(breaks = seq(5, 40, by = 5),
                 fill = "gold",
                 color = "black") +
  # Define histogram breaks at intervals of 5
  theme_minimal() + # Apply a minimal theme for clarity
  labs(title = "Histogram of Mileage, (breaks of 5)",
```

```
     x = "Miles Per Gallon (mpg)",
     y = "Frequency") # Add plot title and axis labels
```



6. **Discussion:**

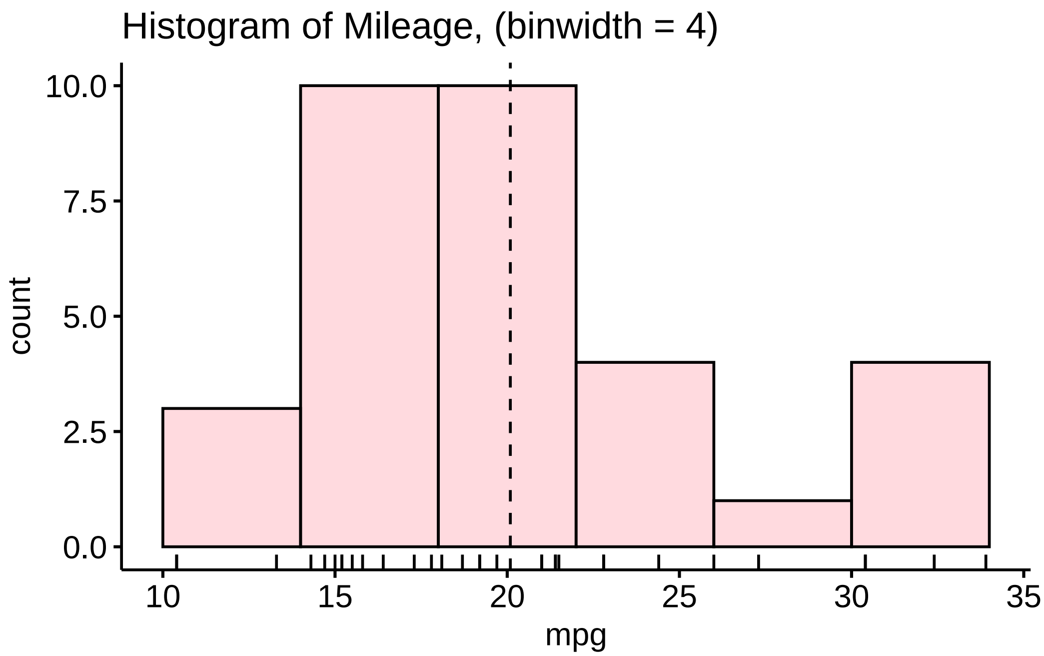
- `ggplot(tb, aes(x = mpg))` initializes a ggplot object with the `tb` data frame and sets the `mpg` column as the x-axis variable.
- `geom_histogram()` adds a histogram layer, in which `breaks = seq(5, 40, by = 5)` specifies bin edges using a sequence that starts at 5, ends at 40, and increases by 5 units. This results in bins like [5,10), [10,15), and so on. [1]

11.2.4 Histogram using `ggpubr`

7. Recreating a histogram with binwidth of 4, using `ggpubr`:

```
# Load the ggpubr package for enhanced ggplot2-based plotting
library(ggpubr)

# Create a histogram using gghistogram from ggpubr
gghistogram(tb,
            x = "mpg",   # Set 'mpg' as the variable for histogram
            binwidth = 4,  # Set the width of bins in the histogram
```

```
             add = "mean",  # Add a line indicating the mean of 'mpg'
             rug = TRUE,  # Add a rug plot to show individual data points
             color = "black",  # Set the color of bins to black
             fill = "lightpink",  # Set the fill color of bins to light pink
             title = "Histogram of Mileage, (binwidth = 4)"
)
```



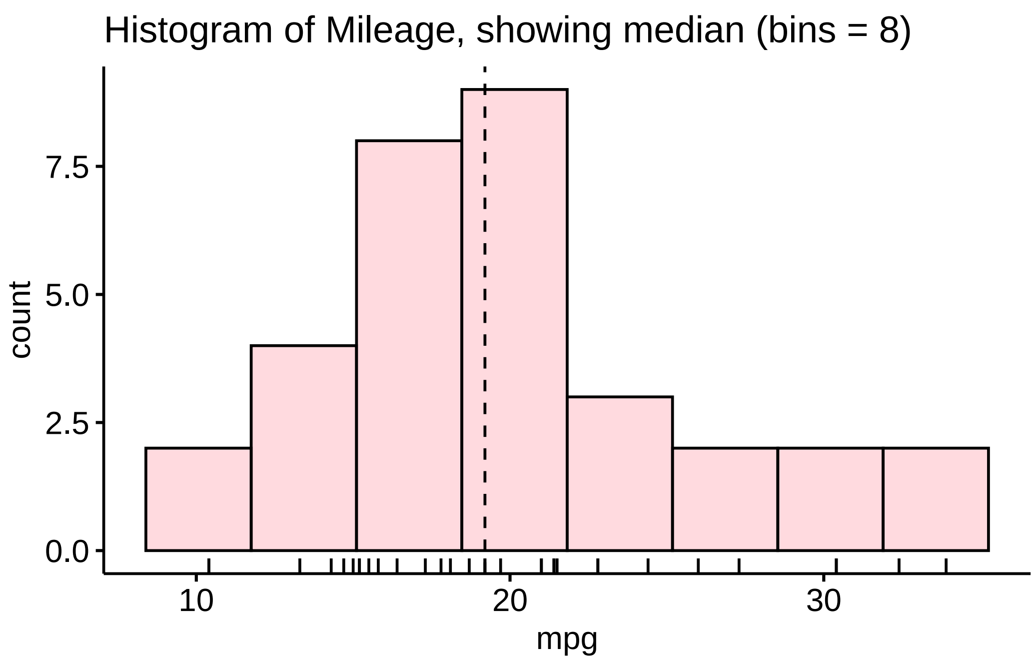
8. **Discussion:**

- Here, we're invoking the `gghistogram()` function from `ggpubr` to craft a histogram. The data source is specified as `tb`, and the variable of interest is `mpg`.
- `x = "mpg"`: This denotes the variable from tb we're visualizing.
- `binwidth = 4`: Each bin in the histogram will span a range of 4 units of `mpg`.
- `add = "mean"`: Superimposes the mean of `mpg` on the histogram.
- `rug = TRUE`: Includes a rug plot at the base, which displays individual data points.
- `color = "black"`: The outline of the bars will be in black.
- `fill = "lightpink"`: Bars in the histogram will be filled with a light pink shade.
- `title =`: Gives a descriptive title to the histogram.

- In sum, we're visualizing the distribution of the mpg variable from the `tb` dataset as a light pink histogram, emphasized with black borders, with a bin width of 4 units. We've also marked the mean value and showcased individual data points as a rug plot beneath the histogram. [2]

9. Recreating a histogram with 8 bins, using package `ggpubr`:

```
# Load the ggpubr package
library(ggpubr)

# Create a histogram of the 'mpg' variable from the tb dataset
gghistogram(tb,
            x = "mpg",  # Specify 'mpg' for the x-axis
            bins = 8,  # Set the number of bins to 8
            add = "median",  # Add a line indicating the median
            rug = TRUE,  # rug plot to show individual data points
            color = "black",  # color the histogram bins black
            fill = "lightpink",  # fill color of the bins light pink
            title = "Histogram of Mileage, showing median (bins = 8)"
)
```



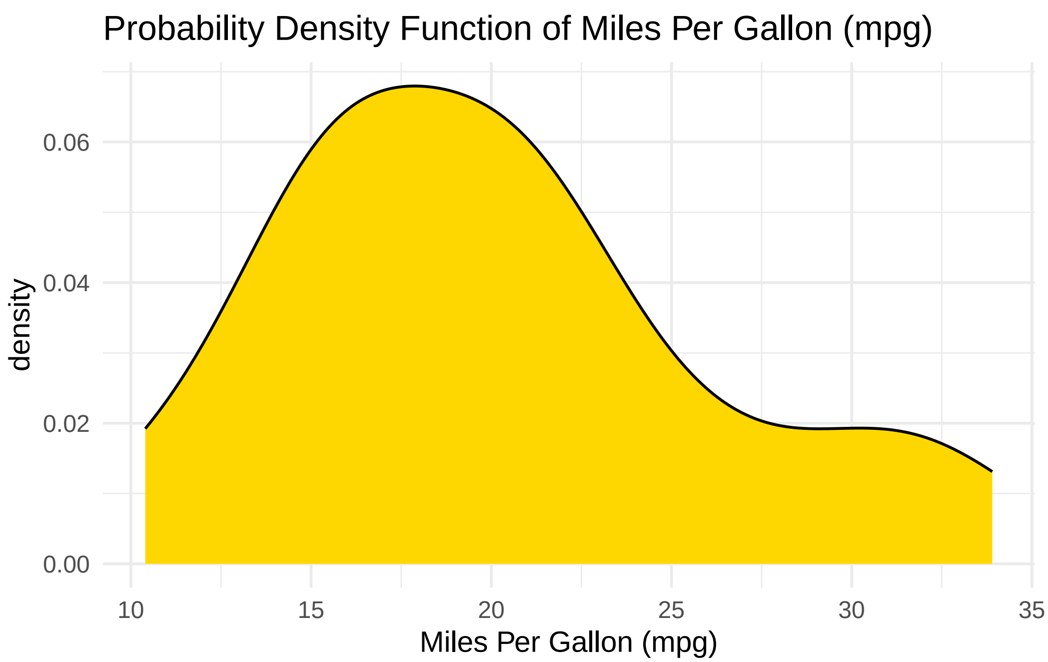
10. **Discussion:**

In essence, the difference is that this code visualizes the mpg data from the `tb` dataset as a histogram with 8 bins, set using `bins = 8`. [2]

11.3 Probability Density Function (PDF) plot using `ggplot2`

1. Recall that this type of plot shows the distribution of a single variable, and the area under the curve represents the probability of an observation falling within a particular range of values. The following code generates it using `ggplot2`:

```
# Use ggplot2 to create a density plot of mpg values from the tb dataset
ggplot(tb, aes(x = mpg)) +
  geom_density(fill = "gold") +  # density plot with gold fill color
  theme_minimal() +  # Apply a minimal theme for a clean  appearance
  labs(title = "Probability Density Function of Miles Per Gallon (mpg)",
       x = "Miles Per Gallon (mpg)",  # Label for the x-axis
       y = "density")  # Label for the y-axis
```



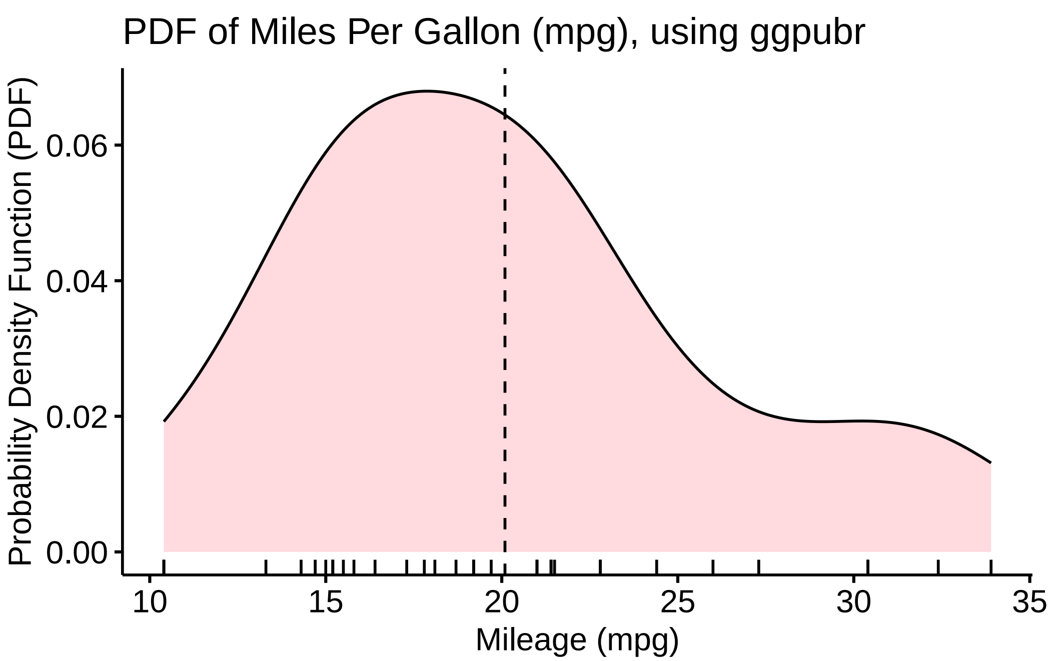
2. **Discussion**:

- We designate our data source and the aesthetic mappings using the `ggplot()` function. The aesthetic mapping for x is `mpg`.
- Subsequently, we append a density plot to our plot by using the `geom_density()` function. We fill the area under the curve by setting `fill` to "gold". [1]

11.3.0.1 PDF using `ggpubr`

3. The following R code creates a PDF of the `mpg` variable in the `tb` dataset, using the `ggdensity()` function from the `ggpubr` package.

```
# Load the ggpubr package for enhanced data visualization
library(ggpubr)

# Create a probability density function plot for 'mpg' using ggdensity
ggdensity(tb,
          x = "mpg",  # Specify 'mpg' as the variable for the plot
          add = "mean",  # Add a line indicating the mean of 'mpg'
          rug = TRUE,  # rug plot to show individual data points
          color = "black",  # color of the plot  black
          fill = "lightpink",  # Set the fill color to light pink
          title = "PDF of Miles Per Gallon (mpg), using ggpubr",
          xlab = "Mileage (mpg)",
          ylab = "Probability Density Function (PDF)"
)
```



4. **Discussion**:

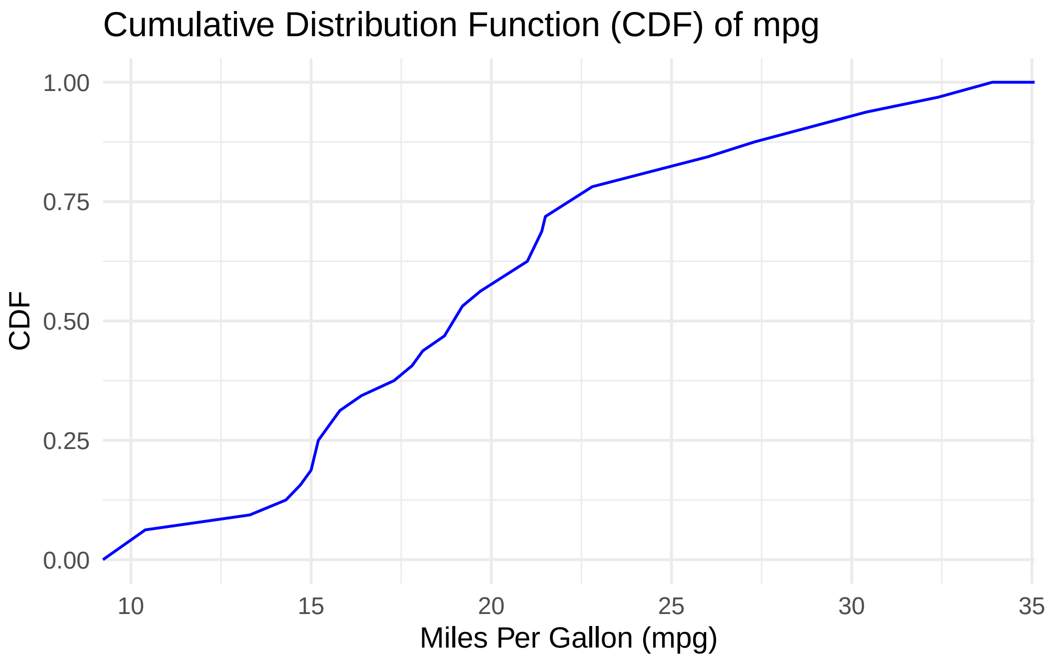
- The `ggdensity()` function from `ggpubr` package, is utilized here to visualize a probability density function (PDF) of the mpg variable from the `tb` dataset.

- `x = "mpg"`: This specifies the column `mpg` from the `tb` dataset as the variable we aim to visualize.
- `add = "mean"`: This argument ensures that a line or marker is added to the plot, indicating the mean value of the mpg data.
- `rug = TRUE`: By setting this to TRUE, a rug plot is added at the bottom, showcasing individual data points.
- `color = "black"`: This defines the border color of the density plot as black.
- `fill = "lightpink"`: This fills the interior of the density plot with a light pink color.
- `title =` : This provides a descriptive title to our plot, aiding in clarity and understanding. [3]

11.4 Cumulative Distribution Function (CDF) Plot using `ggplot2`

1. The following code generates a CDF using `ggplot2`:

```
# Load required library
library(ggplot2)

# Create a CDF (Cumulative Distribution Function) plot for mpg
ggplot(tb, aes(x = mpg)) +
  stat_ecdf(geom = "line", color = "blue") +  # plot the CDF
  labs(x = "Miles Per Gallon (mpg)", y = "CDF",
       title = "Cumulative Distribution Function (CDF) of mpg") +
  theme_minimal()  # Apply a minimalistic theme for clarity
```



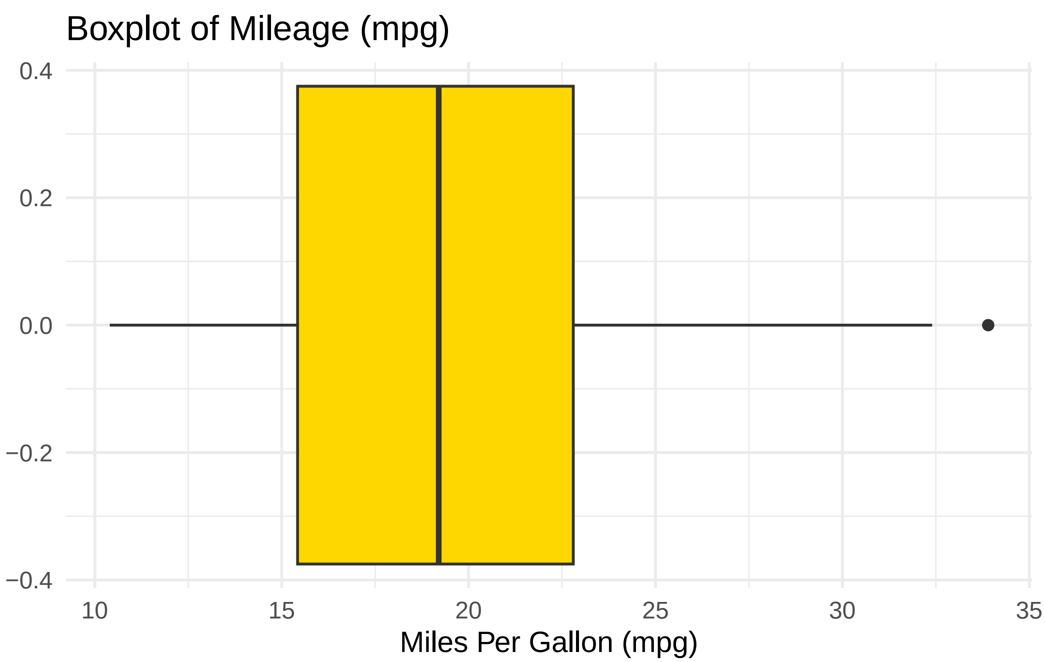
2. **Discussion**:

- Here, we're initiating a plot using the `ggplot()` function, specifying `tb` as our data source and the `mpg` column as the variable of interest. ‘
- We employ the `stat_ecdf()` function to represent the empirical cumulative distribution function (CDF) of the `mpg` data. The `geom = "line"` argument means the CDF will be displayed as a continuous line, and `color = "blue"` ensures this line is blue.
- The `labs()` function is used to define axis labels and a plot title, enhancing readability.
- By invoking `theme_minimal()`, we apply a clean and straightforward theme to our plot, which removes extraneous details and emphasizes content.
- To sum up, this code creates a plot depicting the cumulative distribution function (CDF) of the `mpg` data from the `tb` dataset. [1]

11.5 Boxplots using `ggplot2`

1. The following code generates a boxplot using `ggplot2`:

```r
# Use ggplot2 to create a boxplot for the mpg variable from the tb dataset
ggplot(tb, aes(y = mpg)) +
  geom_boxplot(fill = "gold") +  # Create a boxplot with gold color
  theme_minimal() +  # Apply a minimal theme for a clean appearance
  coord_flip() +  # Flip coordinates to horizontal boxplot
  labs(title = "Boxplot of Mileage (mpg)",  # Set the title
       y = "Miles Per Gallon (mpg)")  # Label for the y-axis
```



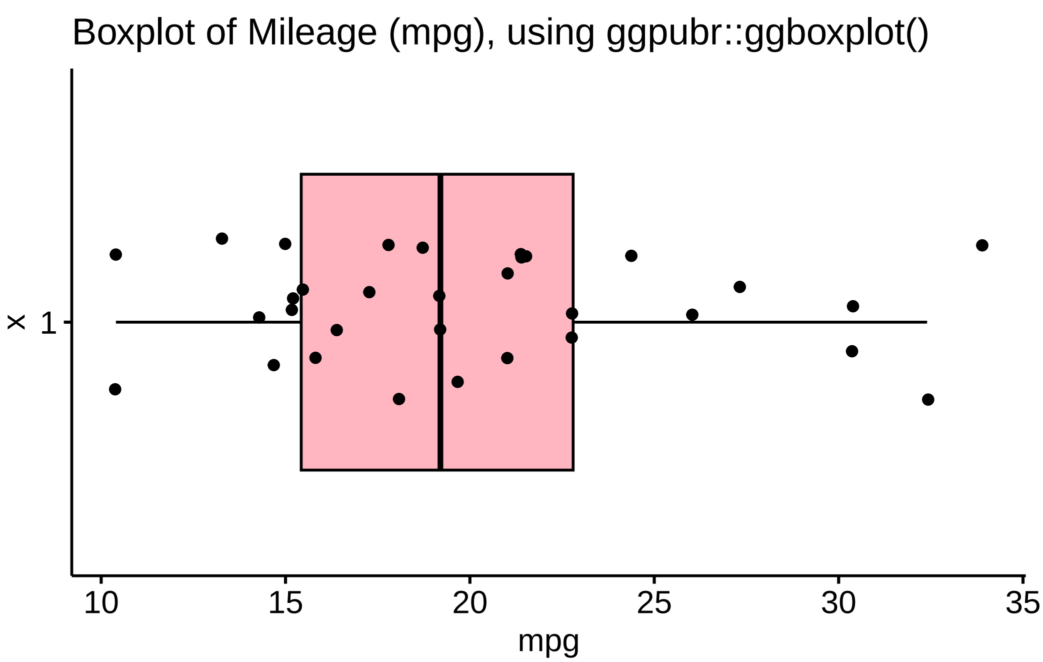
2. **Discussion:**

- `ggplot(tb, aes(y = mpg))`: Initializes a `ggplot2` plot using `tb` with `mpg` as the y-axis.
- `geom_boxplot(fill = "gold")`: Adds a gold-filled boxplot layer.
- `theme_minimal()`: Applies a clean, minimalistic theme to the plot.
- `coord_flip()`: Flips the plot to display a horizontal boxplot.
- `labs()`: Sets the plot title to "Boxplot of Mileage (mpg)" and labels the x-axis as "Miles Per Gallon (mpg)".
- In summary, this code creates a horizontal boxplot of the mpg values from the tb data frame, with the boxes filled in gold color, presented with a minimalistic theme, and labeled appropriately. [1]

11.5.1 Boxplot using `ggpubr`

1. The following code recreates the boxplot using the `ggboxplot()` function from the `ggpubr` package.

```
# Load the ggpubr package for enhanced boxplot functionalities
library(ggpubr)

# Create a boxplot for the 'mpg' using ggboxplot from ggpubr
ggboxplot(tb,
          y = "mpg",  # Specify 'mpg' as the variable for the boxplot
          orientation = "horizontal",  # Set the orientation to horizontal
          rug = TRUE,  # Add a rug plot to show individual data points
          color = "black",  # Set the color to black
          fill = "lightpink",  # Set the fill color  to light pink
          add = "jitter",  # Add jitter to display individual data points
          title = "Boxplot of Mileage (mpg), using ggpubr::ggboxplot()"
)
```



2. **Discussion:**

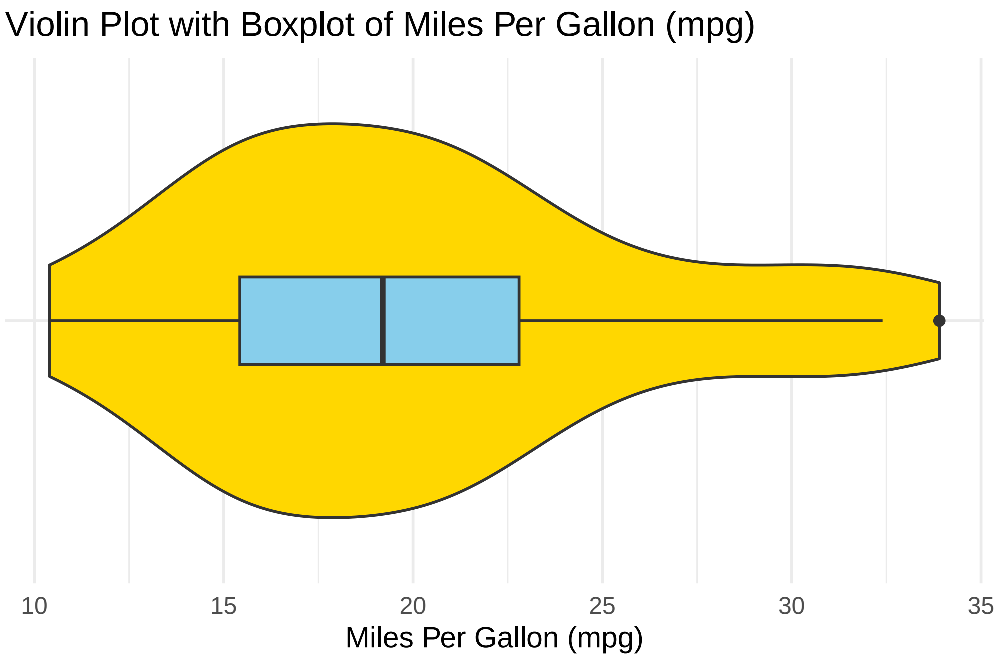
- `ggboxplot(tb, y = "mpg")`: Creates a boxplot of `mpg` values from the `tb` data frame.
- `orientation = "horizontal"`: Sets the boxplot to display horizontally.

- `rug = TRUE`: Shows a rug plot indicating the density of data points.
- `color = "black"`: Sets the boxplot's border color to black.
- `fill = "lightpink"`: Colors the inside of the boxes light pink.
- `add = "jitter"`: Adds jittered points to the boxplot for clearer data visualization.
- `title = .."`: Sets the plot's title. [2]

11.6 Violin plot using `ggplot2`

1. The following code generates a violin plot using `ggplot2`, adding a boxplot to the violin plot:

```
# Use ggplot2 to create a combined violin and boxplot for mpg values
ggplot(tb, aes(x = "", y = mpg)) +
  geom_violin(fill = "gold") +  # Create a violin plot with gold fill color
  geom_boxplot(fill = "skyblue", width = 0.2) +
  # Overlay a boxplot with sky blue fill and reduced width
  labs(x = "", # Set labels for x and y axes (x-label is empty)
       y = "Miles Per Gallon (mpg)",
       title = "Violin Plot with Boxplot of Miles Per Gallon (mpg)") +
  coord_flip() +  # Flip coordinates to make the plot horizontal
  theme_minimal()  # Apply a minimal theme for a simple appearance
```



2. Discussion:

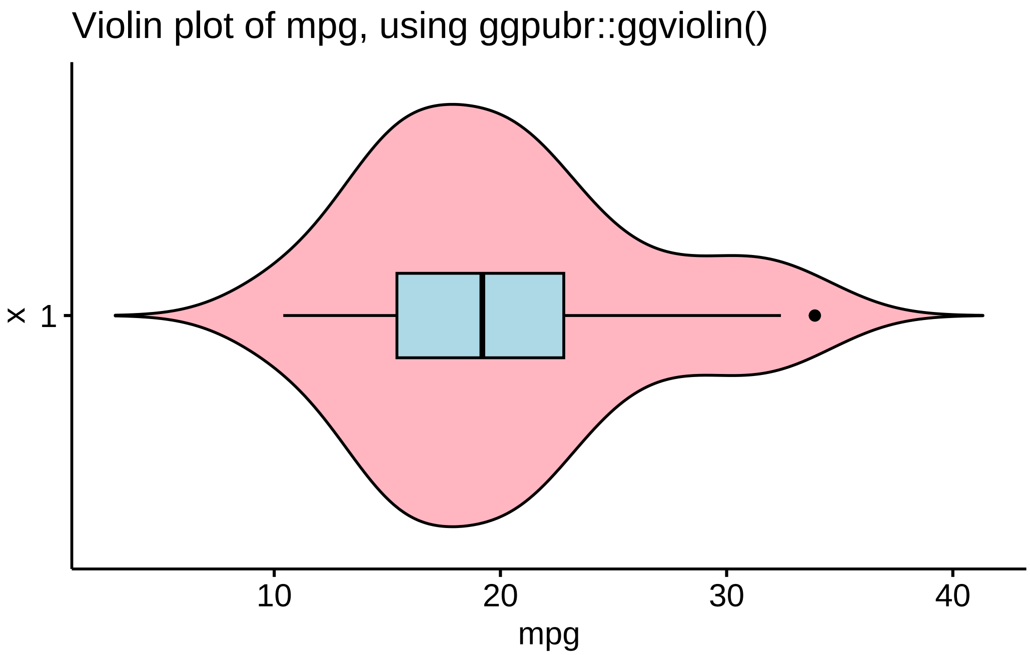
- `geom_violin()` generates the violin plot
- `geom_boxplot()` embeds a box plot within it. [1]

11.6.1 Violin plot using `ggpubr`

3. The following code recreates the violin plot using the `ggviolin()` function from the `ggpubr` package.

```
# Load the ggpubr package for enhanced data visualization
library(ggpubr)

# Create a violin plot for the 'mpg' variable using ggviolin
ggviolin(tb,
         y = "mpg",  # Specify 'mpg' as the variable for the violin plot
         orientation = "horizontal",  # Set the orientation to horizontal
         rug = TRUE,  # Include a rug plot to show individual data points
         color = "black",  # Outline color of the violin  black
         fill = "lightpink",  # Fill color of the violin light pink
         add = "boxplot", add.params = list(fill = "lightblue"),
         # Add a boxplot inside the violin plot with light blue fill
```

```
        title = "Violin plot of mpg, using ggpubr::ggviolin()"
)
```



4. **Discussion:**

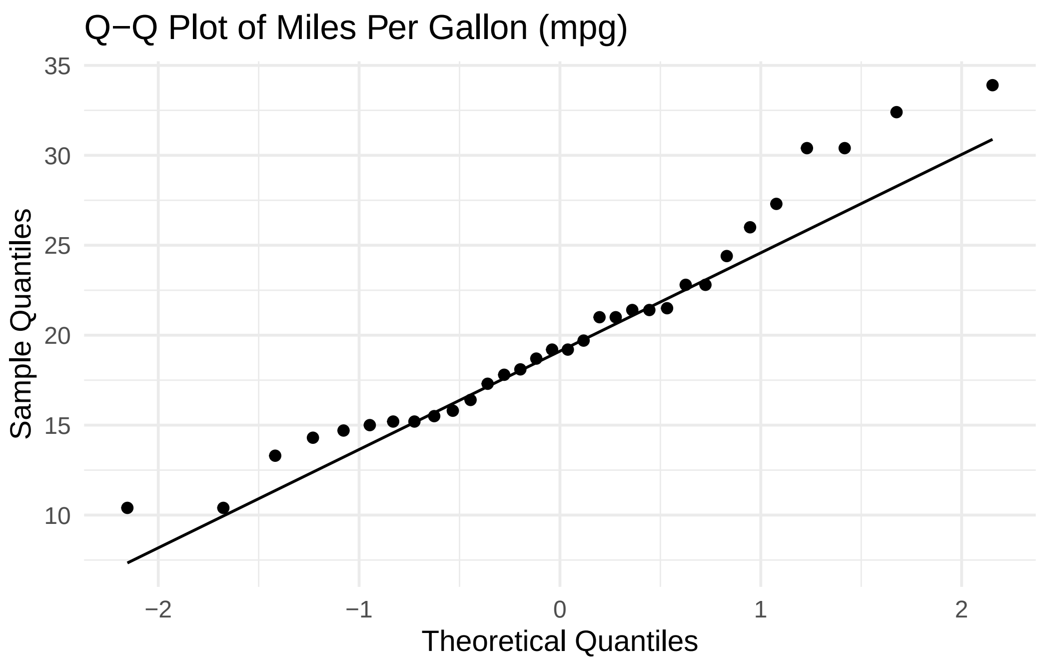
- `tb`: This refers to the dataset we're using. The dataset should have a variable named "mpg" as we've specified it in the next parameter.
- `y = "mpg"`: This indicates that we want the "mpg" variable (Miles Per Gallon) from the `tb` dataset to be plotted on the y-axis.
- `orientation = "horizontal"`: This parameter sets the orientation of the violin plot to be horizontal.
- `rug = TRUE`: This adds a "rug" to the plot, which essentially places small vertical bars (or ticks) at the actual data points along the axis.
- `color = "black"`: The edge color of the violin plot is set to black.
- `fill = "lightpink"`: This sets the main color inside the violin plot to light pink.
- `add = "boxplot"`: This specifies that a boxplot should be added inside the violin plot. A boxplot provides a summary of the distribution, showing the median, quartiles, and potential outliers.
- `add.params = list(fill = "lightblue")`: This further customizes the added boxplot by setting its fill color to light blue.

- `title = "Violin plot of Miles Per Gallon (mpg), using ggpubr::ggviolin()"`: This sets the title of the plot.
- In summary, the code generates a horizontal violin plot for the "mpg" variable from the `tb` dataset. The violin plot showcases the distribution of the "mpg" variable, colored in light pink with a light blue boxplot added inside. The rug adds an additional layer of visualization, showing the actual data points along the axis. [2]

11.7 Quantile-Quantile (Q-Q) Plots using `ggplot2`

1. Recall that a Q-Q plot is a graphical method for comparing two probability distributions by plotting their quantiles against each other. The following code generates a Quantile-Quantile (Q-Q) plot using `ggplot2`. [1]

```
# Use ggplot2 to create a Q-Q plot for the mpg variable
ggplot(tb, aes(sample = mpg)) +
  stat_qq() +  # Create a Q-Q plot
  stat_qq_line() +  # Add a line to the Q-Q plot for reference
  labs(x = "Theoretical Quantiles",  # Label for the x-axis
       y = "Sample Quantiles",  # Label for the y-axis
       title = "Q-Q Plot of Miles Per Gallon (mpg)") +
  theme_minimal()  # Apply a minimal theme for a clean appearance
```



2. **Discussion:**

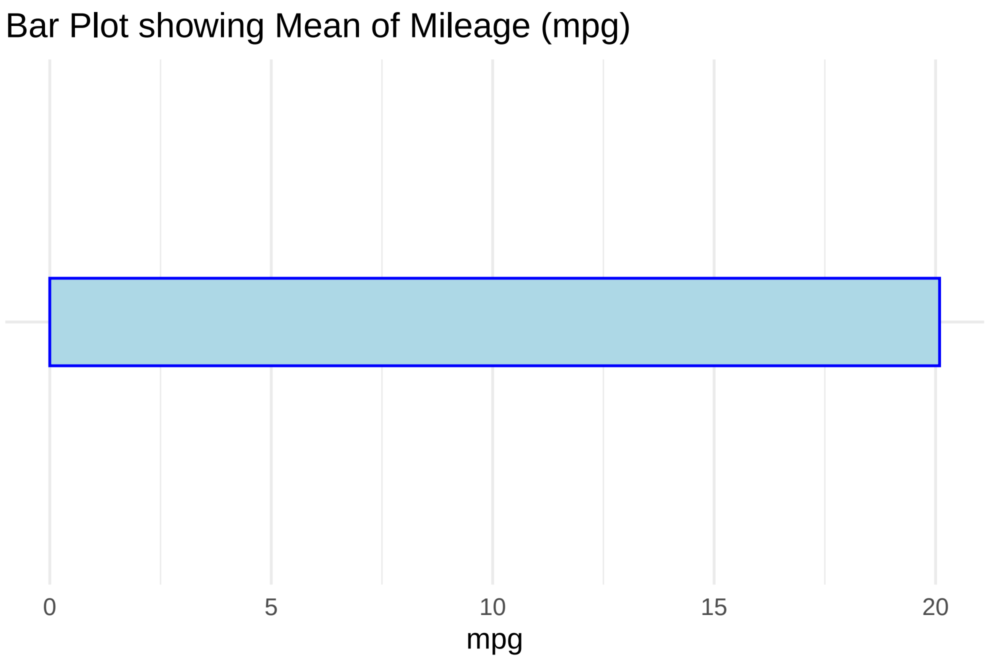
- `ggplot(tb, aes(sample = mpg))` : Here, we initiate a `ggplot` graphic using the `tb` dataset and set the aesthetic (`aes`) to the "mpg" variable. In the context of the `stat_qq()` function (which we'll come to shortly), the `sample` aesthetic specifies the data variable for which we want to create the Q-Q plot.
- `stat_qq()` : This function adds the Q-Q plot points to the graphic. The x-axis of this plot represents the quantiles from a theoretical distribution (often the standard normal distribution), and the y-axis represents the quantiles from our sample data ("mpg").
- `stat_qq_line()` : This function adds a reference line to the Q-Q plot, which represents the expected line if the sample comes from the specified distribution (again, often the standard normal distribution). If our data points lie roughly on this line, it suggests that the data follows the theoretical distribution.
- In summary, this code produces a Q-Q plot for the "mpg" variable from the `tb` dataset. The plot compares the quantiles of "mpg" against the quantiles of a theoretical distribution, often the standard normal distribution. [1]

11.8 Bar Plot visualizing the Mean and SD using `ggplot2`

1. We can create a Bar Plot to visualize the mean of a continuous variable. [1]

```
# Create a summary data frame
mpgSummary <- tb %>%
  summarise(
    MeanMpg = mean(mpg, na.rm = TRUE),  # Calculate the mean of mpg
    SDMpg = sd(mpg, na.rm = TRUE)  # Calculate the standard deviation
  )

# Create a bar plot visualizing the mean
ggplot(mpgSummary,
       aes(y = "",
           x = MeanMpg)) +
  geom_bar(stat = "identity",  # Use bars to represent values
           fill = "lightblue",
           color = "blue",
           width = 0.2  # Set bar color and width
           ) +
  labs(x = "mpg",
       y = "",  # Set x-axis label to 'mpg' with y-axis label empty
```

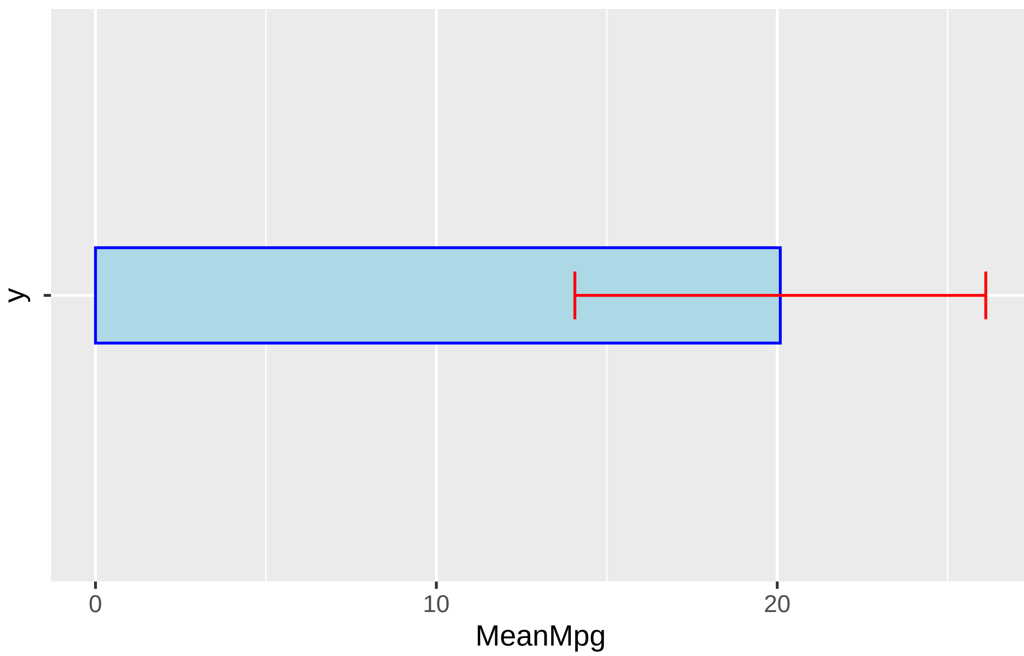
```r
      title = "Bar Plot showing Mean of Mileage (mpg)") +
  theme_minimal()  # a minimal theme for a cleaner look
```



2. We can extend this to create a Bar Plot with Error Bars to visualization the Mean and the Standard Deviation of a continuous variable. [1]

```r
# Create a summary data frame
mpgSummary <- tb %>%
  summarise(
    MeanMpg = mean(mpg, na.rm = TRUE),  # Calculate the mean of mpg
    SDMpg = sd(mpg, na.rm = TRUE)  # Calculate the standard deviation
  )

# Create a bar plot visualizing the mean and error bars visualizing the SD
ggplot(mpgSummary,
       aes(y = "",
           x = MeanMpg)) +
  geom_bar(stat = "identity",
           fill = "lightblue",
           color = "blue",
           width = 0.2  # Create a bar to represent the mean mpg
           ) +
  geom_errorbar(aes(xmin = MeanMpg - SDMpg,
```

```
                   xmax = MeanMpg + SDMpg),
               # Set the lower, upper limit of the error bar
               color = "red",  # Color the error bars red
               width = 0.1)  # Set the width of the error bars
```



3. **Discussion**

Summary Data Frame (`mpgSummary`) Creation:

- **`mpgSummary <- tb %>% summarise()`**: This line is initializing the creation of `mpgSummary` data frame using the data in `tb` (which is assumed to contain the `mtcars` dataset) and the `summarise` function from the `dplyr` package.
- **`MeanMpg = mean(mpg, na.rm = TRUE)`**: Calculates the mean of the `mpg` column, ignoring any NA values, and creates a new column `MeanMpg` in `mpgSummary` to store this value.
- **`SDMpg = sd(mpg, na.rm = TRUE)`**: Calculates the standard deviation of the `mpg` column, ignoring any NA values, and creates a new column `SDMpg` in `mpgSummary` to store this value.

Bar Plot with Error Bars Creation:

- **`ggplot(mpgSummary, aes(y = "", x = MeanMpg)) +`**: Initializes the creation of a ggplot object using the `mpgSummary` data frame and sets the aesthetic mappings where `x` is mapped to `MeanMpg` and `y` is set to an empty string.

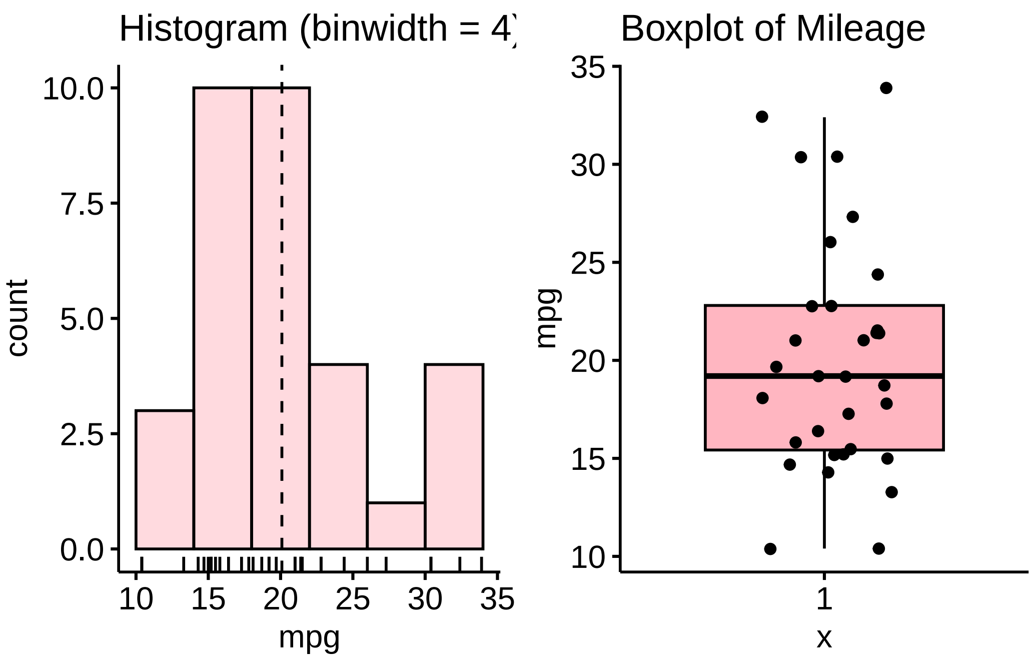
- **geom_bar(stat = "identity", fill = "lightblue", color = "blue", width = 0.3)**: Adds a bar geometry to represent the mean mpg (`MeanMpg`). The bar is colored light blue with a blue border and has a width of 0.3.
- **geom_errorbar(aes(xmin = MeanMpg - SDMpg, xmax = MeanMpg + SDMpg), color = "red", width = 0.3)**: Adds error bars to represent the variability of mpg. The error bars extend from `MeanMpg - SDMpg` to `MeanMpg + SDMpg` and are colored red with a width of 0.3.
- **labs(x = "mpg", y = "", title = "Bar Plot showing (Mean +/- SD) of Mileage (mpg)")**: Adds labels to the plot, with "mpg" as the x-axis label, no y-axis label, and a title indicating that the plot shows the mean and standard deviation of mileage.
- **theme_minimal()**: Applies a minimal theme to the plot for a clean and uncluttered appearance.
- The end result of executing this code is a bar plot visualizing the mean of the `mpg` column and error bars representing one standard deviation above and below the mean, giving a sense of the variability of the miles per gallon in the `mtcars` dataset.

11.9 Combining Plots using `ggarrange()`

1. The following code showcases two plots side-by-side. [2]

```
# Load necessary libraries
library(ggplot2)
library(ggpubr)

# Create a histogram of the 'mpg' variable using gghistogram
PlotHist <- gghistogram(tb,
            x = "mpg",  # Set 'mpg' as the variable for the histogram
            binwidth = 4,  # Define the width of the bins
            add = "mean",  # Add a line indicating the mean of 'mpg'
            rug = TRUE,  # Include a rug plot to show data points
            color = "black",  # Set the color of the histogram to black
            fill = "lightpink",  # Set the fill color of the histogram
            title = "Histogram (binwidth = 4)"  # Title of the histogram
)

# Create a box plot of the 'mpg' variable using ggboxplot
PlotBox <- ggboxplot(tb,
            y = "mpg",  # Set 'mpg' as the variable for the box plot
```

```r
            rug = TRUE,  # Include a rug plot to show individual data points
            color = "black",  # Set the color of the box plot to black
            fill = "lightpink",  # Set the fill color of the box plot
            add = "jitter",  # Add jitter to show individual data points
            title = "Boxplot of Mileage"
)

# Combine the plots using ggarrange()
combined_plot <- ggarrange(PlotHist, PlotBox, ncol = 2)
# Arrange the histogram and box plot side by side

# Display the combined plot
print(combined_plot)
```



2. **Discussion:**

- We create a histogram and a boxplot using the same code discussed above.
- `ggarrange(PlotHist, PlotBox, ncol = 2)` : This code helps us combine multiple plots into one. Here, `PlotHist` (the histogram) and `PlotBox` (the boxplot) are arranged side by side, as indicated by `ncol = 2`.
- In essence, this code facilitates a side-by-side comparison of the distribution of the `mpg` variable from the `tb` dataset using two different types of visualizations: a histogram and a boxplot. [1]

11.10 Summary of Chapter 11 – Univariate Continuous data (Part 2 of 2)

In this chapter, we explore how to visualize univariate continuous data using the `ggplot2` package in R. We use the `mtcars` data set, converting it to a tibble called `tb` for easier manipulation.

The visualization methods we cover include histograms, density plots (Probability Density Function and Cumulative Density Function), box plots, bee swarm plots, violin plots, and Q-Q plots. These are created using functions like `geom_histogram()`, `geom_density()`, `geom_boxplot()`, `geom_beeswarm()`, `geom_violin()`, `stat_qq()`, and `stat_qq_line()`.

For the histogram, we can adjust bin width, color, and number of bins, or define custom bin ranges. The density plots provide a visual representation of the distribution of a variable, and we can color the area under the curve. To create the CDF plot, we first arrange our data and calculate the cumulative distribution, which is plotted as a line graph. For the violin plot, we show how to add a box plot within the violin for additional information. Finally, we explore Q-Q plots, which compare the quantiles of our data to a theoretical distribution, useful for assessing if the data follows a certain theoretical distribution.

11.11 References

ggplot2:

[1] Wickham, H. (2016). ggplot2: Elegant Graphics for Data Analysis. Springer-Verlag New York. Retrieved from https://ggplot2.tidyverse.org

Wickham, H., & Grolemund, G. (2016). R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. O'Reilly Media.

Wickham, H. (2020). ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics (Version 3.3.2) [Computer Software]. Retrieved from https://CRAN.R-project.org/package=ggplot2

Wickham, H., et al. (2020). dplyr: A Grammar of Data Manipulation (Version 1.0.2) [Computer Software]. Retrieved from https://CRAN.R-project.org/package=dplyr

Wilkinson, L. (2005). The Grammar of Graphics (2nd ed.). Springer-Verlag.

Wickham, H., et al. (2020). tibble: Simple Data Frames (Version 3.0.3) [Computer Software]. Retrieved from https://CRAN.R-project.org/package=tibble

ggpubr:

[2] Kassambara, A. (2023). ggpubr: 'ggplot2' Based Publication Ready Plots. R Package Version 0.6.0. Retrieved from https://rpkgs.datanovia.com/ggpubr/.

ggthemes:

[3] Arnold, J.B. (2020). ggthemes: Extra Themes, Scales and Geoms for 'ggplot2' (Version 4.2.0) [Computer Software]. Retrieved from https://CRAN.R-project.org/package=ggthemes

ggbeeswarm:

[4] Eklund, A. (2020). ggbeeswarm: Categorical Scatter (Violin Point) Plots. R Package Version 0.6.0. Retrieved from https://CRAN.R-project.org/package=ggbeeswarm

12 Bivariate Continuous data (Part 1 of 4)

Chapter 12.

1. This chapter explores **Categorical** x **Continuous** data. It explains how to summarize and visualize *bivariate continuous data across categories*. Here, we delve into the intersection of continuous data and categorical variables, examining how continuous data can be split, summarized, and compared across different levels of one or more categorical variables.

2. We bring to light methods for generating statistics per group and data manipulation techniques. This includes processes like grouping, filtering, and summarizing continuous data, contingent on categorical variables. We visualize such data by creating juxtaposed box plots, segmented histograms, and density plots that reveal the distribution of continuous data across varied categories.

3. **Data**: Suppose we run the following code to prepare the `mtcars` data for subsequent analysis and save it in a tibble called `tb`. [1]

```
# Load the required libraries, suppressing annoying startup messages
library(dplyr, quietly = TRUE, warn.conflicts = FALSE)
library(tibble, quietly = TRUE, warn.conflicts = FALSE)
library(knitr) # For formatting tables

# Read the mtcars dataset into a tibble called tb
data(mtcars)
tb <- as_tibble(mtcars)

# Convert relevant columns into factor variables
tb$cyl <- as.factor(tb$cyl) # cyl = {4,6,8}, number of cylinders
tb$am <- as.factor(tb$am) # am = {0,1}, 0:automatic, 1: manual transmission
tb$vs <- as.factor(tb$vs) # vs = {0,1}, v-shaped engine, 0:no, 1:yes
tb$gear <- as.factor(tb$gear) # gear = {3,4,5}, number of gears

# Directly access the data columns of tb, without tb$mpg
attach(tb)
```

12.1 Summarizing Continuous Data

12.1.1 Across one Category

- We review the use of the inbuilt functions (i) `aggregate()`; (ii) `tapply()`; and the function (iii) `describeBy()` from package `pysch`, to summarize continuous data split across a category.

1. **Using `aggregate()`**

- We use the `aggregate()` function to investigate the bivariate relationship between mileage (`mpg`) and number of cylinders (`cyl`). The following code displays a summary table showing the average mileage of the cars broken down by number of cylinders (`cyl` = 4, 6, 8) using `aggregate()`. [1] [2]

```
# Aggregating 'mpg' column by the 'cyl' column.
# The aggregation function used is 'mean' to calculate the average.
A0 <- aggregate(tb$mpg,
                by = list(tb$cyl),
                FUN = mean)

# Renaming the column names of the aggregated data frame 'A0'.
# 'Cylinders' for the first column and 'Mean_mpg' for the second.
names(A0) <- c("Cylinders", "Mean_mpg")

# Displaying the aggregated data using 'kable' from the knitr package.
# Setting the number of digits to 2 for formatting and adding a caption.
kable(A0, digits=2,
      caption = "Mean of Mileage (mpg) by Cylinder (cyl=4,6,8)")
```

Table 12.1: Mean of Mileage (mpg) by Cylinder (cyl=4,6,8)

Cylinders	Mean_mpg
4	26.66
6	19.74
8	15.10

2. **Discussion**:

- The first argument in `aggregate()` is the data vector `tb$mpg`.
- The second argument, `by`, denotes a list of variables to group by. Here, we have supplied

tb$cyl, since we wish to partition our data based on the unique values of cyl.

- The third argument, FUN, is the function we want to apply to each subset of data. We are using mean here, calculating the average mpg for each unique cyl value. We can alternately aggregate based on a variety of statistical functions including sum, median, min, max, sd, var, length, IQR.
- The output of aggregate() is saved in a new tibble named agg. We utilize the names() function to rename the columns and display agg. [1]

3. **Using tapply()**

- The tapply() function is another convenient tool to apply a function to subsets of a vector, grouped by some factors.

```
# Applying the 'mean' function to the 'mpg' column,
# grouping the data by the 'cyl' column
# The 'tapply' function applies a function to each cell.
A1 <- tapply(tb$mpg,
             tb$cyl,
             mean)
print(A1)
```

```
       4        6        8
26.66364 19.74286 15.10000
```

4. **Discussion**:

- In this code, tapply(tbmpg, tbcyl, mean) calculates the average miles per gallon (mpg) for each unique number of cylinders (cyl) within the tb tibble.
- tb$mpg represents the vector to which we want to apply the function.
- tb$cyl serves as our grouping factor.
- mean is the function that we're applying to each subset of our data.
- The result will be a vector where each element is the average mpg for a unique number of cylinders (cyl), as determined by the unique values of tb$cyl. [1]

5. **Using describeBy() from package psych**

- The describeBy() function, part of the psych package, can be used to compute descriptive statistics of a numeric variable, broken down by levels of a grouping variable. [3]

```r
# Loading the 'psych' package
library(psych)

# Using the 'describeBy' from the 'psych' package for descriptive
# statistics of 'mpg' grouped by 'cyl'.
A2 <- describeBy(mpg, cyl)
print(A2)
```

```
 Descriptive statistics by group
group: 4
   vars  n  mean   sd median trimmed  mad  min  max range skew kurtosis   se
X1    1 11 26.66 4.51     26   26.44 6.52 21.4 33.9  12.5 0.26    -1.65 1.36
------------------------------------------------------------
group: 6
   vars n  mean   sd median trimmed  mad  min  max range  skew kurtosis   se
X1    1 7 19.74 1.45   19.7   19.74 1.93 17.8 21.4   3.6 -0.16    -1.91 0.55
------------------------------------------------------------
group: 8
   vars  n mean   sd median trimmed  mad  min  max range  skew kurtosis   se
X1    1 14 15.1 2.56   15.2   15.15 1.56 10.4 19.2   8.8 -0.36    -0.57 0.68
```

6. **Discussion**:

- `describeBy(mpg, cyl)` computes descriptive statistics of miles per gallon `mpg` variable, broken down by the unique values in the number of cylinders (`cyl`).
- It calculates statistics such as the mean, sd, median, for `mpg`, separately for each unique number of cylinders (`cyl`). [3]

12.1.2 Across two Categories

- We extend the above discussion and study how to summarize continuous data split across **two** categories.
- We review the use of the inbuilt functions (i) `aggregate()` and the function (ii) `describeBy()` from package `pysch`. While the `tapply()` function can theoretically be employed for this task, the resulting code tends to be long and lacks efficiency. Therefore, we opt to exclude it from practical use.

1. Using **`aggregate()`**

- Distribution of Mileage (`mpg`) by Cylinders (`cyl` = {4,6,8}) and Transmisson Type (`am` = {0,1}) [1]

```
# Aggregating 'mpg' column by both 'cyl' and 'am' columns.
# The aggregation function used is 'mean'.
BO <- aggregate(tb$mpg,
                by = list(tb$cyl, tb$am),
                FUN = mean)

# Renaming the column names of the aggregated data frame 'BO'.
# 'Cylinders' 'Transmission'  'Mean_mpg' .
names(BO) <- c("Cylinders", "Transmission", "Mean_mpg")

# Displaying the aggregated data using 'kable' from the knitr package.
# Setting the number of digits to 2 for formatting and adding a caption.
kable(BO,
      digits=2,
      caption = "Mean of Mileage (mpg) by Cylinders and Transmission")
```

Table 12.2: Mean of Mileage (mpg) by Cylinders and Transmission

Cylinders	Transmission	Mean_mpg
4	0	22.90
6	0	19.12
8	0	15.05
4	1	28.08
6	1	20.57
8	1	15.40

2. **Discussion**:

- In our code, the first argument of `aggregate()` is `tb$mpg`, indicating that we want to perform computations on the `mpg` variable.
- The by argument is a list of variables by which we want to group our data, specified as `list(tb$cyl, tb$am)`. This means that separate computations are done for each unique combination of `cyl` and `am`.
- The `FUN` argument indicates the function to be applied to each subset of our data. Here, we use mean, meaning that we compute the mean mpg for each group. [1]

3. Using `aggregate()` for multiple continuous variables: Consider this extension of the above code for calculating the mean of three variables - `mpg`, `wt`, and `hp`, grouped by

both am and cyl variables:

- Distribution of Mileage (`mpg`), Weight (`wt`), Horsepower (`hp`) by Cylinders (`cyl` = {4,6,8}) and Transmisson Type (`am` = {0,1})

```
# Aggregating mpg, wt, hp by 'am' and 'cyl' columns.
# The 'mean' function is used to calculate the average
B1 <- aggregate(list(mpg, wt, hp),
                by = list(am, cyl),
                FUN = mean)

# Renaming the column names of the aggregated data frame
names(B1) <- c("Transmission", "Cylinders",
               "Mean_mpg", "Mean_wt", "Mean_hp")

# Displaying the aggregated data using 'kable' from the knitr package.
# Formatting numbers to two decimal places and adding a descriptive caption.
kable(B1,
      digits=2,
      caption = "Mean of mpg, wt, hp by am, cyl")
```

Table 12.3: Mean of mpg, wt, hp by am, cyl

Transmission	Cylinders	Mean_mpg	Mean_wt	Mean_hp
0	4	22.90	2.94	84.67
1	4	28.08	2.04	81.88
0	6	19.12	3.39	115.25
1	6	20.57	2.76	131.67
0	8	15.05	4.10	194.17
1	8	15.40	3.37	299.50

4. **Discussion**:

- In this code, the `aggregate()` function takes a list of the three variables as its first argument, indicating that the mean should be calculated for each of these variables separately within each combination of `am` and `cyl`.

- The sequence of the categorizing variables also varies - initially, the data is grouped by `cyl`, followed by a subdivision based on `am`.

5. Using **`aggregate()`** with multiple functions: Consider an extension of the above code for calculating the mean and the SD of `mpg`, grouped by both `am` and `cyl` factor variables:

- Distribution of Mileage (`mpg`), by Cylinders (`cyl` = {4,6,8}) and Transmission Type (`am` = {0,1}) [1]

```
# Calculating the mean of 'mpg' grouped by 'cyl' and 'am'.
agg_mean <- aggregate(tb$mpg,
                      by = list(tb$cyl, tb$am),
                      FUN = mean)

# Calculating the standard deviation (sd) of 'mpg'  by 'cyl' and 'am'.
agg_sd <- aggregate(tb$mpg,
                    by = list(tb$cyl, tb$am),
                    FUN = sd)

# Calculating the median of 'mpg' for each group of 'cyl' and 'am'.
agg_median <- aggregate(tb$mpg,
                        by = list(tb$cyl, tb$am),
                        FUN = median)

# Merging the mean and sd data frames based on 'cyl' and 'am'.
B2 <- merge(agg_mean, agg_sd,
            by = c("Group.1", "Group.2"))

# Merging the above result
B2 <- merge(B2, agg_median,
            by = c("Group.1", "Group.2"))

# Renaming the columns of the merged data
names(B2) <- c("Cylinders", "Transmission",
               "Mean_mpg", "SD_mpg", "Median_mpg")

# Displaying the merged data
kable(B2,
      digits=2, caption = "Mean, SD, Median of Mileage (mpg)
      by Cylinders (cyl=4,6,8) and Transmission (am=0,1)")
```

Table 12.4: Mean, SD, Median of Mileage (mpg) by Cylinders (cyl=4,6,8) and Transmission (am=0,1)

Cylinders	Transmission	Mean_mpg	SD_mpg	Median_mpg
4	0	22.90	1.45	22.80
4	1	28.08	4.48	28.85
6	0	19.12	1.63	18.65

Cylinders	Transmission	Mean_mpg	SD_mpg	Median_mpg
6	1	20.57	0.75	21.00
8	0	15.05	2.77	15.20
8	1	15.40	0.57	15.40

6. **Discussion**:

- We analyze our dataset to comprehend the relationships between vehicle miles per gallon (`mpg`), number of cylinders (`cyl`), and type of transmission (`am`).
- Initially, we computed the mean, standard deviation, and median of `mpg` for every unique combination of `cyl` and `am`.
- After individual computations, we combined these results into a single, comprehensive data frame called `merged_data`. This structured dataset now clearly presents the average, variability, and median of fuel efficiency segmented by cylinder count and transmission type. [1]

7. **Using `describeBy()` from package `psych`**

- The `describeBy()` function, part of the `psych` package, can be used to compute descriptive statistics of continuous variable, broken down by levels of a two categorical varaibles. Consider the following code:

```r
# Selecting specific columns ('mpg', 'wt', 'hp')
tb_columns <- tb[c("mpg", "wt", "hp")]

# Creating a list of factors ('am' and 'cyl') for grouping in describeBy.
tb_factors <- list(tb$am, tb$cyl)

# Using the 'describeBy' function to provide  descriptive statistics
# grouped by 'am' and 'cyl'.
B3 <- describeBy(tb_columns,
                 tb_factors)

print(B3)
```

```
 Descriptive statistics by group
: 0
: 4
    vars n  mean    sd median trimmed  mad   min   max range  skew kurtosis
mpg    1 3 22.90  1.45  22.80   22.90 1.93 21.50 24.40  2.90  0.07    -2.33
```

```
wt     2 3  2.94  0.41   3.15    2.94 0.06  2.46  3.19  0.73 -0.38    -2.33
hp     3 3 84.67 19.66  95.00   84.67 2.97 62.00 97.00 35.00 -0.38    -2.33
      se
mpg  0.84
wt   0.24
hp  11.35
------------------------------------------------------------
: 1
: 4
    vars n  mean    sd median trimmed   mad   min    max range  skew kurtosis
mpg    1 8 28.08  4.48  28.85   28.08  4.74 21.40  33.90 12.50 -0.21    -1.66
wt     2 8  2.04  0.41   2.04    2.04  0.36  1.51   2.78  1.27  0.35    -1.15
hp     3 8 81.88 22.66  78.50   81.88 20.76 52.00 113.00 61.00  0.14    -1.81
      se
mpg 1.59
wt  0.14
hp  8.01
------------------------------------------------------------
: 0
: 6
    vars n   mean   sd median trimmed  mad    min    max range  skew kurtosis
mpg    1 4  19.12 1.63  18.65   19.12 1.04  17.80  21.40  3.60  0.48    -1.91
wt     2 4   3.39 0.12   3.44    3.39 0.01   3.21   3.46  0.25 -0.73    -1.70
hp     3 4 115.25 9.18 116.50  115.25 9.64 105.00 123.00 18.00 -0.09    -2.33
      se
mpg 0.82
wt  0.06
hp  4.59
------------------------------------------------------------
: 1
: 6
    vars n   mean    sd median trimmed  mad    min    max range  skew kurtosis
mpg    1 3  20.57  0.75  21.00   20.57 0.00  19.70  21.00  1.30 -0.38    -2.33
wt     2 3   2.76  0.13   2.77    2.76 0.16   2.62   2.88  0.25 -0.12    -2.33
hp     3 3 131.67 37.53 110.00  131.67 0.00 110.00 175.00 65.00  0.38    -2.33
      se
mpg  0.43
wt   0.07
hp  21.67
------------------------------------------------------------
: 0
: 8
    vars  n   mean    sd median trimmed   mad    min    max range  skew
```

```
mpg    1 12  15.05  2.77  15.20   15.10  2.30  10.40  19.20  8.80 -0.28
wt     2 12   4.10  0.77   3.81    4.04  0.41   3.44   5.42  1.99  0.85
hp     3 12 194.17 33.36 180.00  193.50 40.77 150.00 245.00 95.00  0.28
    kurtosis   se
mpg    -0.96 0.80
wt     -1.14 0.22
hp     -1.44 9.63
------------------------------------------------------------
: 1
: 8
    vars n   mean    sd median trimmed   mad    min    max range skew kurtosis
mpg    1 2  15.40  0.57  15.40   15.40  0.59  15.00  15.80   0.8    0    -2.75
wt     2 2   3.37  0.28   3.37    3.37  0.30   3.17   3.57   0.4    0    -2.75
hp     3 2 299.50 50.20 299.50  299.50 52.63 264.00 335.00  71.0    0    -2.75
      se
mpg  0.4
wt   0.2
hp  35.5
```

7. **Discussion**:

- We specify a subset of the dataframe `tb` that includes only the columns of interest – `mpg`, `wt`, and `hp` and save it into a variable `tb_columns`.
- Next, we create a list, `tb_factors`, that contains the factors `am` and `cyl`.
- After that, we call the `describeBy()` function from the `psych` package. This function calculates descriptive statistics for each combination of levels of the factors `am` and `cyl` and for each of the continuous variables `mpg`, `wt`, and `hp`. [3]

12.2 Visualizing Continuous Data

Let's take a closer look at some of the most effective ways of visualizing univariate continuous data, including

(i) Bee Swarm plots;

(ii) Stem-and-Leaf plots;

(iii) Histograms;

(iv) PDF and CDF Density plots;

(v) Box plots;

(vi) Violin plots;

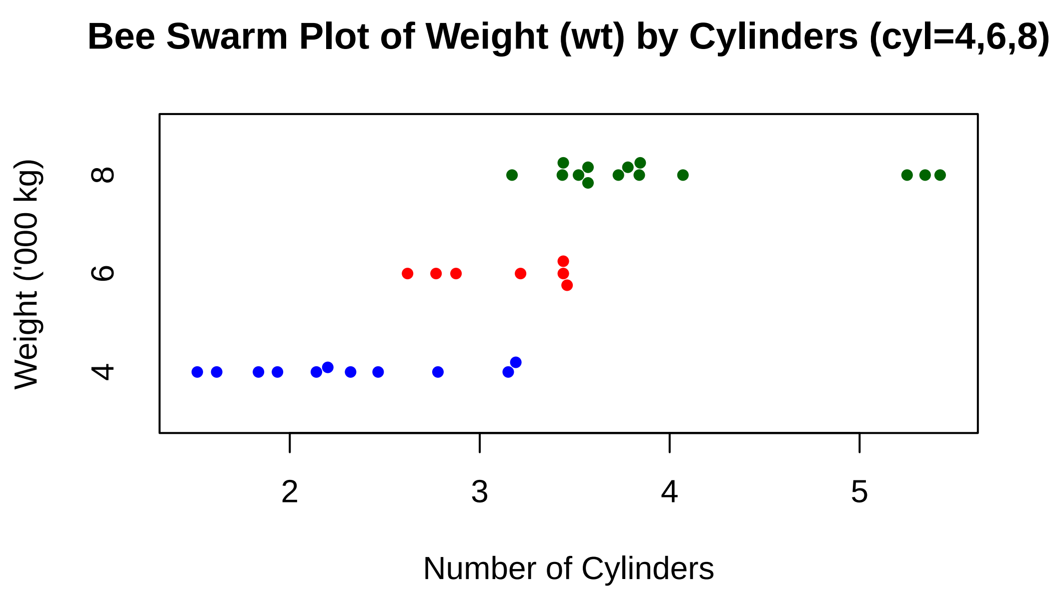
(vii) Q-Q plots.

12.2.1 Bee Swarm Plot

1. We extend a Bee Swarm plot of a *one-dimensional scatter plot* for a continuous variable, split by a categorical variable. [4]

2. Consider the following code, which generates a beeswarm plot displaying vehicle weights (`wt`) segmented by their number of cylinders (`cyl`):

```
# Loading the 'beeswarm' package
library(beeswarm)

# Creating a bee swarm plot of the 'wt' (weight) column,
# grouped by the 'cyl' (cylinders) column.
beeswarm(tb$wt ~ tb$cyl,
         main="Bee Swarm Plot of Weight (wt) by Cylinders (cyl=4,6,8)",
         xlab="Number of Cylinders",  # Label for the x-axis
         ylab="Weight ('000 kg)",     # Label for the y-axis
         pch=16,  # Type of points used in the plot (solid circle)
         cex=0.8, # Size of the points
         col=c("blue","red","darkgreen"),  # Colors for different groups
         horizontal = TRUE) # Orientation of the plot (horizontal)
```



3. **Discussion**:

- **Data**: We use tb*wt tb*cyl to specify that we want a beeswarm plot for Weight (`wt`), split by no of cylinders (`cyl`),
- **Title**: It is labeled "Bee Swarm Plot of Weight (wt) by Number of Cylinders".
- **Axes Labels**: The x-axis shows "Number of Cylinders", while the y-axis denotes "Weight ('000 kg)".
- **Data Points**: Using `pch=16`, data points appear as solid circles.
- **Size of Points**: With `cex=0.8`, these circles are slightly smaller than default.
- **Colors**: The `col` parameter assigns colors ("blue", "red", and "dark green") based on cylinder counts.
- **Orientation**: Set as horizontal with `horizontal=TRUE`.
- To summarize, this visual distinguishes vehicle weights across cylinder counts and highlights data point densities for each group. [4]

12.2.2 Stem-and-Leaf Plot across one Category

1. Suppose we wanted to visualize the distribution of a continuous variable across different levels of a categorical variable, using stem-and-leaf plots.
2. To illustrate, let us display vehicle weights (`wt`) separately for each transmission type (`am`) using stem-and-leaf plots.

```
# Selecting 'wt' and 'am' columns from the 'tb' dataframe
tb3 <- tb[, c("wt", "am")]

# Splitting the 'tb3' dataframe into subsets based on the 'am' column.
tb_split <- split(tb3, tb3$am)

# Applying a function to each subset in 'tb_split' using 'lapply'.
# Create a stem-and-leaf plot for the 'wt' column in each subset.
lapply(tb_split,
       function(x) stem(x$wt))
```

```
  The decimal point is at the |

  2 | 5
  3 | 22244445567888
  4 | 1
```

```
  5 | 334


  The decimal point is at the |

  1 | 5689
  2 | 123
  2 | 6889
  3 | 2
  3 | 6

$`0`
NULL

$`1`
NULL
```

3. **Discussion**:

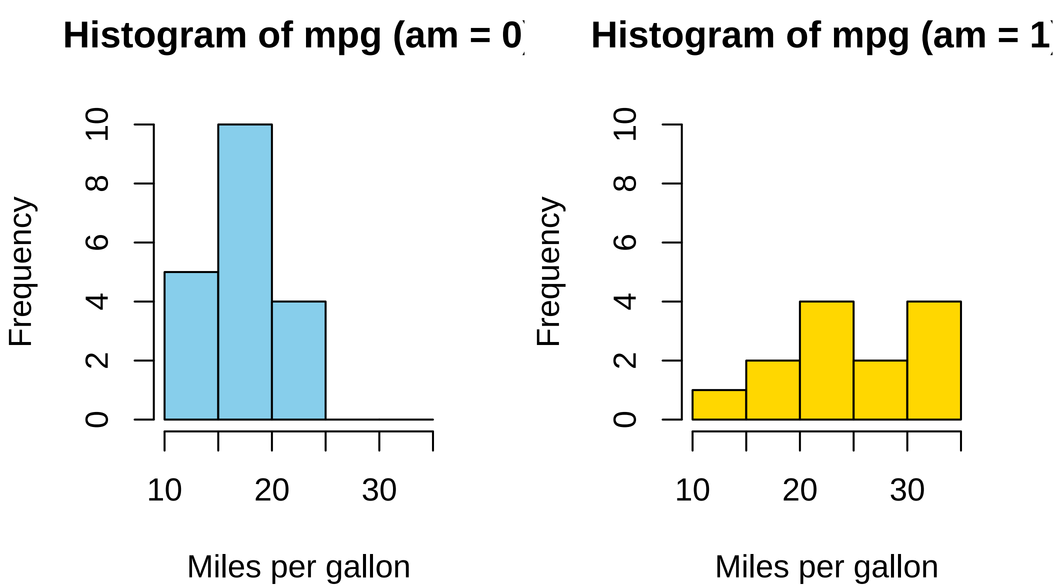
- Column Selection: The code extracts the `wt` (weight) and `am` (transmission type) columns from `tb` and saves them in `tb3`.
- Data Splitting: It then divides `tb3` into subsets based on `am` values, resulting in separate groups for each transmission type.
- Visualization: Using `lapply()`, the code generates stem-and-leaf plots for the `wt` values in each subset, showcasing weight distributions for different transmission types. In this context, it shows the distribution of vehicle weights for each transmission type (automatic and manual). [5]

12.2.3 Histograms across one Category

1. Visualizing histograms of car mileage (`mpg`) broken down by transmission (`am`=0,1)

```r
split_data <- split(tb$mpg, tb$am) # Split the data by 'am' variable
# Splitting the 'mpg' data from the 'tb' dataframe by the 'am' variable.
split_data <- split(tb$mpg, tb$am)

# Setting up the graphics layout for two plots side by side.
par(mfrow = c(1, 2))

# Defining a color vector for use in the histograms.
color_vector <- c("skyblue", "gold")
```

```r
# Creating a histogram for the subset with 'am' equals 0.
hist(split_data[[1]],
     main = "Histogram of mpg (am = 0)",
     breaks = seq(10, 35, by = 5), #  bin ranges (10-15, 15-20, etc.)
     xlab = "Miles per gallon",
     col = color_vector[1], # Applying the first color
     border = "black", # Setting the border color of the bins.
     ylim = c(0, 10)) # Setting y-axis limits.

# Creating a histogram for the subset with 'am' equals 1.
hist(split_data[[2]],
     main = "Histogram of mpg (am = 1)",
     breaks = seq(10, 35, by = 5), #  bin ranges (10-15, 15-20, etc.)
     xlab = "Miles per gallon",
     col = color_vector[2], # Applying the second color
     border = "black", # Setting the border color of the bins.
     ylim = c(0, 10)) # Setting y-axis limits.
```



In Appendix A1, we have alternative code written using a for loop

2. **Discussion**:

- We aim to visualize the distribution of the `mpg` values from the `tb` dataset based on the `am` variable, which can be either 0 or 1.

- Data Splitting: We segregate `mpg` values into two subsets using the `split` function, depending on the `am` values. In R, the double brackets [[ ]] are used to access the elements of a list or a specific column of a data frame. `split_data[[1]]` accesses the first element of the list `split_data`.

- Layout Setting: The `par` function is configured to display two plots side by side in a single row and two columns format.

3. Color Vector: We introduce a `color_vector` to assign distinct colors to each histogram for differentiation.

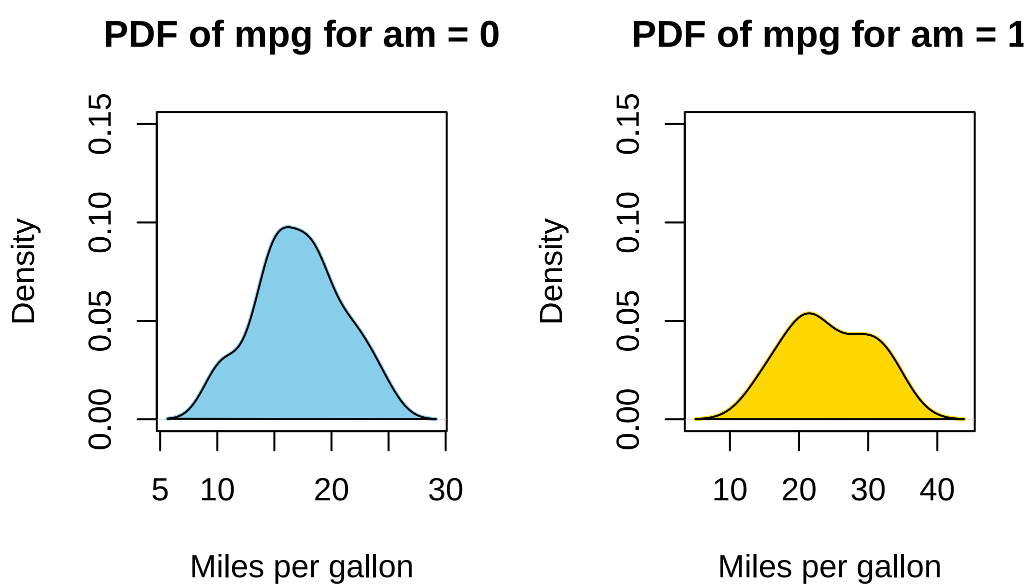
4. Histogram: Two histograms are generated, one for each `am` value (0 and 1). These histograms use various parameters like title, x-axis label, color, and y-axis limits to provide a clear representation of the data's distribution. [1]

12.2.4 Probability Density Function (PDF) across one Category

1. Visualizing Probability Density Functions (PDF) of car mileage (`mpg`) broken down by transmission (`am`=0,1) [1]

```
# Splitting 'mpg' data from the 'tb' dataframe by 'am' values.
split_data <- split(tb$mpg, tb$am)

# Setting up the graphical layout for two plots side by side.
par(mfrow = c(1, 2))

# Defining color scheme for the plots.
color_vector <- c("skyblue", "gold")

# Calculating the density for the subset with 'am' equals 0 and plotting it.
dens_0 <- density(split_data[[1]])
plot(dens_0,
     main = "PDF of mpg for am = 0",  # Title for the first plot.
     xlab = "Miles per gallon",  # X-axis label.
     col = color_vector[1],  # Applying the first color.
     border = "black",
     ylim = c(0, 0.15),  # Setting y-axis limits.
     lwd = 2)  # Line width for the density plot.
polygon(dens_0, col = color_vector[1], border = "black")
# Filling under the density curve.

# Calculating the density for the subset with 'am' equals 1 and plotting it.
dens_1 <- density(split_data[[2]])
```

```
plot(dens_1,
     main = "PDF of mpg for am = 1",  # Title for the second plot.
     xlab = "Miles per gallon",  # X-axis label.
     col = color_vector[2],  # Applying the second color.
     border = "black",
     ylim = c(0, 0.15),  # Setting y-axis limits.
     lwd = 2)  # Line width for the density plot.
polygon(dens_1, col = color_vector[2], border = "black")
```




```
# Filling under the density curve.
```

In Appendix A2, we have alternative code written using a for loop

2. **Discussion**:

- `dens_0 <- density(split_data[[1]])` calculates the density values for the subset where `am` is 0.
- The subsequent plot function visualizes the density curve, setting various parameters like the title, x-axis label, color, and line width.
- The polygon function fills the area under the density curve with the specified color, giving a shaded appearance to the plot.
- The process is repeated for the subset where `am` is 1. The code calculates the density, plots it, and then uses the polygon function to shade the area under the curve. [1]

In Appendix A3, we demonstrate how to draw overlapping PDFs on the same plot, using base R functions.

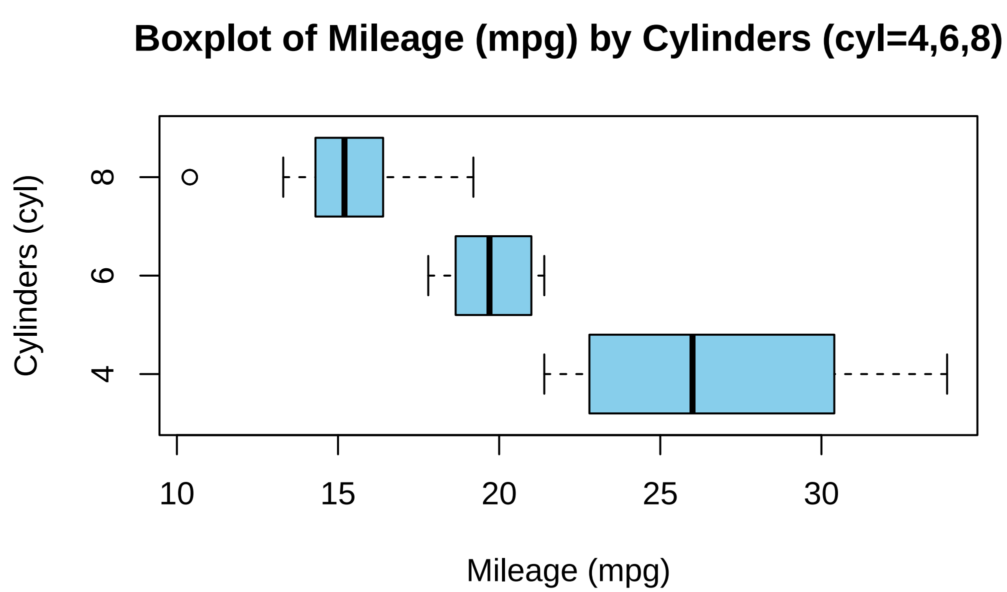
12.2.5 Cumulative Density Function (CDF) across one Category

In Appendix A4, we demonstrate how to draw a CDF, using base R functions

12.2.6 Box Plots across one Category

1. Visualizing Median using Box Plot – median weight of the cars broken down by cylinders (`cyl`=4,6,8)

```
# Creating a boxplot for 'mpg' values grouped by 'cyl'.
boxplot(mpg ~ cyl,
        main = "Boxplot of Mileage (mpg) by Cylinders (cyl=4,6,8)",
        xlab = "Mileage (mpg)",  # Label for the x-axis.
        ylab = "Cylinders (cyl)",  # Label for the y-axis.
        col = c("skyblue"),  # Color of the boxplots.
        horizontal = TRUE  # Setting the boxplot to be horizontal.
        )
```

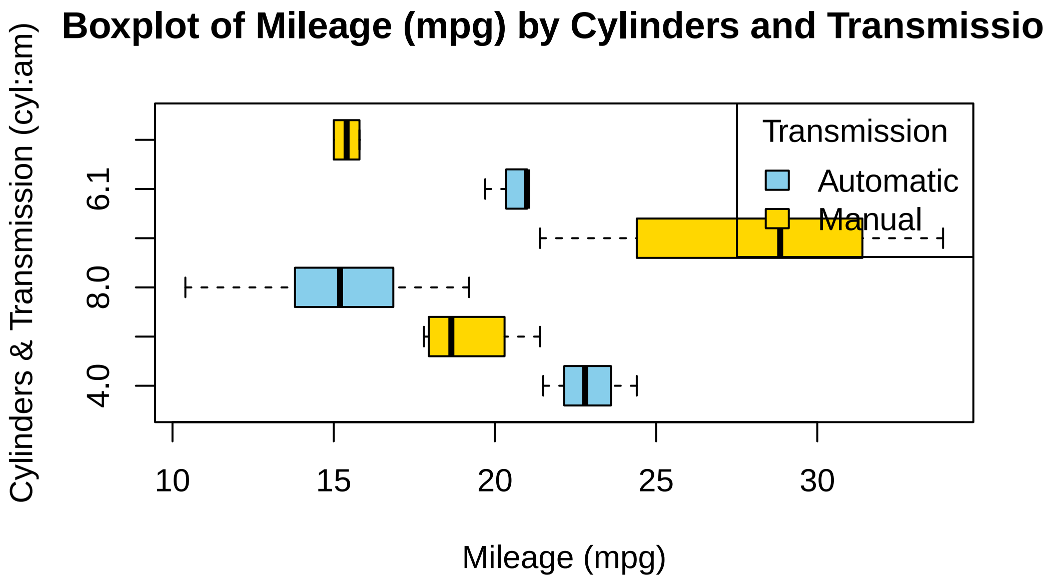


2. **Discussion**:

- This code creates a visual representation of the distribution of miles per gallon (`mpg`) based on the number of cylinders (`cyl`), using the `boxplot` function from the base graphics package in R. down:
- Data Input: The formula `mpg ~ cyl` instructs R to create separate boxplots for each unique value of `cyl`, with each boxplot representing the distribution of `mpg` values for that particular cylinder count.
- Title Configuration: `main` specifies the title of the plot as "Boxplot of Miles Per Gallon (mpg) by Cylinders."
- Axis Labels: The labels for the x-axis and y-axis are set using `xlab` and `ylab`, respectively. Here, `xlab` labels the mileage (or `mpg`), while `ylab` labels the number of cylinders (`cyl`).
- Color Choice: The `col` argument is set to "skyblue," which colors the body of the boxplots in a light blue shade.
- Orientation: By setting `horizontal` to `TRUE`, the boxplots are displayed in a horizontal orientation rather than the default vertical orientation.
- In essence, we're visualizing the variations in car mileage based on the number of cylinders using horizontal boxplots. This type of visualization helps in understanding the central tendency, spread, and potential outliers of mileage for different cylinder counts.

3. **Visualizing Median using Box Plot** – median weight of the cars broken down by cylinders (`cyl`=4,6,8) and Transmission (`am`=0,1). [1] [7]

```
# Creating a boxplot for 'mpg' values grouped by both 'cyl' and 'am'.
boxplot(mpg ~ cyl * am,
        main = "Boxplot of Mileage (mpg) by Cylinders and Transmission",
        xlab = "Mileage (mpg)",  # Label for the x-axis.
        ylab = "Cylinders & Transmission (cyl:am)",  # Label for the y-axis.
        col = c("skyblue", "gold"), # Colors of the boxplots
        horizontal = TRUE  # Setting the boxplot to be horizontal.
        )

# Adding a legend to the plot.
legend("topright",  # Position of the legend in the plot.
       legend = c("Automatic", "Manual"),  # Text for the legend.
       fill = c("skyblue", "gold"),  # Color filling
       title = "Transmission"  # Title of the legend.
)
```



4. **Discussion**:

- This R code presents a horizontal boxplot showcasing the distribution of mileage (`mpg`) based on the interaction between the number of car cylinders (`cyl`) and the type of transmission (`am`).
- Boxplot Creation: The `boxplot` function is used to generate the visualization. With the formula `mpg ~ cyl * am`, we plot the distribution of `mpg` for every combination of `cyl` and `am`.
- Color Configuration: The `col` argument specifies the colors for the boxplots. We've opted for "skyblue" and "gold" for differentiation. Depending on the order of factor levels in your data, one color typically represents one level of `am` (e.g., automatic) and the other color represents the second level (e.g., manual).
- Orientation: The `horizontal` argument, set to `TRUE`, orients the boxplots horizontally.
- Legend Addition: Following the boxplot, we add a legend using the `legend` function. Placed at the "topright" position, this legend differentiates between "Automatic" and "Manual" transmissions using the designated colors.
- In essence, our code generates a detailed visualization that elucidates the mileage distribution for various combinations of cylinder counts and transmission types in cars. [7]

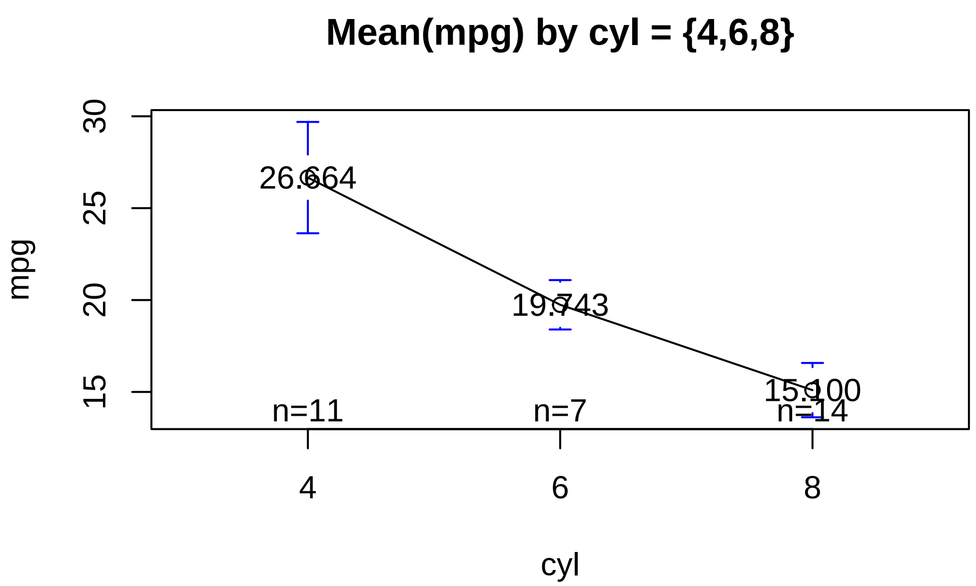
12.2.7 Means Plot across one Category

Visualizing Means – mean plot showing the average weight of the cars, broken down by transmission (`am`= 0 or 1). [8]

```
# Loading the 'gplots' package
library(gplots)
```

```

Attaching package: 'gplots'


The following object is masked from 'package:stats':

    lowess
```

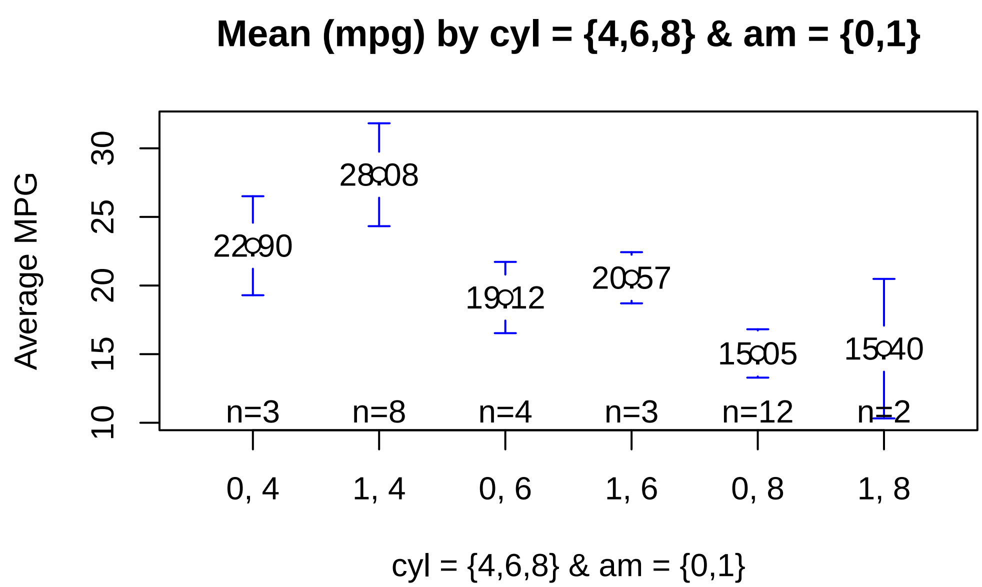
```
# Creating a plot of mean 'mpg' values for each 'cyl' category
plotmeans(data = tb,
          mpg ~ cyl,  # Formula indicating 'mpg' plotted against 'cyl'.
          mean.labels = TRUE,  # Option to display the mean value labels.
          digits = 3,  # Number of digits for the mean value labels.
          main = "Mean(mpg) by cyl = {4,6,8}"  # Main title of the plot.
          )
```



12.2.8 Means Plot across two Categories

We show a mean plot showing the mean weight of the cars broken down by Transmission Type (`am`= 0 or 1) & cylinders (`cyl` = 4,6,8). [8]

```r
# Loading the 'gplots' package for plotting data.
library(gplots)

# The plot will show mean 'mpg' values
# for each combination of 'am' and 'cyl' categories
plotmeans(mpg ~ interaction(am, cyl, sep = ", "),
          data = tb,  # Specifying the dataframe.
          mean.labels = TRUE,  # display  mean value labels on the plot.
          digits = 2,  # Setting the number of digits to 2.
          connect = FALSE,  # Disabling the connection lines between means.
          main = "Mean (mpg) by cyl = {4,6,8} & am = {0,1}",
          xlab = "cyl = {4,6,8} & am = {0,1}",  # Label for the x-axis.
          ylab = "Average MPG"  # Label for the y-axis.
          )
```



12.3 Summary of Chapter 12 – Bivariate Continuous data (Part 1 of 4)

This chapter on R programming skillfully navigates the analysis of bivariate continuous data across categorical variables. It begins with preparing the `mtcars` dataset, transforming key variables like cylinders and transmission type into categorical factors for in-depth analysis.

The chapter primarily focuses on summarizing continuous data using R functions like `aggregate()`, `tapply()`, and `describeBy()` from the `psych` package. These functions are adept at calculating and presenting summary statistics such as means and standard deviations for different categorical groups.

In terms of visualization, the chapter showcases a variety of methods to graphically represent continuous data across categories. Techniques such as Bee Swarm plots, histograms, density plots, and box plots are highlighted for their effectiveness in illustrating data distributions and variations across categorical groups.

Overall, the chapter provides a comprehensive yet concise framework for analyzing and visualizing the interplay between continuous and categorical data in R, offering valuable insights for data analysis and interpretation.

12.4 References

Basic R Programming:

[1] Chambers, J. M. (2008). Software for Data Analysis: Programming with R (Vol. 2, No. 1). Springer.

Crawley, M. J. (2012). The R Book. John Wiley & Sons.

Gardener, M. (2012). Beginning R: The Statistical Programming Language. John Wiley & Sons.

Grolemund, G. (2014). Hands-On Programming with R: Write Your Own Functions and Simulations. O'Reilly Media, Inc.

Kabacoff, R. (2022). R in Action: Data Analysis and Graphics with R and Tidyverse. Simon and Schuster.

Peng, R. D. (2016). R Programming for Data Science (pp. 86-181). Leanpub.

R Core Team. (2020). R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing. Retrieved from https://www.R-project.org/.

Tippmann, S. (2015). Programming Tools: Adventures with R. Nature, 517(7532), 109-110.

Wickham, H., Çetinkaya-Rundel, M., & Grolemund, G. (2023). R for Data Science. O'Reilly Media, Inc.

knitr:

[2] Xie, Y. (2018). knitr: A Comprehensive Tool for Reproducible Research in R. In Implementing Reproducible Research (pp. 3-31). Chapman and Hall/CRC.

psych:

[3] Revelle, W. (2020). psych: Procedures for Psychological, Psychometric, and Personality Research. Northwestern University, Evanston, Illinois. R Package Version 2.0.12. Retrieved from https://CRAN.R-project.org/package=psych.

beeswarm:

[4] Eklund, A. (2020). ggbeeswarm: Categorical Scatter (Violin Point) Plots. R Package Version 0.6.0. Retrieved from https://CRAN.R-project.org/package=ggbeeswarm.

Stem and Leaf:

[5] Lovie, S. (2005). Stem and Leaf Plot. Encyclopedia of Statistics in Behavioral Science.

Histograms:

[6] Scott, D. W. (1979). On Optimal and Data-Based Histograms. Biometrika, 66(3), 605-610.

Wand, M. P., & Jones, M. C. (1995). Kernel Smoothing. Chapman and Hall/CRC.

Box Plots:

[7] McGill, R., Tukey, J. W., & Larsen, W. A. (1978). Variations of Box Plots. The American Statistician, 32(1), 12-16.

gplots:

[8] Warnes, M. G. R., Bolker, B., Bonebakker, L., Gentleman, R., Huber, W., & Liaw, A. (2016). Package 'gplots': Various R Programming Tools for Plotting Data.

Statistics using R:

[9] Braun, W. J., & Murdoch, D. J. (2021). A First Course in Statistical Programming with R. Cambridge University Press.

Cohen, Y., & Cohen, J. Y. (2008). Statistics and Data with R: An Applied Approach Through Examples. John Wiley & Sons.

Dalgaard, P. (2008). Introductory Statistics with R. Springer.

Davies, T. M. (2016). The Book of R: A First Course in Programming and Statistics. No Starch Press.

Field, A., Miles, J., & Field, Z. (2012). Discovering Statistics Using R. Sage Publications.

Fox, J., & Weisberg, S. (2018). An R Companion to Applied Regression. Sage Publications.

Hyndman, R. J., & Fan, Y. (1996). Sample Quantiles in Statistical Packages. The American Statistician, 50(4), 361-365.

Matloff, N. (2011). The Art of R Programming: A Tour of Statistical Software Design. No Starch Press.

R Core Team. (2020). R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing. Retrieved from https://www.R-project.org/.

Schumacker, R. E. (2014). Learning Statistics Using R. Sage Publications.

Schumacker, R., & Tomek, S. (2013). Understanding Statistics Using R. Springer Science & Business Media.

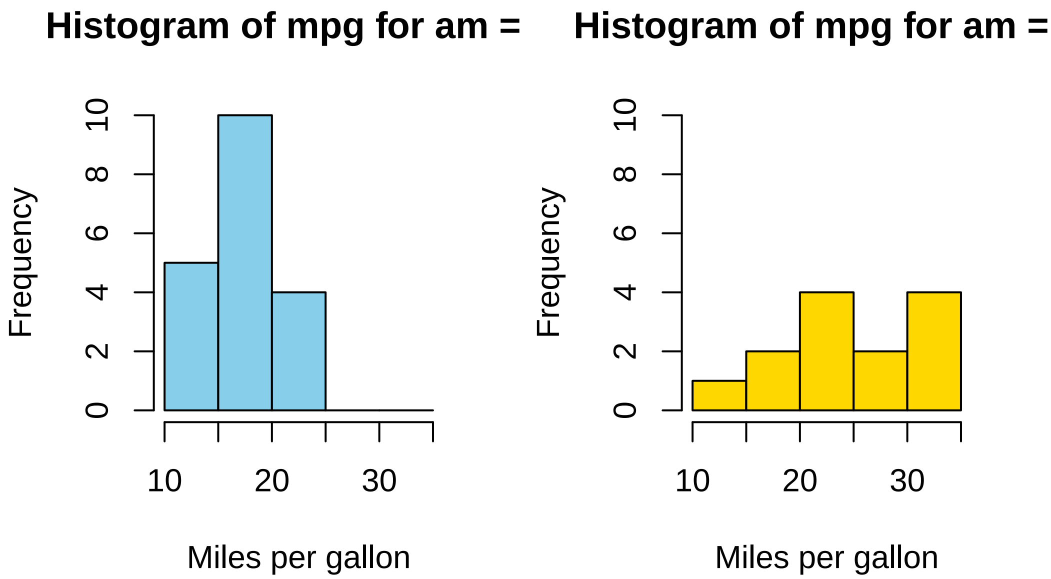
12.5 Appendix

Appendix A1

Visualizing histograms of car mileage (`mpg`) broken down by transmission (`am`=0,1)

Code written using a for loop

```
# Split the data by 'am' variable
split_data <- split(tb$mpg, tb$am)

# Create a 1-row 2-column layout
par(mfrow = c(1, 2))

# Define the color vector
color_vector <- c("skyblue", "gold")

# Create a histogram for each subset
for (i in 1:length(split_data)) {
  hist(split_data[[i]],
       main = paste("Histogram of mpg for am =", i - 1),
       breaks = seq(10, 35, by = 5), # bins with ranges 10-15, 15-20
       xlab = "Miles per gallon",
       col = color_vector[i], # Use the color vector,
       border = "black",
       ylim = c(0, 10))
}
```

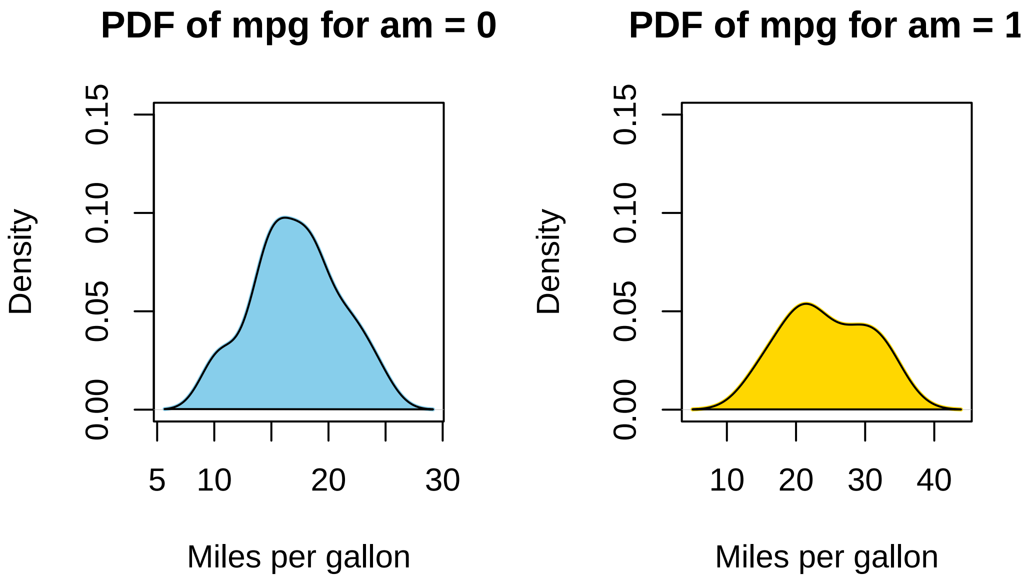


Appendix A2

Visualizing Probability Density Function (PDF) of car milegage (`mpg`) broken down by transmission (`am`=0,1), using for loop

```
# Split the data by 'am' variable
split_data <- split(tb$mpg, tb$am)

# Create a 1-row 2-column layout
par(mfrow = c(1, 2))

# Define the color vector
color_vector <- c("skyblue", "gold")

# Create a density plot for each subset
for (i in 1:length(split_data)) {
  # Calculate density
  dens <- density(split_data[[i]])

  # Plot density
  plot(dens,
       main = paste("PDF of mpg for am =", i - 1),
       xlab = "Miles per gallon",
       col = color_vector[i],
       border = "black",
       ylim = c(0, 0.15), # Adjust this value if necessary
       lwd = 2) # line width
```

```
  # Add a polygon to fill under the density curve
  polygon(dens, col = color_vector[i], border = "black")
}
```

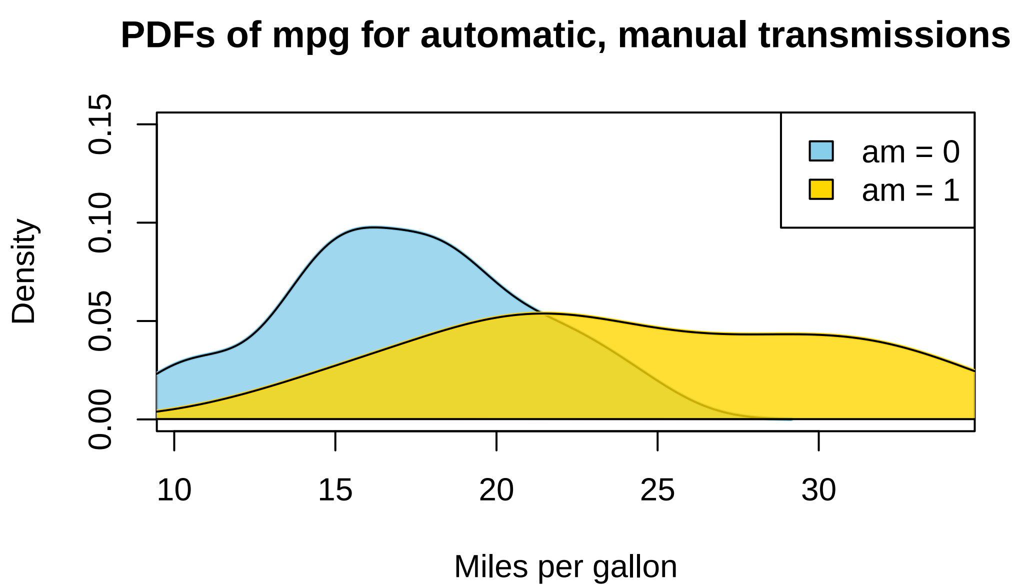


Appendix A3

Visualizing Probability Density Function (PDF) of car milegage (`mpg`) broken down by transmission (`am`=0,1), overlapping PDFs on the same plot

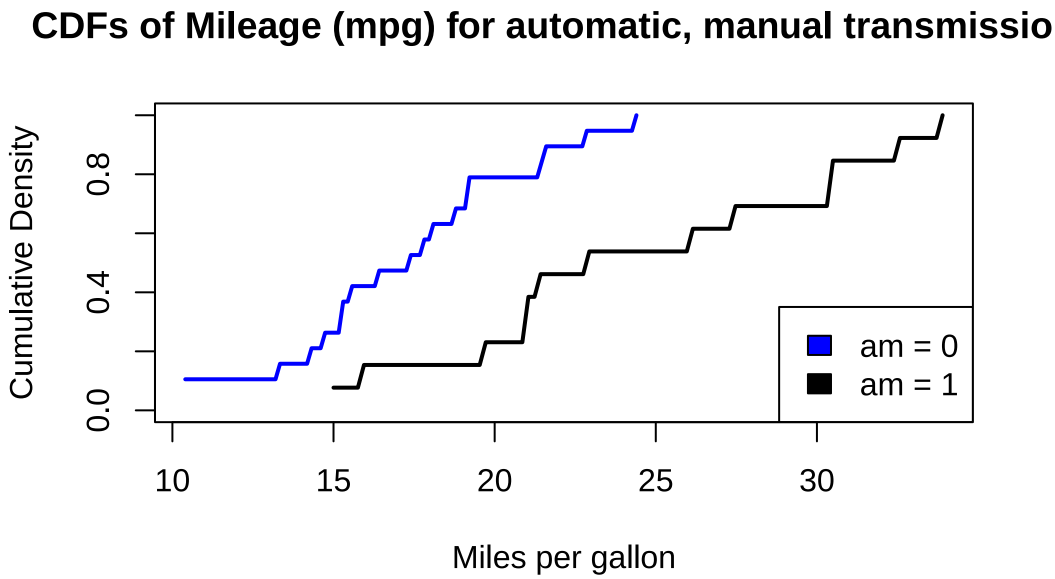
```
# Split the data by 'am' variable
split_data <- split(tb$mpg, tb$am)

# Define the color vector
color_vector <- c("skyblue", "gold")

# Define the legend labels
legend_labels <- c("am = 0", "am = 1")

# Create a density plot for each subset
# Start with an empty plot with ranges accommodating both data sets
plot(0, 0,
     xlim = range(tb$mpg), ylim = c(0, 0.15),
     type = "n",
     xlab = "Miles per gallon", ylab = "Density",
     main = "PDFs of mpg for automatic, manual transmissions")
```

```r
for (i in 1:length(split_data)) {
  # Calculate density
  dens <- density(split_data[[i]])

  # Add density plot
  lines(dens,
        col = color_vector[i],
        lwd = 2) # line width

  # Add a polygon to fill under the density curve
  polygon(dens,
          col = adjustcolor(color_vector[i],
                            alpha.f = 0.8),
          border = "black")
}

# Add legend to the plot
legend("topright",
       legend = legend_labels,
       fill = color_vector,
       border = "black")
```



Appendix A4

```r
# Split the data by 'am' variable
split_data <- split(tb$mpg, tb$am)

# Define the color vector
color_vector <- c("blue", "black")

# Define the legend labels
legend_labels <- c("am = 0", "am = 1")

# Create a cumulative density plot for each subset
# Start with an empty plot with ranges accommodating both data sets
plot(0, 0, xlim = range(mtcars$mpg), ylim = c(0, 1), type = "n",
     xlab = "Miles per gallon", ylab = "Cumulative Density",
     main = "CDFs of Mileage (mpg) for automatic, manual transmissions")

for (i in 1:length(split_data)) {
  # Calculate empirical cumulative density function
  ecdf_func <- ecdf(split_data[[i]])

  # Add CDF plot using curve function
  curve(ecdf_func(x),
        from = min(split_data[[i]]), to = max(split_data[[i]]),
        col = color_vector[i],
        add = TRUE,
        lwd = 2) # line width
}
# Add legend to the plot
legend("bottomright",
       legend = legend_labels,
       fill = color_vector,
       border = "black")
```



CDFs of Mileage (mpg) for automatic, manual transmissio
Cumulative Density
0.0
0.4
0.8
10
15
20
25
30
Miles per gallon
am = 0
am = 1

13 Bivariate Continuous data (Part 2 of 4)

Chapter 13.

In this chapter, we analyze categorical and continuous data using R's dplyr and ggplot2 packages. It demonstrates various ggplot2 visualization techniques for continuous data within single categories, including Bee Swarm, Histogram, Probability Density Function (PDF), Cumulative Density Function (CDF), Box plot, and Violin plot.

Next, we use dplyr and ggplot2 for data summarization. This includes calculating and visualizing summary statistics such as the mean and standard deviation. Techniques like line and bar plots with error bars are employed to elucidate relationships. The chapter further extends to bivariate analyses, examining relationships between multiple continuous variables and between different categories. This approach offers an in-depth guide to effectively summarizing and visualizing continuous data in R.

Data: Suppose we run the following code to prepare the `mtcars` data for subsequent analysis and save it in a tibble called `tb`. [1]

```
# Load the required libraries, suppressing annoying startup messages
library(dplyr, quietly = TRUE, warn.conflicts = FALSE)
library(tibble, quietly = TRUE, warn.conflicts = FALSE)
library(knitr) # For formatting tables

# Read the mtcars dataset into a tibble called tb
data(mtcars)
tb <- as_tibble(mtcars)
# Convert relevant columns into factor variables
tb$cyl <- as.factor(tb$cyl) # cyl = {4,6,8}, number of cylinders
tb$am <- as.factor(tb$am) # am = {0,1}, 0:automatic, 1: manual transmission
tb$vs <- as.factor(tb$vs) # vs = {0,1}, v-shaped engine, 0:no, 1:yes
tb$gear <- as.factor(tb$gear) # gear = {3,4,5}, number of gears

# Directly access the data columns of tb, without tb$mpg
attach(tb)
```

13.1 Visualizing Continuous Data using `ggplot2`

Let's take a closer look at some of the most effective ways of visualizing continuous data, across one Category, **using ggplot2**, including

(i) Bee Swarm plots, using ggplot2;

(ii) Histograms, using ggplot2;

(iii) PDF and CDF Density plots, using ggplot2;

(iv) Box plots, using ggplot2;

(v) Violin plots, using ggplot2;

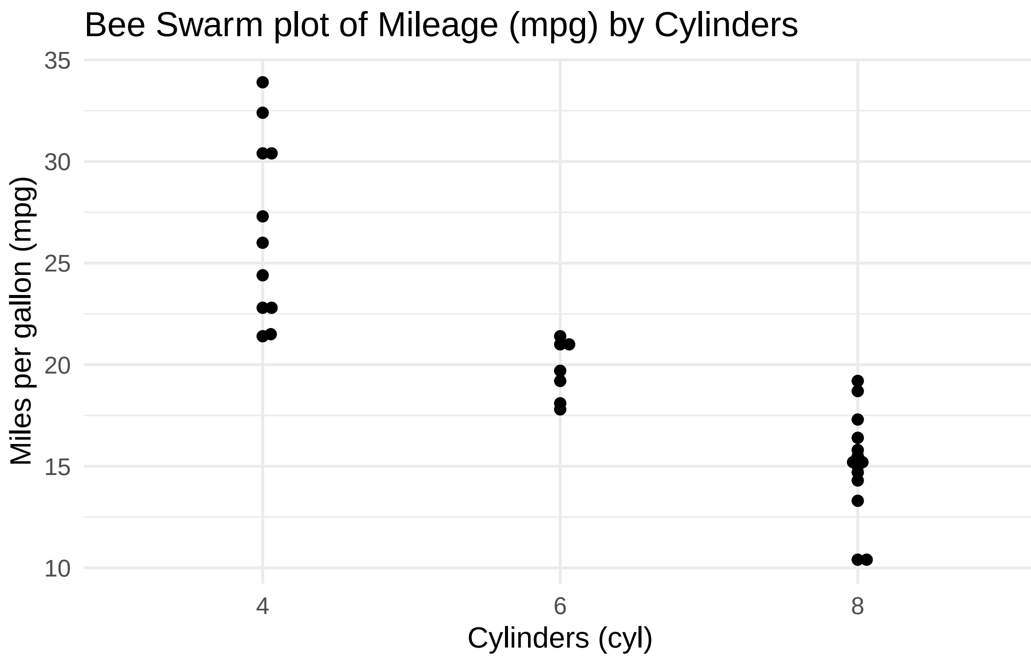
13.1.1 Bee Swarm Plot across one Category using `ggbeeswarm`

- Visualizing Median using Box Plot – median weight of the cars broken down by cylinders (`cyl`=4,6,8). [2] [3]

```
# Loading the ggplot2 package for data visualization
library(ggplot2)
```

```

Attaching package: 'ggplot2'


The following object is masked from 'tb':

    mpg
```

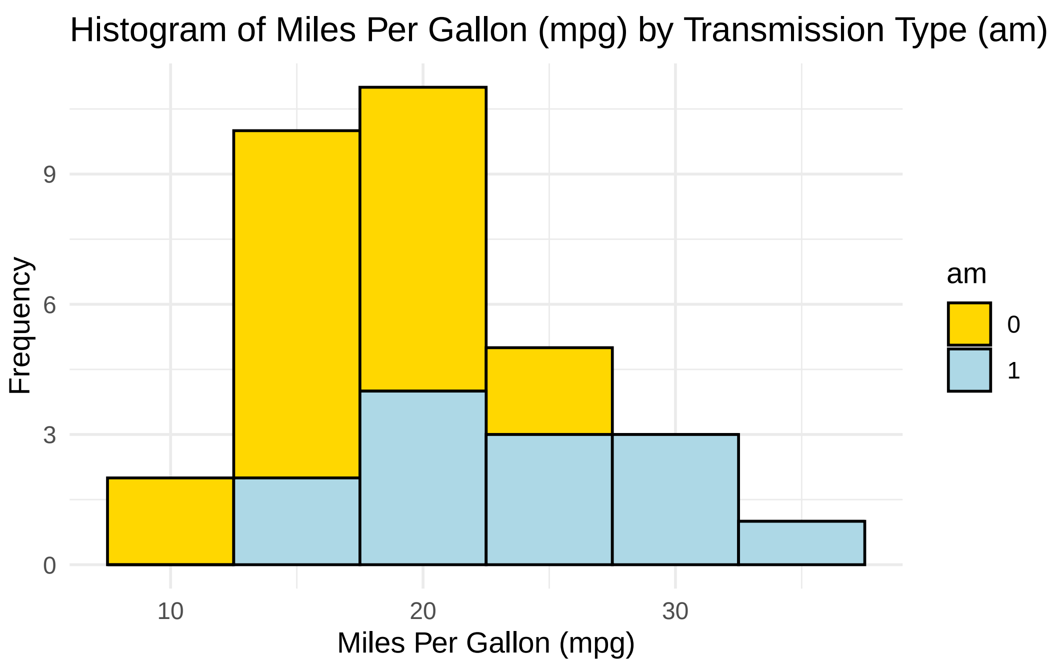
```
# Loading the ggbeeswarm package for beeswarm plot
library(ggbeeswarm)

# Creating a beeswarm plot using ggplot
ggplot(mtcars,  # Specifying the data source as 'mtcars' dataset
       aes(x = factor(cyl),  # Mapping 'cyl' as a factor to the x-axis
           y = mpg)) +  # Mapping 'mpg' to the y-axis
  geom_beeswarm() +  # Adding the beeswarm layer
  labs(title = "Bee Swarm plot of Mileage (mpg) by Cylinders",
       x = "Cylinders (cyl)",  # Label for the x-axis
       y = "Miles per gallon (mpg)") +  # Label for the y-axis
  theme_minimal()  # Applying a minimal theme to the plot
```



13.1.2 Histograms across one Category using ggplot2

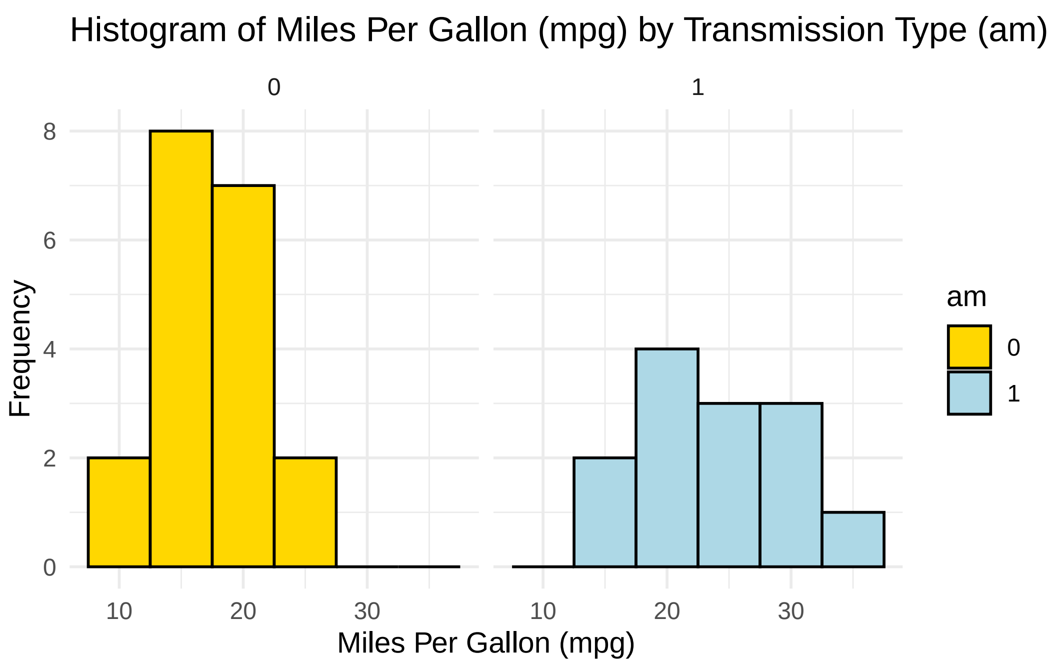
- Visualizing histograms of car milegage (mpg) broken down by transmission (am=0,1). [2] [4] [5]

```
# Using ggplot2 to create a histogram
ggplot(tb, aes(x = mpg,  # Setting 'mpg' as the x-axis variable
               fill = am)) +  # Filling bars based on 'am'
  geom_histogram(binwidth = 5,
                 color = "black") +  # histogram with bin width of 5
  scale_fill_manual(values = c("gold", "lightblue")) +  # setting fill colors
  theme_minimal() +  # Applying a minimalistic theme
  labs(title = "Histogram of Miles Per Gallon (mpg) by Transmission Type (am)",
       x = "Miles Per Gallon (mpg)",  # Label for the x-axis
       y = "Frequency")  # Label for the y-axis
```



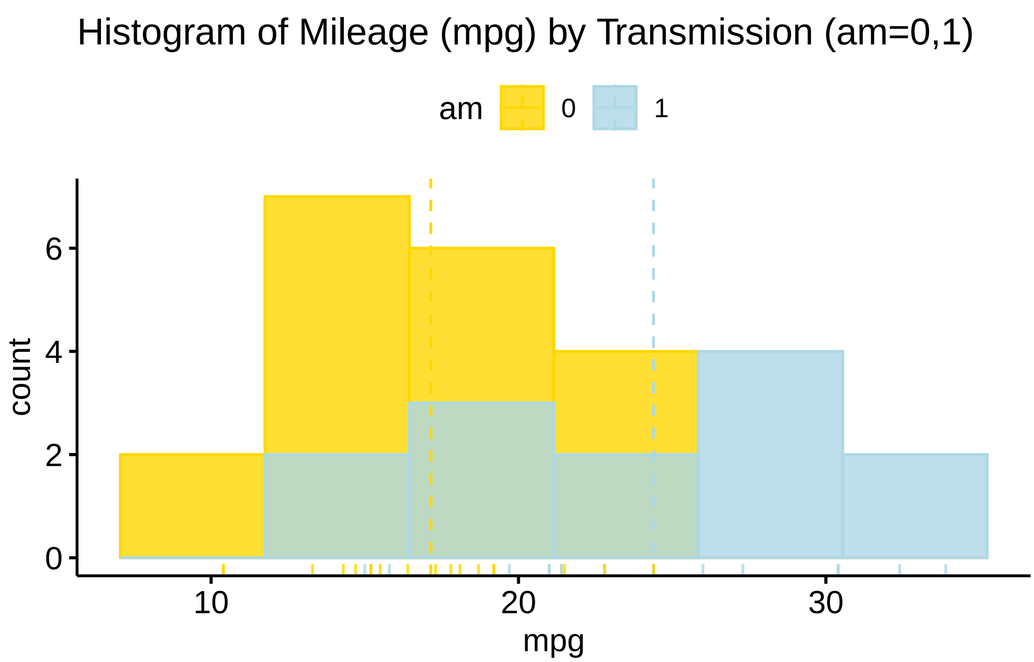
- **Discussion**: If we want separate histograms, we can set facet_wrap(~ am).

```
# Creating a histogram using ggplot2 with 'tb' dataset
ggplot(tb, aes(x = mpg,  # Setting 'mpg' as the x-axis variable
               fill = am)) +  # Filling bars based on 'am' (transmission type)
  geom_histogram(binwidth = 5, color = "black") +  # bin width of 5
  scale_fill_manual(values = c("gold", "lightblue")) +  # fill colors
  facet_wrap(~ am) +  # Separating the histograms by 'am' value for comparison
  theme_minimal() +  # Applying a minimalistic theme to the plot
  labs(title = "Histogram of Miles Per Gallon (mpg) by Transmission Type (am)",
       x = "Miles Per Gallon (mpg)",  # Label for the x-axis
       y = "Frequency")  # Label for the y-axis
```



13.1.2.1 Histogram across one Category using `ggpubr`

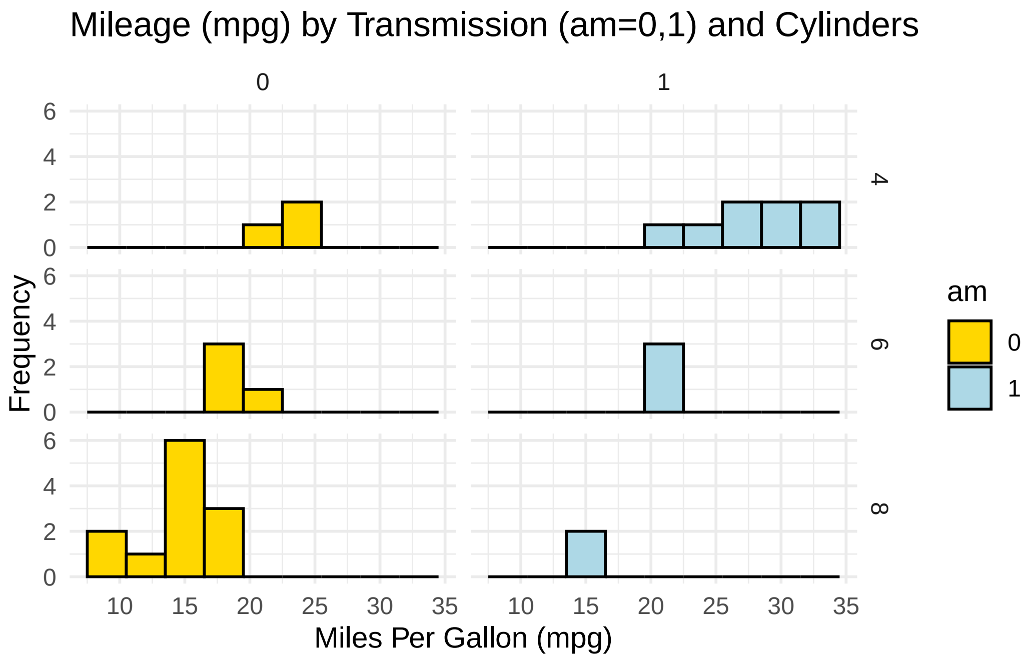
```
# Loading the ggpubr package
library(ggpubr)

# Creating a histogram with enhanced features using gghistogram from ggpubr.
gghistogram(tb,
            x = "mpg",  # Setting 'mpg' as the variable for histogram.
            bins = 6,  # Specifying the number of bins.
            add = "mean",  # Adding a line to indicate the mean .
            rug = TRUE,  # Adding a rug plot at the bottom .
            color = "am",  # Setting the color .
            fill = "am",  # Filling the bars based on 'am'.
            alpha = 0.8,  # Setting transparency level of the fill.
            palette = c("gold", "lightblue"),  # Defining a color palette
            title = "Histogram of Mileage (mpg) by Transmission (am=0,1)")
```



13.1.3 Histograms across two Categories using `ggplot2`

- Visualizing histograms of car milegage (`mpg`) by transmission (`am`=0,1) and cylinders (`cyl`=4,6,8). [2] [4] [5]

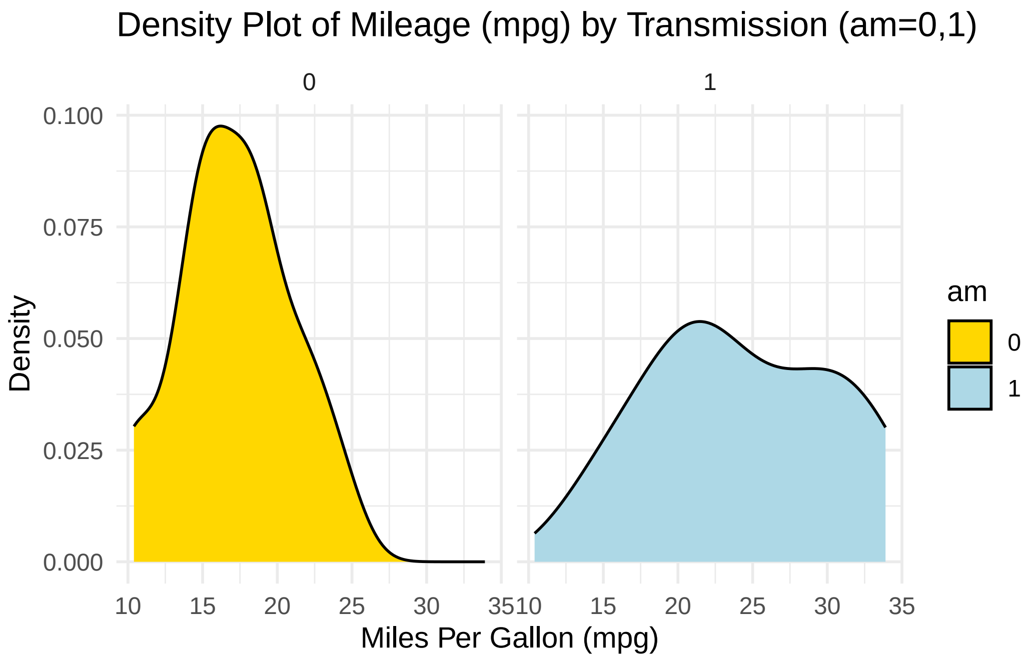
```
ggplot(tb, aes(x = mpg,
               fill = am)) +
  geom_histogram(binwidth = 3, color = "black") +  # bin width of 3
  scale_fill_manual(values = c("gold", "lightblue")) +  # fill
  facet_grid(cyl ~ am) +  # grid faceted by 'cyl' and 'am'
  theme_minimal() +  # Applying a minimalistic theme to the plot
  labs(title = "Mileage (mpg) by Transmission (am=0,1) and Cylinders",
       x = "Miles Per Gallon (mpg)",  # Label for the x-axis
       y = "Frequency")  # Label for the y-axis
```



13.1.4 PDF across one Category using `ggplot2`

- Visualizing the Probability Density Functions (PDF) of car milegage (`mpg`) by transmission (`am`=0,1). [2], [5]

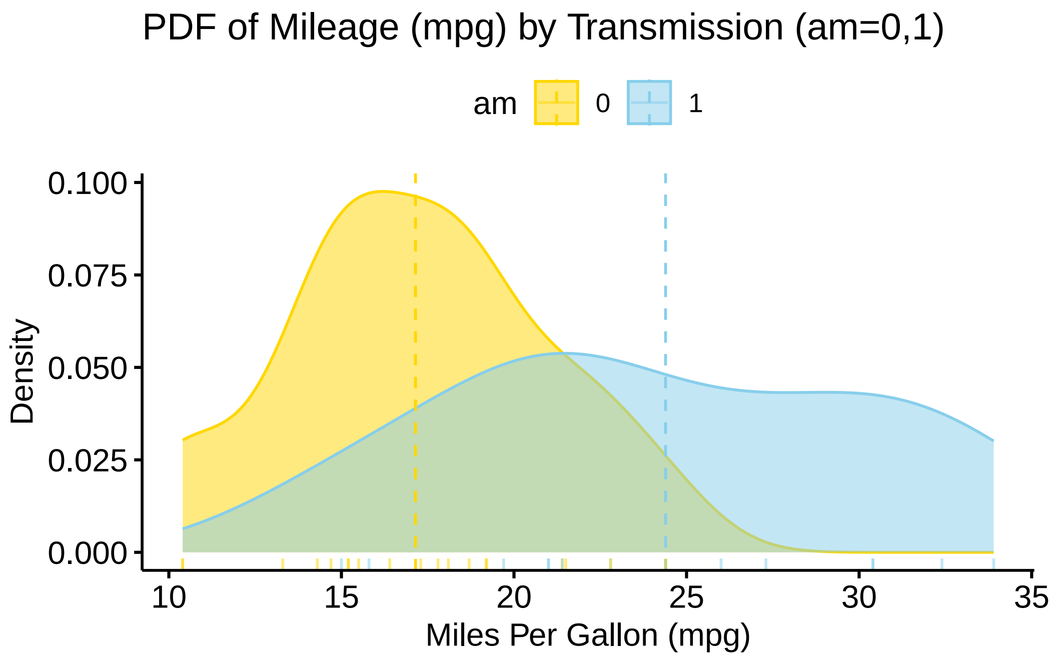
```
# Using ggplot2 to create a density plot
ggplot(tb, aes(x = mpg,  # Setting 'mpg' as the x-axis variable
               fill = am)) +  # Filling the plot based on 'am'
  geom_density(color = "black") +  # Creating a density plot
  scale_fill_manual(values = c("gold", "lightblue")) +  # fill colors
  facet_wrap(~ am) +  # Separating the plots by 'am'
  theme_minimal() +  # Applying a minimalistic theme to the plot
  labs(title = "Density Plot of Mileage (mpg) by Transmission (am=0,1)",
       x = "Miles Per Gallon (mpg)",  # Label for the x-axis
       y = "Density")  # Label for the y-axis
```



13.1.4.1 PDF across one Category using `ggpubr`

- The provided R code creates a Boxplot of the `mpg` (miles per gallon) variable, using the `ggboxplot()` function from the `ggpubr` package.

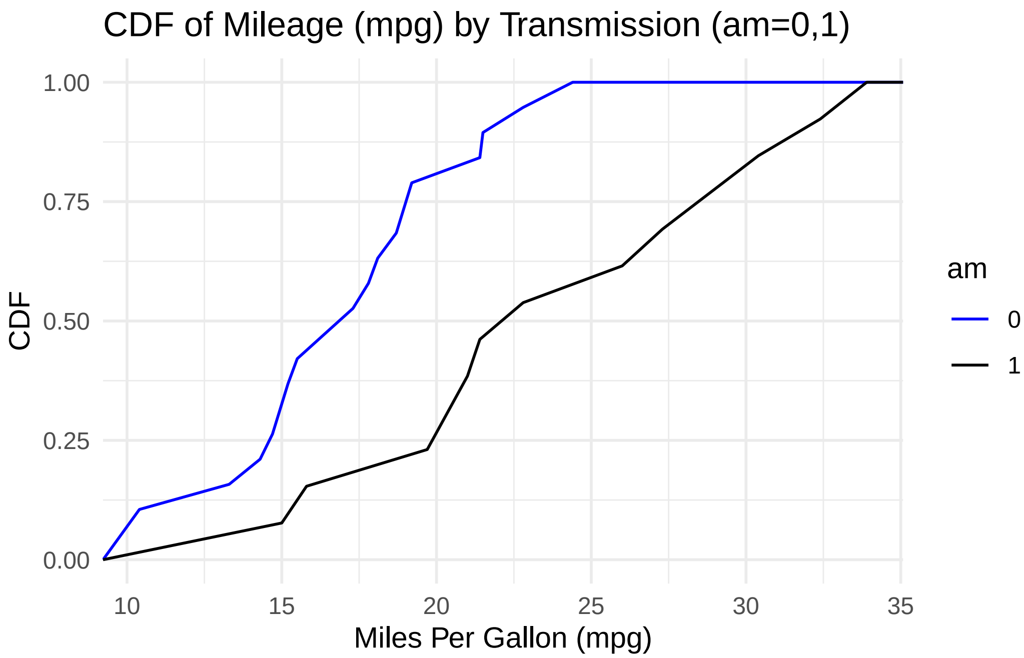
```
# Loading the ggpubr package
library(ggpubr)

# Creating a density plot with enhanced features using ggdensity from ggpubr.
ggdensity(tb,
          x = "mpg",  # Setting 'mpg' as the variable for the density plot.
          color = "am",  # Setting the color of the lines based on 'am'.
          fill = "am",  # Filling the plot based on 'am'.
          add = "mean",  # Adding a line for the mean of the distribution.
          rug = TRUE,  # Adding a rug plot at the bottom
          palette = c("gold", "skyblue"),  # Defining a color palette
          title = "PDF of Mileage (mpg) by Transmission (am=0,1)",
          ylab = "Density",  # Label for the y-axis.
          xlab = "Miles Per Gallon (mpg)"  # Label for the x-axis.
)
```



13.1.5 CDF across one Category using `ggplot2`

- Visualizing the Cumulative Density Functions (CDF) of car milegage (`mpg`) by transmission (`am`=0,1). [2], [5]

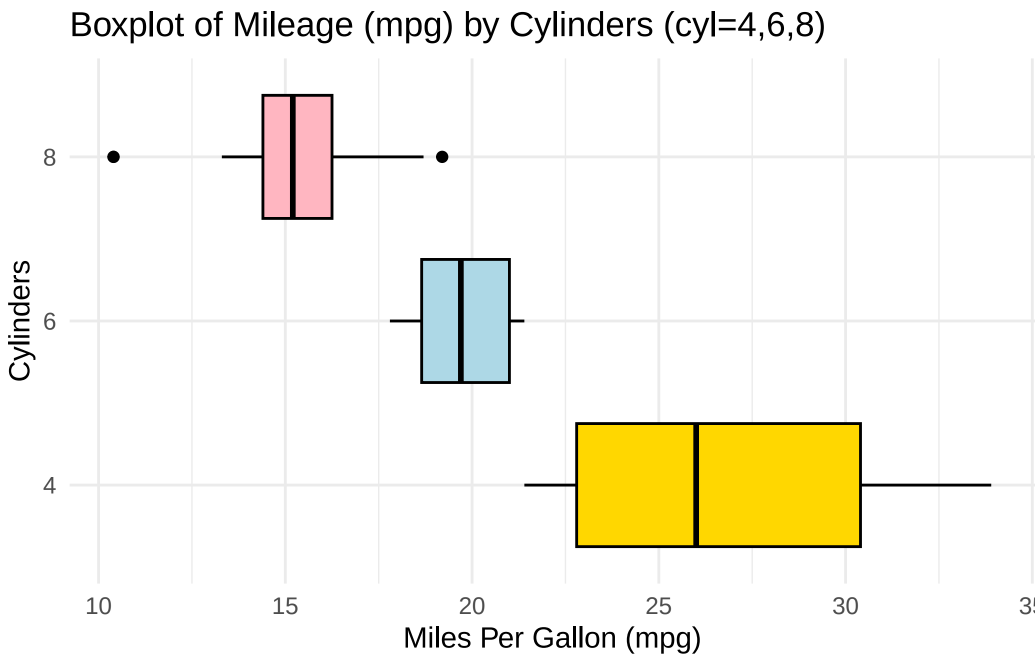
```
# Loading the ggplot2 package for data visualization.
library(ggplot2)

# Creating cumulative distribution function (CDF) plots for 'mpg' based on 'am'.
ggplot(tb, aes(x = mpg,
               color = factor(am))) +  # Mapping 'mpg' to x-axis
  stat_ecdf(geom = "line") +  # Using 'stat_ecdf' to compute CDF.
  scale_color_manual(values = c("blue", "black")) +  #  colors
  labs(x = "Miles Per Gallon (mpg)", y = "CDF",  # Setting labels .
       title = "CDF of Mileage (mpg) by Transmission (am=0,1)",
       color = "am") +  # Labeling the color legend as 'am'.
  theme_minimal()  # Applying a minimalistic theme to the plot.
```



13.1.6 Box Plot across one Category using `ggplot2`

- Visualizing Boxplots of car milegage (`mpg`) broken down by cylinders (`cyl`=4,6,8). [2], [6]

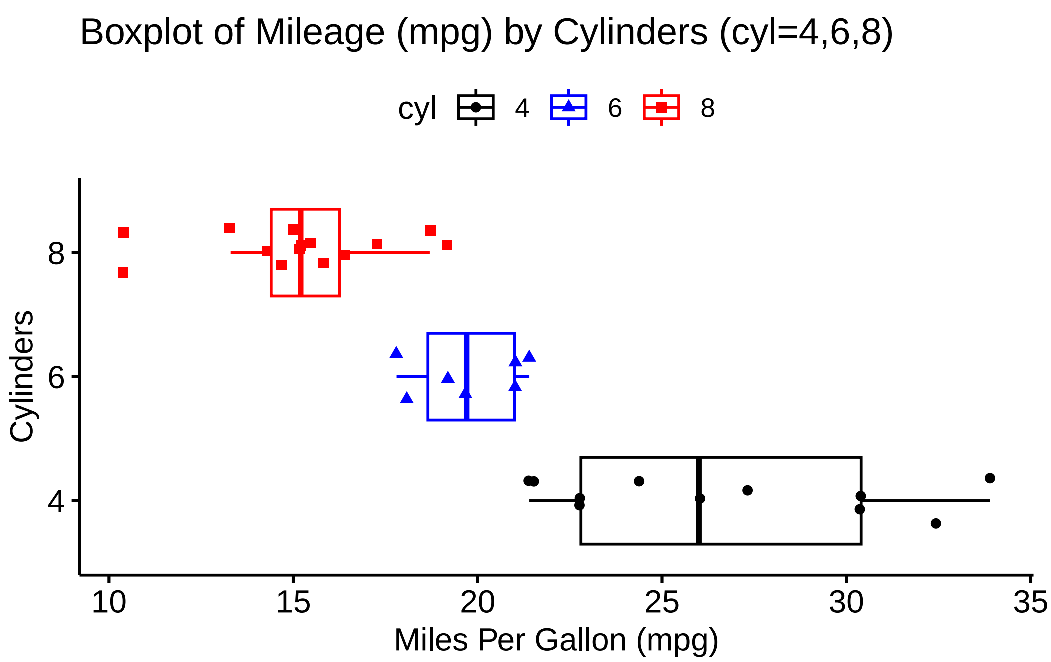
```
# Loading the ggplot2 package for data visualization.
library(ggplot2)

# Creating a boxplot using ggplot2.
ggplot(tb, aes(x = cyl,  # Setting 'cyl' as the x-axis variable
               y = mpg)) +  # Setting 'mpg' as the y-axis variable
  geom_boxplot(fill = c("gold", "lightblue", "lightpink"),  # fill colors
               color = "black") +  # Setting the color of the box borders
  coord_flip() +  # Flipping the coordinates to make the boxplot horizontal
  labs(title = "Boxplot of Mileage (mpg) by Cylinders (cyl=4,6,8)",
       y = "Miles Per Gallon (mpg)",  # Label y-axis (flipped to x-axis)
       x = "Cylinders") +  # Label for the x-axis (flipped to y-axis)
  theme_minimal()  # Applying a minimalistic theme to the plot
```



13.1.6.1 Box Plot across one Category using `ggpubr`

- The provided R code creates a Boxplot of the `mpg` (miles per gallon) variable, using the `ggboxplot()` function from the `ggpubr` package. [2], [6]

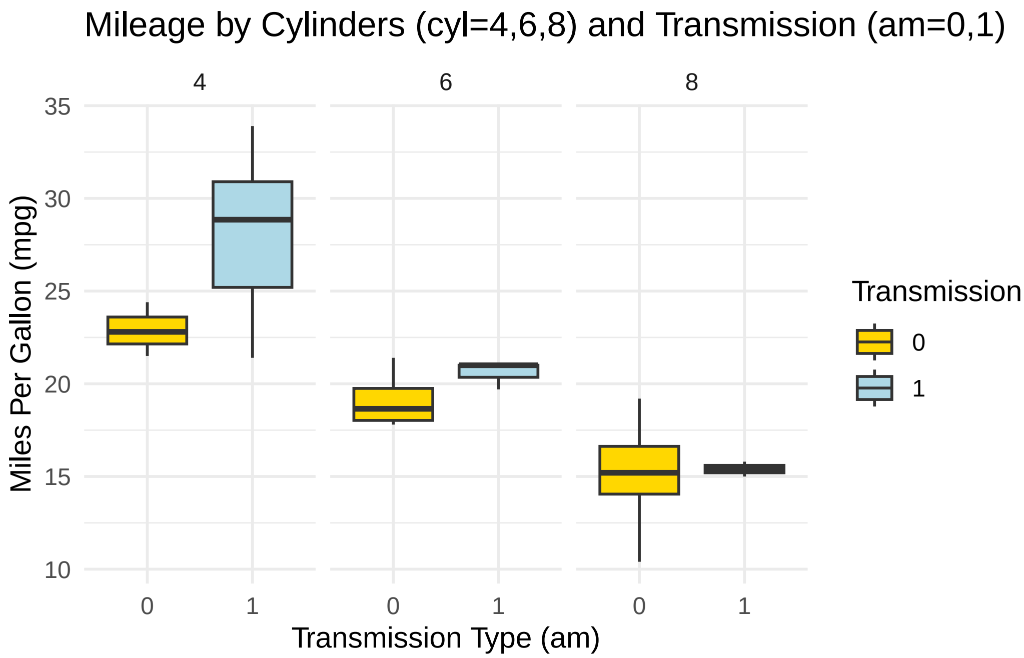
```
# Loading the ggpubr package
library(ggpubr)

# Creating a boxplot with enhanced features using ggboxplot from ggpubr.
ggboxplot(tb,
          y = "mpg",  # Setting 'mpg' as the y-axis variable.
          x = "cyl",  # Setting 'cyl' as the x-axis variable.
          color = "cyl",  # Setting the outline color of the boxes.
          fill = "white",  # Setting the fill color of the boxes to white.
          palette = c("black", "blue", "red"),  # Defining a color palette.
          shape = "cyl",  # Defining the shape of data points based on 'cyl'.
          orientation = "horizontal",  # Setting the orientation.
          add = "jitter",  # Adding jitter to display individual points.
          title = "Boxplot of Mileage (mpg) by Cylinders (cyl=4,6,8)",
          ylab = "Miles Per Gallon (mpg)",  # Label for the y-axis.
          xlab = "Cylinders"  # Label for the x-axis.
)
```



13.1.7 Box Plot across two Categories using `ggplot2`

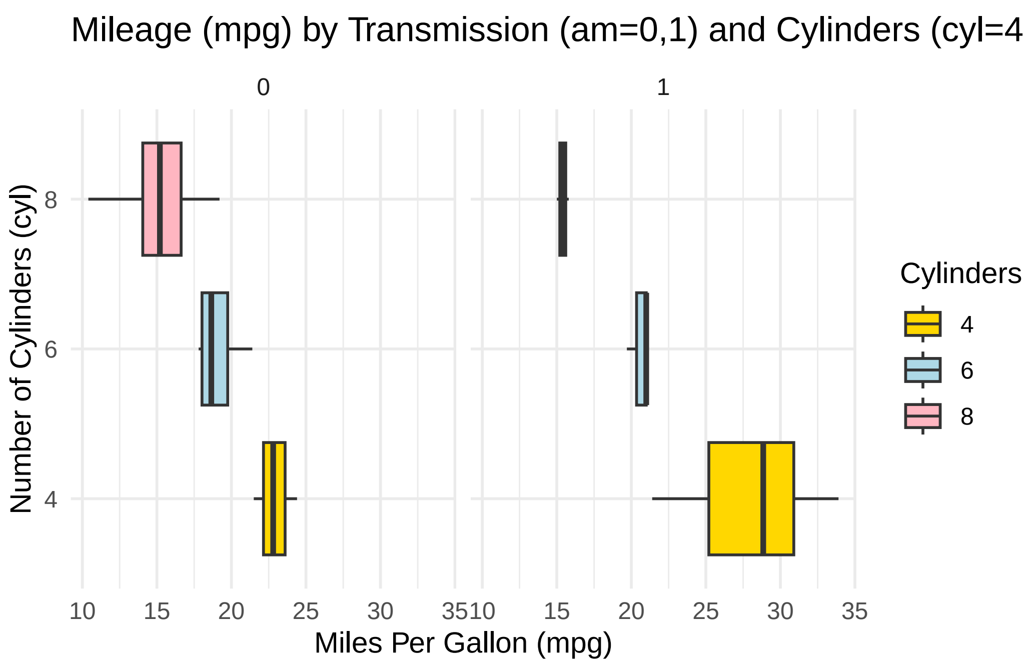
- Visualizing Boxplots of car milegage (`mpg`) broken down by cylinders (`cyl`=4,6,8) and Transmission (am=0,1). [2], [6]

```
# Creating a boxplot using ggplot2
ggplot(tb,
       aes(x = as.factor(am),
           y = mpg, # Mapping 'am' and 'mpg' to x and y axes
           fill = as.factor(am))) +  # filling boxplots based on 'am'.
  geom_boxplot() +  # Adding the boxplot layer.
  scale_fill_manual(values = c("gold", "lightblue"),
                    name = "Transmission") +  # fill colors
  facet_grid(~ cyl) +  # grid of boxplots faceted by 'cyl'
  theme_minimal() +  # a minimalistic theme for a cleaner look.
  labs(title = "Mileage by Cylinders (cyl=4,6,8) and Transmission (am=0,1)",
       x = "Transmission Type (am)",  # Label for the x-axis.
       y = "Miles Per Gallon (mpg)")  # Label for the y-axis.
```



Alternately:

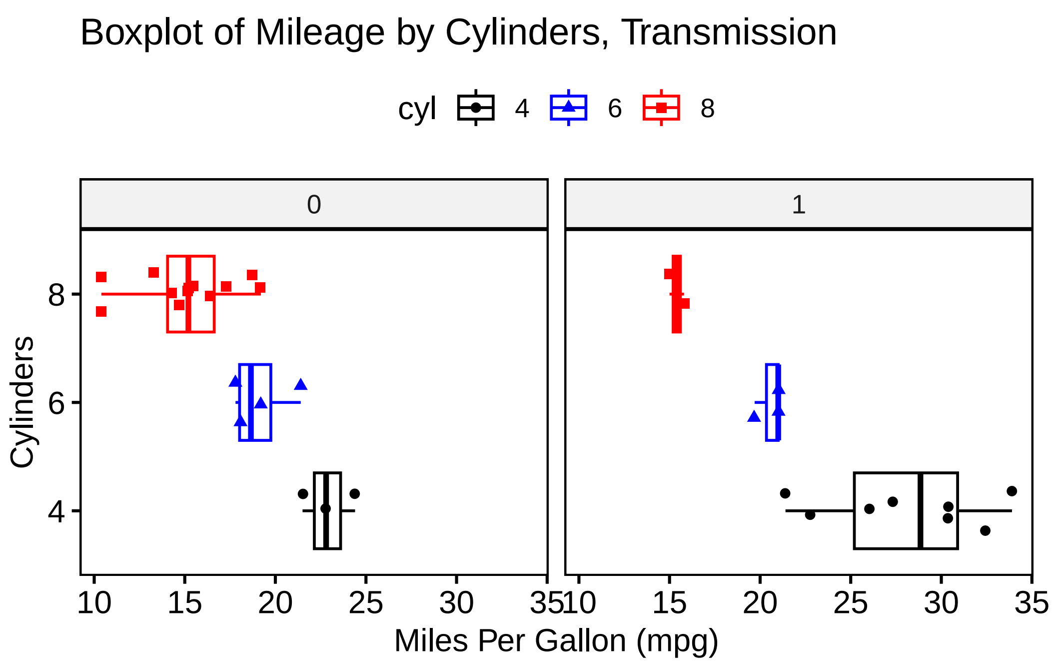
```
# Creating a boxplot using ggplot2
ggplot(tb, aes(x = as.factor(cyl),
               y = mpg, # Mapping 'cyl' and 'mpg' to x and y axes
               fill = as.factor(cyl))) +  # filling boxplots based on 'cyl'
  geom_boxplot() +  # Adding the boxplot layer.
  scale_fill_manual(values = c("gold", "lightblue", "lightpink"),
                    name = "Cylinders") +  # fill colors
  facet_grid(~ am) +  # Grid of boxplots faceted by 'am' (transmission)
  theme_minimal() +  # Applying a minimalistic theme for a cleaner look.
  coord_flip() +  # Flipping the coordinates to make the boxplot horizontal.
  labs(title = "Mileage (mpg) by Transmission (am=0,1) and Cylinders (cyl=4,6,8)",
       x = "Number of Cylinders (cyl)",  # Label for the x-axis.
       y = "Miles Per Gallon (mpg)")  # Label for the y-axis.
```



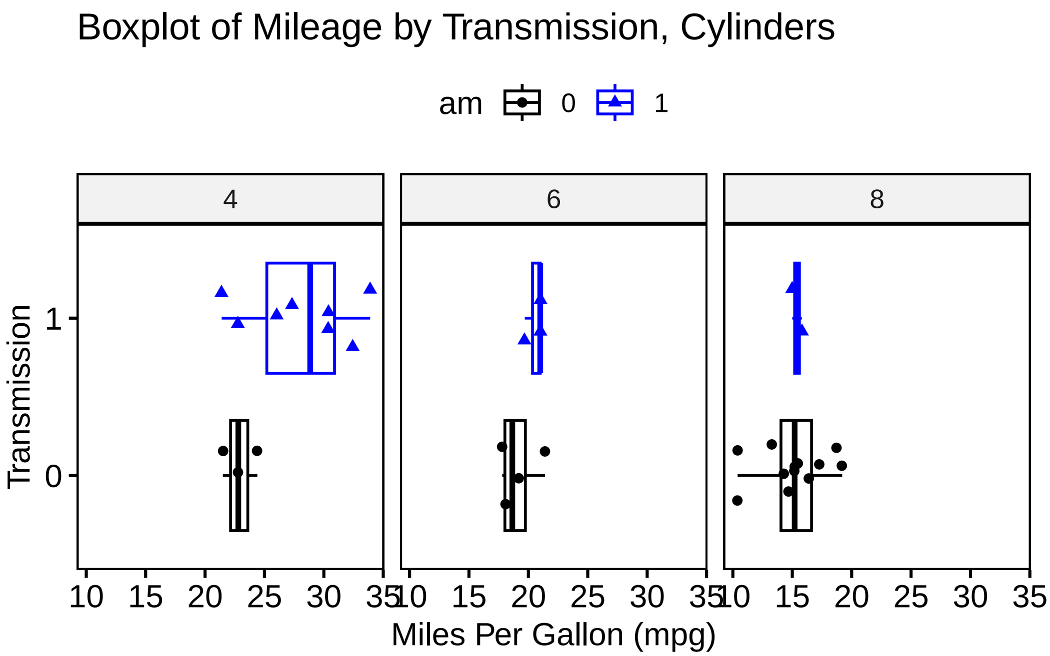
13.1.7.1 Box Plot across two Categories using `ggpubr`

- The provided R code creates Boxplots of the `mpg` (miles per gallon) variable, using the `ggboxplot()` function from the `ggpubr` package.

```
# Loading the ggpubr package for enhanced ggplot2 functionalities.
library(ggpubr)

# Creating a boxplot with additional features using ggboxplot from ggpubr.
ggboxplot(tb,
          y = "mpg",  # Setting 'mpg' as the y-axis variable.
          x = "cyl",  # Setting 'cyl' as the x-axis variable.
          color = "cyl",  # Setting the outline color of the boxes
          fill = "white",  # Setting the fill color of the boxes to white.
          palette = c("black", "blue", "red"),  # Defining a color palette
          shape = "cyl",  # Defining the shape of data points
          orientation = "horizontal",  # Setting the orientation
          add = "jitter",  # Adding jitter .
          facet.by = "am",  # Faceting the plot by 'am' .
          title = "Boxplot of Mileage by Cylinders, Transmission",
          ylab = "Miles Per Gallon (mpg)",  # Label for the y-axis.
          xlab = "Cylinders"  # Label for the x-axis.
)
```



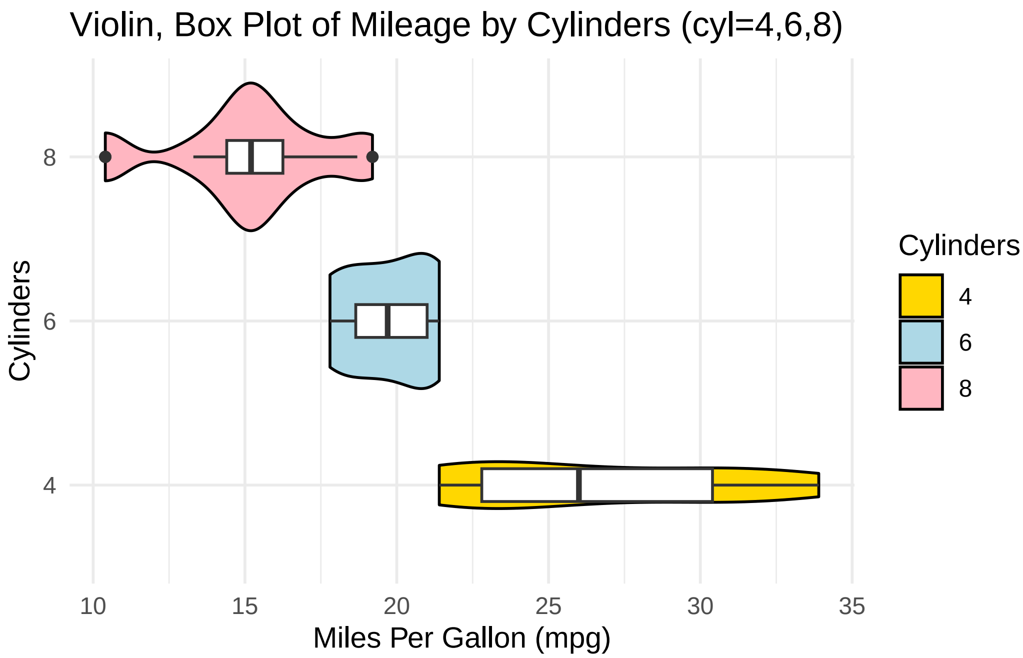

```
# Loading the ggpubr package for enhanced ggplot2 functionalities.
library(ggpubr)

# Creating a boxplot with additional features using ggboxplot from ggpubr.
ggboxplot(tb,
          y = "mpg",  # Setting 'mpg' as the y-axis variable.
          x = "am",  # Setting 'am' as the x-axis variable.
          color = "am",  # Setting the outline color of the boxes
          fill = "white",  # Setting the fill color of the boxes.
          palette = c("black", "blue"),  # Defining a color palette .
          shape = "am",  # Defining the shape of data points based on 'am'.
          orientation = "horizontal",  # Setting the orientation
          add = "jitter",  # Adding jitter
          facet.by = "cyl",  # Faceting the plot by 'cyl' .
          title = "Boxplot of Mileage by Transmission, Cylinders",
          ylab = "Miles Per Gallon (mpg)",  # Label for the y-axis.
          xlab = "Transmission"  # Label for the x-axis.
)
```



13.1.8 Violin Plot across one Category using `ggplot2`

- We can embed boxplots within the above Violin plots, as follows. [2], [6]

```
# Loading the ggplot2 package for data visualization.
library(ggplot2)

# Creating a combined plot of violin and box plots using ggplot.
ggplot(tb, aes(x = factor(cyl),  # Setting 'cyl' as the x-axis variable,
               y = mpg)) +  # Setting 'mpg' as the y-axis variable.
  geom_violin(aes(fill = factor(cyl)),  # Creating violin plots, .
              color = "black") +  # Setting the outline color
  scale_fill_manual(values = c("gold", "lightblue", "lightpink"),
                    name = "Cylinders") +  # Setting the legend title.
  geom_boxplot(width = 0.2,  # Adding box plots with specified width.
               fill = "white") +  # Setting the box fill color to white.
  coord_flip() +  # Flipping the coordinates to create horizontal plots.
  labs(title = "Violin, Box Plot of Mileage by Cylinders (cyl=4,6,8)",
       y = "Miles Per Gallon (mpg)",  # Label for the y-axis.
       x = "Cylinders") +  # Label for the x-axis.
  theme_minimal()  # Applying a minimal theme to the plot.
```



13.2 Summarizing Continuous Data using dplyr and ggplot2

13.2.1 Across one Category using dplyr and ggplot2

1. Calculating the mean and standard deviation

- We demonstrate the bivariate relationship between Miles Per Gallon (mpg) and Cylinders (cyl) using ggplot2. [1], [2]

```
# Loading the dplyr package for data manipulation.
suppressPackageStartupMessages(library(dplyr))

# Using dplyr to calculate summary statistics for 'mpg' grouped by 'cyl'.
s1 <- tb %>%
  group_by(cyl) %>%  # Grouping the data by 'cyl'.
  summarise(Mean_mpg = mean(mpg, na.rm = TRUE),
            SD_mpg = sd(mpg, na.rm = TRUE))

# Creating a table using the kable function with specified formatting.
kable(s1,
      digits = 2,
      caption = "Summary Statistics of Mileage (mpg) by Cylinders")
```

Table 13.1: Summary Statistics of Mileage (mpg) by Cylinders

cyl	Mean_mpg	SD_mpg
4	26.66	4.51
6	19.74	1.45
8	15.10	2.56

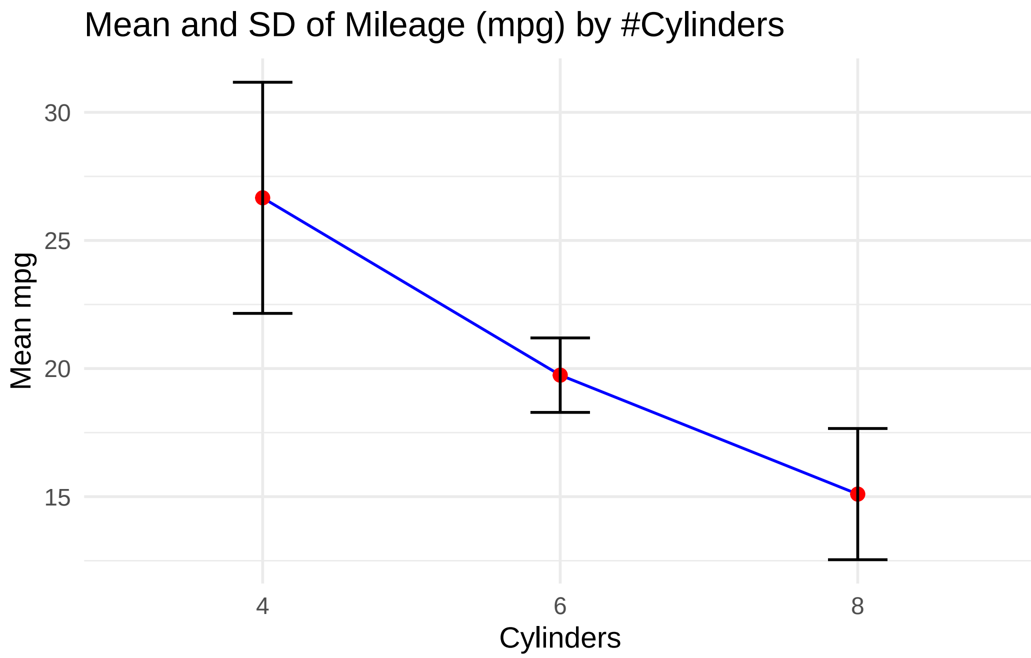
2. **Discussion:**

- In this code, we use the pipe operator `%\>%` to perform a series of operations. We first group the data by the `cyl` column using the `group_by()` function. We then use `summarise()` to apply the `mean()` and `sd()` functions to the `mpg` column.
- The results are stored in new columns, aptly named `Mean_mpg` and `SD_mpg`.
- We set `na.rm = TRUE` in both `mean()` and `sd()` function calls, to remove any missing values before calculation.
- The data resulting from the above code consists of grouped cylinder counts (`cyl`), their corresponding mean miles per gallon (`Mean_mpg`), and the standard deviation of miles per gallon (`SD_mpg`). [1], [2]

3. **Visualizing the mean and standard deviation**

- A simple way to visualize this data is to create a **line plot** for the mean miles per gallon with **error bars** indicating the standard deviation. Here is an example of how we could do this with `ggplot2`:

```r
# Loading the ggplot2 package for data visualization.
suppressPackageStartupMessages(library(ggplot2))

# Creating a line plot with error bars to visualize the Mean and SD.
ggplot(s1,
       aes(x = cyl, y = Mean_mpg)) +  # Defining the x and y aesthetics.
  geom_line(group = 1, color = "blue") +  # blue line connecting points.
  geom_point(size = 2, color = "red") +  # red points for the mean
  geom_errorbar(aes(ymin = Mean_mpg - SD_mpg,
                    ymax = Mean_mpg + SD_mpg),
                width = 0.2, colour = "black") +  # Adding error bars
  labs(x = "Cylinders", y = "Mean mpg",  # Labeling the axes
       title = "Mean and SD of Mileage (mpg) by #Cylinders") +
  theme_minimal()  # Applying a minimal theme to the plot.
```



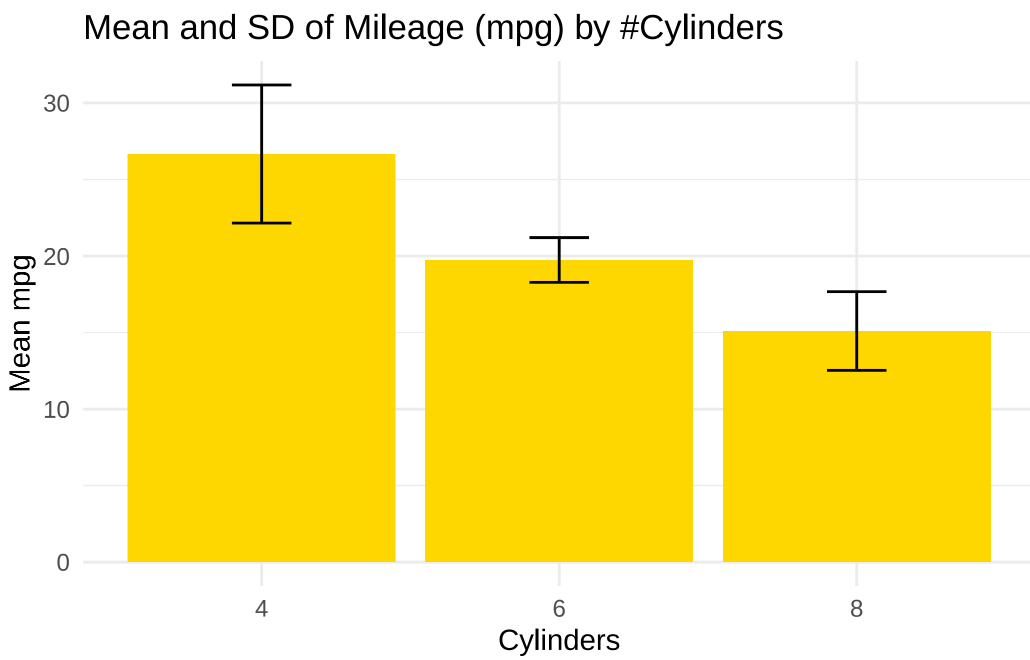
4. **Discussion**:

- `aes(x = cyl, y = Mean_mpg)` assigns the `cyl` values to the x-axis and `Mean_mpg` to the y-axis.
- `geom_line(group=1, color = "blue")` adds a blue line connecting the data points.
- `geom_point(size = 2, color = "red")` adds red points for each data point.
- `geom_errorbar(aes(ymin = Mean_mpg - SD_mpg, ymax = Mean_mpg + SD_mpg), width = .2, colour = "black")` adds error bars, where the error is the standard deviation.
- The `ymin` and `ymax` arguments define the range of the error bars.
- `labs(x = "Cylinders", y = "Mean mpg")` labels the x and y axes.
- `theme_minimal()` applies a minimal theme to the plot. [1], [2]

5. **Visualizing the mean and standard deviation - Alternate Method**

- An alternate method is to visualize this mean by creating a **bar plot**, with **error bars** indicating the standard deviation. Here is an example of how we could do this with `ggplot2`:

```r
# Loading the ggplot2 package for data visualization.
library(ggplot2)

# Creating a bar plot with error bars to visualize the Mean and SD.
ggplot(s1,
       aes(x = cyl, y = Mean_mpg)) +  # Defining the x and y aesthetics.
  geom_bar(stat = "identity",
           fill = "gold") +  # Creating a bar plot with gold-colored bars.
  geom_errorbar(aes(ymin = Mean_mpg - SD_mpg,
                    ymax = Mean_mpg + SD_mpg),
                width = 0.2) +  # Adding error bars .
  labs(x = "Cylinders", y = "Mean mpg",  # Labeling the axes
       title = "Mean and SD of Mileage (mpg) by #Cylinders") +
  theme_minimal()  # Applying a minimal theme to the plot.
```



6. **Discussion**:

- `ggplot(s1, aes(x = cyl, y = Mean_mpg))`: The `ggplot()` function initializes a ggplot object using dataframe `s1` and mapping aesthetic elements. Here, `aes(x = cyl, y = Mean_mpg)` specifies that the x-axis represents `cyl` (number of cylinders) and the y-axis represents `Mean_mpg` (mean miles per gallon).
- `geom_bar(stat = "identity", fill = "gold")`: The `geom_bar()` function is used to create a bar chart. Setting `stat = "identity"` indicates that the heights of the bars

represent the values in the data (in this case, `Mean_mpg`). The `fill = "gold"` argument sets the color of the bars.

- `geom_errorbar()` adds error bars to the plot. The arguments `aes(ymin = Mean_mpg - SD_mpg, ymax = Mean_mpg + SD_mpg)` set the bottom (`ymin`) and top (`ymax`) of the error bars to represent one standard deviation below and above the mean, respectively. `width = .2` sets the horizontal width of the error bars.
- `labs(x = "Cylinders", y = "Mean mpg")`: The `labs()` function is used to specify the labels for the x-axis and y-axis.
- `theme_minimal()`: The `theme_minimal()` function is used to set a minimalistic theme for the plot. [1], [2]

7. We extend this code to demonstrate how to measure the bivariate relationships between multiple continuous variables from the mtcars data and the categorical variable number of Cylinders (`cyl`), using `ggplot2`. Specifically, we want to measure the mean and SD of continuous variables (i) Miles Per Gallon (`mpg`); (ii) Weight (`wt`); (iii) Horsepower (`hp`) across the number of Cylinders (`cyl`).

```
# Loading the dplyr package for data manipulation.
library(dplyr)

# Calculating summary statistics for Mileage (mpg), Weight (wt), and
# Horsepower (hp) grouped by Cylinders (cyl).
s3 <- tb %>%
  group_by(cyl) %>%
  summarise(
    Mean_mpg = mean(mpg, na.rm = TRUE),  # mean of Mileage (mpg).
    SD_mpg = sd(mpg, na.rm = TRUE),  # standard deviation of Mileage
    Mean_wt = mean(wt, na.rm = TRUE),  # mean of Weight (wt).
    SD_wt = sd(wt, na.rm = TRUE),  # standard deviation of Weight (wt).
    Mean_hp = mean(hp, na.rm = TRUE),  # mean of Horsepower (hp).
    SD_hp = sd(hp, na.rm = TRUE)  # standard deviation of (hp).
  )

# Creating a table (kable) to display the summary statistics.
kable(s3,
      digits = 2,
      caption = "Summary of Mileage, Weight, Horsepower by Cylinders")
```

Table 13.2: Summary of Mileage, Weight, Horsepower by Cylinders

cyl	Mean_mpg	SD_mpg	Mean_wt	SD_wt	Mean_hp	SD_hp
4	26.66	4.51	2.29	0.57	82.64	20.93
6	19.74	1.45	3.12	0.36	122.29	24.26
8	15.10	2.56	4.00	0.76	209.21	50.98

8. **Discussion**:

- With `tb %>%`, we indicate that we are going to perform a series of operations on the `tb` data frame. The `group_by(cyl)` groups the data by the `cyl` variable.
- The `summarise()` function calculates the mean and standard deviation (SD) of three variables (`mpg`, `wt`, and `hp`). The`na.rm = TRUE` argument inside `mean()` and `sd()` functions is used to exclude any NA values from these calculations.
- The resulting calculations are assigned to new variables (`Mean_mpg`, `SD_mpg`, `Mean_wt`, `SD_wt`, `Mean_hp`, and `SD_hp`) which will be the columns in the summarised data frame.
- To summarize, this script groups the data in the `tb` tibble by `cyl` and then calculates the mean and standard deviation of the `mpg`, `wt`, and `hp` variables for each group. [1], [2]

13.2.2 Across two Categories using `ggplot2`

1. We demonstrate the relationship between Miles Per Gallon (`mpg`) and Cylinders (`cyl`) and Transmission type (`am`) using `ggplot2`. Recall that a car's transmission may be automatic (`am=0`) or manual (`am=1`). [1], [2]

```r
# Loading the dplyr package for data manipulation.
library(dplyr)

# Calculating summary statistics for Mileage (mpg)
# grouped by Cylinders (cyl) and Transmission (am).
s4 <- tb %>%
  group_by(cyl, am) %>%
  summarise(
    Mean_mpg = mean(mpg, na.rm = TRUE),  # Calculating the mean.
    SD_mpg = sd(mpg, na.rm = TRUE)  # Calculating the standard deviation
  )
```

```
`summarise()` has grouped output by 'cyl'. You can override using the `.groups`
argument.
```

```r
# Creating a table (kable) to display the summary statistics
kable(s4,
      digits = 2,
      caption = "Summary of Mileage (mpg) by Cylinders and Transmission")
```

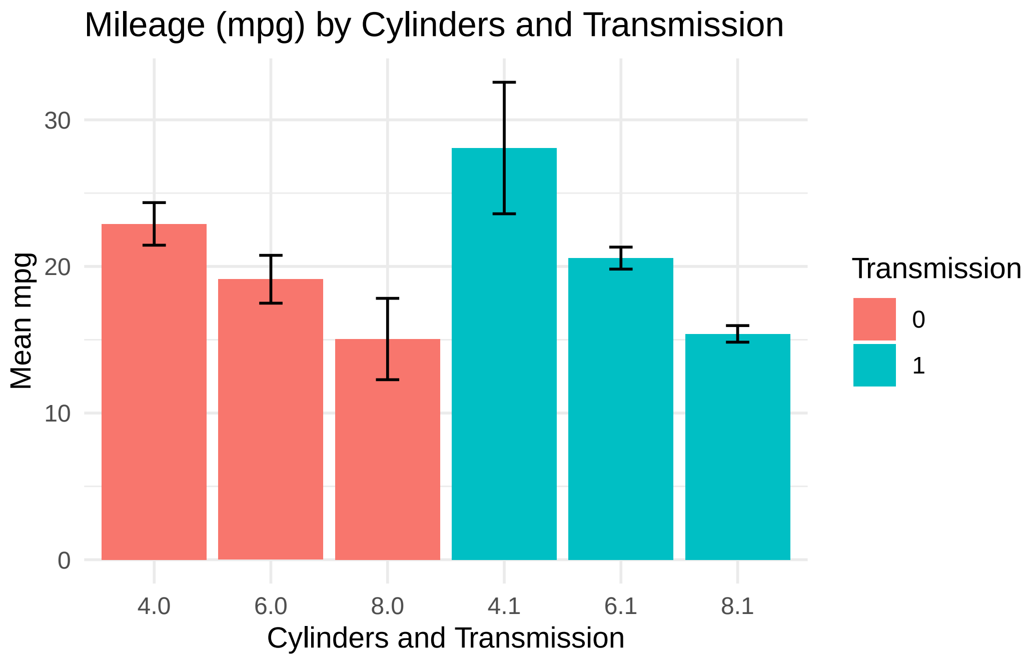
Table 13.3: Summary of Mileage (mpg) by Cylinders and Transmission

cyl	am	Mean_mpg	SD_mpg
4	0	22.90	1.45
4	1	28.08	4.48
6	0	19.12	1.63
6	1	20.57	0.75
8	0	15.05	2.77
8	1	15.40	0.57

2. **Discussion**:

- The above code provides the mean and standard deviation of `mpg` for each unique combination of `cyl` and `am`. [1], [2]
- Here is how it can be visualized:

```r
# Create the plot using ggplot2
# Create an interaction variable for 'cyl' and 'am'
ggplot(s4, aes(x = interaction(cyl, am),
               y = Mean_mpg,  # Set 'Mean_mpg' as the y-axis variable
               fill = as.factor(am))) +  # Fill bars by the 'am' factor
  geom_bar(stat = "identity",  # Use the "identity" statistic to plot the bars
           position = position_dodge()) +  # Dodge the bars for each 'am' level
  geom_errorbar(aes(ymin = Mean_mpg - SD_mpg,  # Add error bars
                    ymax = Mean_mpg + SD_mpg),
                width = .2,  # Set the width of error bars
                position = position_dodge(.9)) +  # Dodge error bars
  labs(x = "Cylinders and Transmission",  # Set x-axis label
       y = "Mean mpg",  # Set y-axis label
       fill = "Transmission",  # Set legend title for fill color
       title = "Mileage (mpg) by Cylinders and Transmission") +
  theme_minimal()  # Use the minimal theme for the plot
```



3. In the below code, the order of the variables is reversed - the data is first grouped by `am`, then by `cyl`. So, the function first sorts the data by the `am` variable, and within each `am` group, it further groups the data by `cyl`. [1], [2]

```
# Load the dplyr library for data manipulation
library(dplyr)

# Group the dataframe by 'am', 'cyl' columns and calculate summary statistics
s5 <- tb %>%
  group_by(am, cyl) %>%
  summarise(Mean_mpg = mean(mpg, na.rm = TRUE),  # Calculate mean 'mpg'
            SD_mpg = sd(mpg, na.rm = TRUE))  # Calculate standard deviation
```

```
`summarise()` has grouped output by 'am'. You can override using the `.groups`
argument.
```

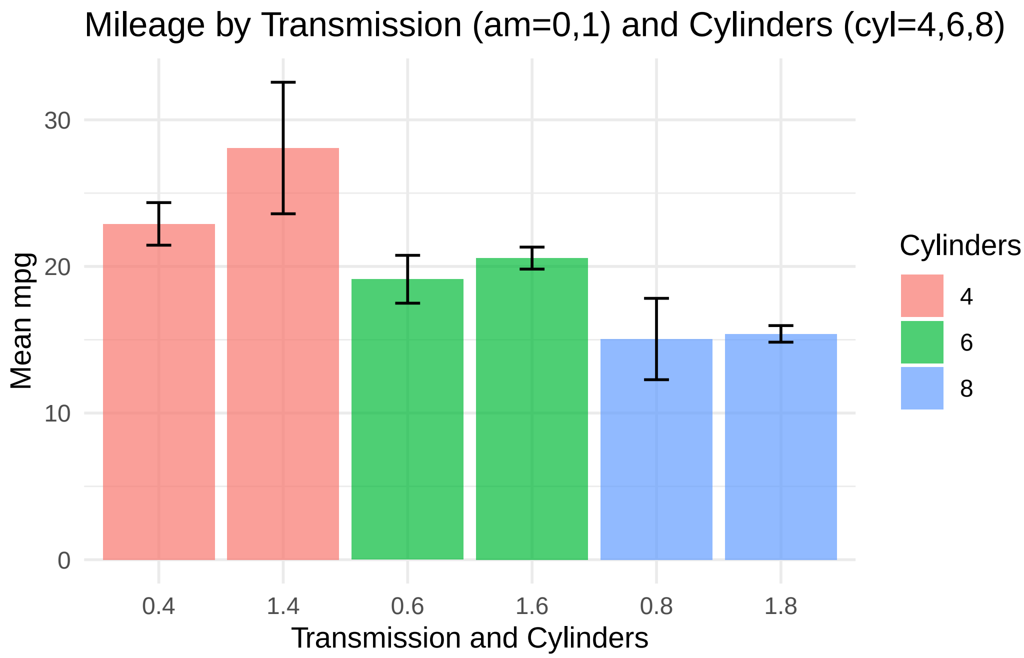
```
# Create a table (kable) of the summary statistics with specified formatting
kable(s5,
      digits = 2,  # Set the number of digits to display
      caption = "Summary of Mileage (mpg) by Transmission and Cylinders")
```

Table 13.4: Summary of Mileage (mpg) by Transmission and Cylinders

am	cyl	Mean_mpg	SD_mpg
0	4	22.90	1.45
0	6	19.12	1.63
0	8	15.05	2.77
1	4	28.08	4.48
1	6	20.57	0.75
1	8	15.40	0.57

- Here is how it can be visualized:

```
# Create the plot using ggplot2
ggplot(s5,
       aes(x = interaction(am, cyl),  #  'am' and 'cyl' as the x-axis variable
           y = Mean_mpg,  # Mean 'mpg' as the y-axis variable
           fill = as.factor(cyl))) +  # Fill the bars by 'cyl' (Cylinders)

  geom_bar(stat = "identity",  # Create a bar plot with actual data values
           alpha = 0.7,  # Set the transparency of bars to 0.7
           position = position_dodge()) +  # Dodge bars for better visualization

  geom_errorbar(aes(ymin = Mean_mpg - SD_mpg,  # Add error bars
                    ymax = Mean_mpg + SD_mpg),
                width = .2,  # Set the width of the error bars
                position = position_dodge(.9)) +  # Dodge error bars

  labs(x = "Transmission and Cylinders",  # Set the x-axis label
       y = "Mean mpg",  # Set the y-axis label
       fill = "Cylinders",  # Set the legend label for fill color
       title = "Mileage by Transmission (am=0,1) and Cylinders (cyl=4,6,8)") +

  theme_minimal()  # Use a minimal theme for the plot
```



4. The following code produces a new data frame that contains the mean and standard deviation of the continuous variables `mpg`, `wt`, and `hp` for each combination of the factor variables `am` and `cyl`. [1], [2]

```
# Summary statistics 'mpg', 'wt', 'hp' by Transmission (am) and Cylinders (cyl)
s6 <- tb %>%
  group_by(am, cyl) %>%
  summarise(
    Mean_mpg = mean(mpg, na.rm = TRUE),  # Calculate mean 'mpg'
    SD_mpg = sd(mpg, na.rm = TRUE),  # Calculate standard deviation of 'mpg'
    Mean_wt = mean(wt, na.rm = TRUE),  # Calculate mean 'wt' (weight)
    SD_wt = sd(wt, na.rm = TRUE),  # Calculate standard deviation of 'wt'
    Mean_hp = mean(hp, na.rm = TRUE),  # Calculate mean 'hp' (horsepower)
    SD_hp = sd(hp, na.rm = TRUE)  # Calculate standard deviation of 'hp'
    )
```

```
`summarise()` has grouped output by 'am'. You can override using the `.groups`
argument.
```

```
# Create a table (kable) to display the summary statistics
kable(s6,
      digits=2,  # Set the number of decimal digits to display
      caption = "mpg, wt, hp by am (am=0,1), cyl (cyl=4,6,8)")
```

Table 13.5: mpg, wt, hp by am (am=0,1), cyl (cyl=4,6,8)

am	cyl	Mean_mpg	SD_mpg	Mean_wt	SD_wt	Mean_hp	SD_hp
0	4	22.90	1.45	2.94	0.41	84.67	19.66
0	6	19.12	1.63	3.39	0.12	115.25	9.18
0	8	15.05	2.77	4.10	0.77	194.17	33.36
1	4	28.08	4.48	2.04	0.41	81.88	22.66
1	6	20.57	0.75	2.76	0.13	131.67	37.53
1	8	15.40	0.57	3.37	0.28	299.50	50.20

13.3 Summary of Chapter 13 – Bivariate Continuous data (Part 2 of 4)

In this chapter, we delve into the analysis of categorical and continuous data using the versatile R packages `dplyr` and `ggplot2`. To begin, we prepare the `mtcars` dataset and save it as a tibble named `tb`. After converting relevant columns into factor variables, we can directly access the data columns without referencing `tb$` for each variable.

We then explore various visualization techniques for continuous data within one category, employing `ggplot2`. These techniques include Bee Swarm plots, Histograms, Probability Density Functions (PDF), Cumulative Density Functions (CDF), Box plots, and Violin plots. For instance, we create a Bee Swarm plot to visualize the median weight of cars categorized by the number of cylinders (`cyl`). Additionally, we construct histograms to display car mileage (`mpg`) breakdown by transmission type (`am`), using separate histograms for each transmission category for easier comparison. The PDF and CDF plots showcase the distribution of `mpg` with distinct colors representing transmission types. Box plots and Violin plots provide insights into mileage distribution across different cylinder counts and transmission types.

We then delve into summarizing continuous data using `dplyr` and `ggplot2`. Within one category, such as the number of cylinders (`cyl`), we calculate and visualize summary statistics like the mean and standard deviation of car mileage (`mpg`). We generate line plots and bar plots with error bars, highlighting the relationship between `cyl` and the mean `mpg`. An alternative method of measuring bivariate relationships extends the analysis to multiple continuous variables, namely `mpg`, `wt` (weight), and `hp` (horsepower) across `cyl`. A similar approach is taken for relationships across two categories, where we explore the interaction between `cyl` and transmission type (`am`). This analysis provides a comprehensive understanding of how to summarize and visualize continuous data.

13.4 References

Basic R Programming:

[1] Chambers, J. M. (2008). Software for Data Analysis: Programming with R (Vol. 2, No. 1). Springer.

Crawley, M. J. (2012). The R Book. John Wiley & Sons.

Gardener, M. (2012). Beginning R: The Statistical Programming Language. John Wiley & Sons.

Grolemund, G. (2014). Hands-On Programming with R: Write Your Own Functions and Simulations. O'Reilly Media, Inc.

Kabacoff, R. (2022). R in Action: Data Analysis and Graphics with R and Tidyverse. Simon and Schuster.

Peng, R. D. (2016). R Programming for Data Science (pp. 86-181). Leanpub.

R Core Team. (2020). R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing. Retrieved from https://www.R-project.org/.

Tippmann, S. (2015). Programming Tools: Adventures with R. Nature, 517(7532), 109-110.

Wickham, H., Çetinkaya-Rundel, M., & Grolemund, G. (2023). R for Data Science. O'Reilly Media, Inc.

ggplot2:

[2] Wickham, H. (2016). ggplot2: Elegant Graphics for Data Analysis. Springer-Verlag New York. Retrieved from https://ggplot2.tidyverse.org

Wickham, H., & Grolemund, G. (2016). R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. O'Reilly Media.

Wickham, H. (2020). ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics (Version 3.3.2) [Computer Software]. Retrieved from https://CRAN.R-project.org/package=ggplot2

Wickham, H., et al. (2020). dplyr: A Grammar of Data Manipulation (Version 1.0.2) [Computer Software]. Retrieved from https://CRAN.R-project.org/package=dplyr

Wilkinson, L. (2005). The Grammar of Graphics (2nd ed.). Springer-Verlag.

Wickham, H., et al. (2020). tibble: Simple Data Frames (Version 3.0.3) [Computer Software]. Retrieved from https://CRAN.R-project.org/package=tibble

beeswarm:

[3] Eklund, A. (2020). ggbeeswarm: Categorical Scatter (Violin Point) Plots. R Package Version 0.6.0. Retrieved from https://CRAN.R-project.org/package=ggbeeswarm

Histograms:

[4] Scott, D. W. (1979). On Optimal and Data-Based Histograms. Biometrika, 66(3), 605-610.

Wand, M. P., & Jones, M. C. (1995). Kernel Smoothing. Chapman and Hall/CRC.

ggpubr:

[5] Kassambara, A. (2023). ggpubr: 'ggplot2' Based Publication Ready Plots. R Package Version 0.6.0. Retrieved from https://rpkgs.datanovia.com/ggpubr/.

Box Plots:

[6] McGill, R., Tukey, J. W., & Larsen, W. A. (1978). Variations of Box Plots. The American Statistician, 32(1), 12-16.

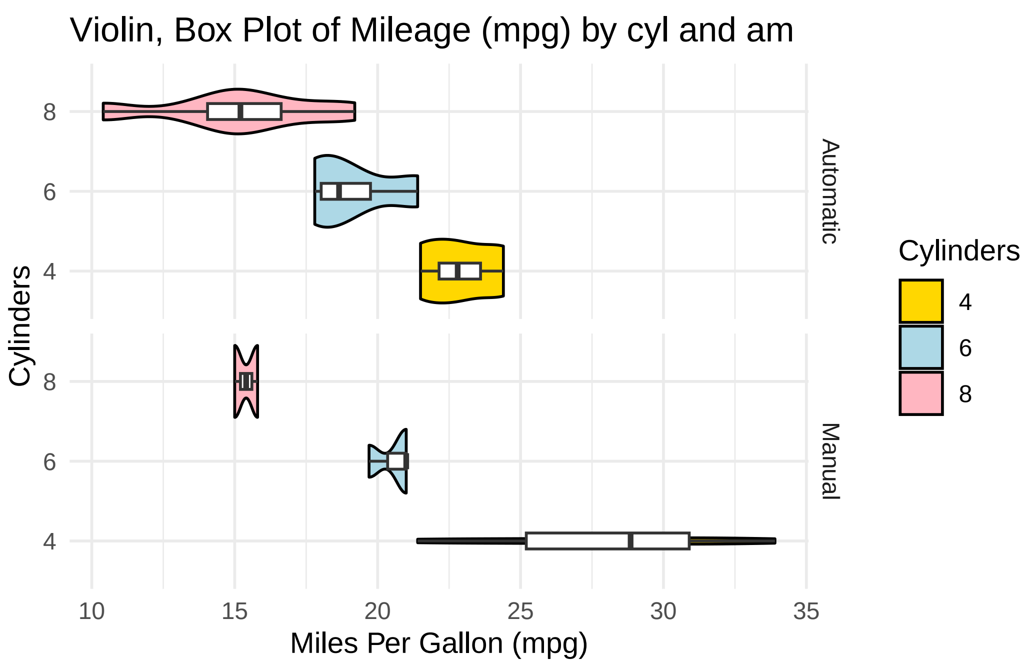
13.5 Appendix A

13.5.1 Appendix A1: Violin Plot across two Categories using `ggplot2`

- We can embed boxplots within the above Violin plots, as follows.

```
# Load the ggplot2 library
library(ggplot2)

# Violin plot and box plot of 'mpg' by 'cylinders' and 'am' (Transmission)
ggplot(tb, aes(x = factor(cyl), y = mpg)) +
  geom_violin(aes(fill = factor(cyl)), color = "black") +
  scale_fill_manual(values = c("gold", "lightblue", "lightpink"),
                    name = "Cylinders") +
  geom_boxplot(width = 0.2,
               fill = "white") +  # A box plot with specified width, color
  coord_flip() +  # Flip the coordinates to create horizontal plots
  labs(title = "Violin, Box Plot of Mileage (mpg) by cyl and am",
       y = "Miles Per Gallon (mpg)",  # Label for the y-axis
       x = "Cylinders") +  # Label for the x-axis
  facet_grid(am ~ .,
             scales = "free_y",
             space = "free_y",
             # Create facets for 'am' (Transmission), scales and spacing
             labeller = labeller(
```

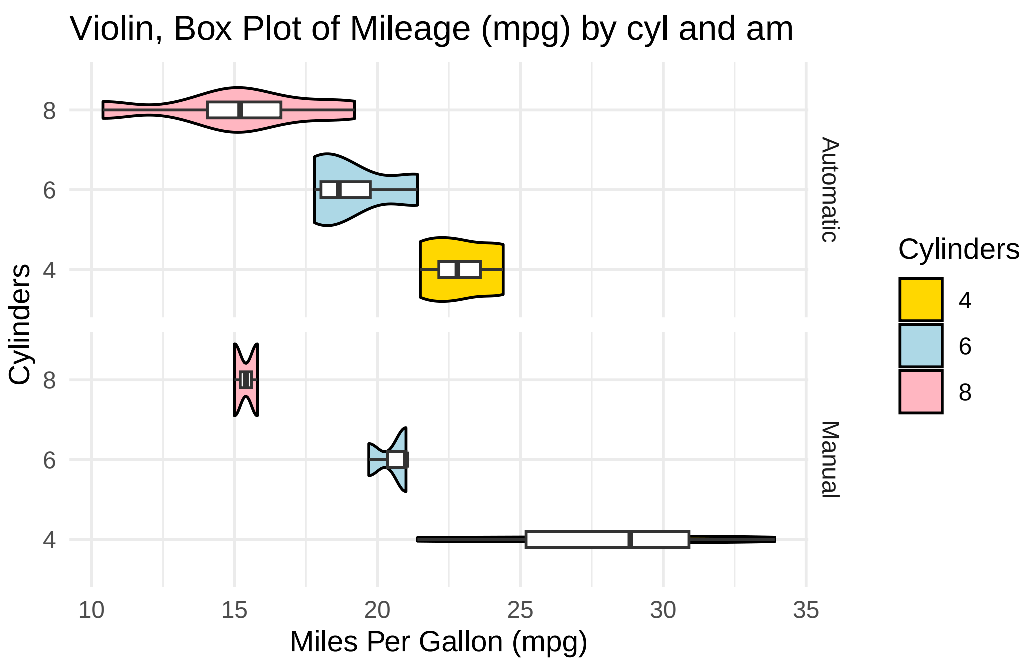
```
                am = function(x) ifelse(x == 0, "Automatic", "Manual")
                )) +
  theme_minimal()  # Use a minimal theme for the plot
```



- Alternately, We can embed boxplots within the above Violin plots, as follows.

```
# Load the ggplot2 and dplyr libraries
library(ggplot2)
library(dplyr)

# Modify the data first: Convert 'am' to factor with custom labels
tb_modified <- tb %>%
  mutate(am = factor(am, levels = c(0, 1), labels = c("Automatic", "Manual")))

# Create the plot
ggplot(tb_modified,
       aes(x = factor(cyl),
           y = mpg)) +
  geom_violin(aes(fill = factor(cyl)),
              color = "black") +  # Create a violin plot by 'cylinders'
  scale_fill_manual(values = c("gold", "lightblue", "lightpink"),
                    name = "Cylinders") +  # Set custom fill colors
  geom_boxplot(width = 0.2, fill = "white") +  # Create a box plot
```

```r
  coord_flip() +  # Flip the coordinates to create horizontal plots
  labs(title = "Violin, Box Plot of Mileage (mpg) by cyl and am",
       y = "Miles Per Gallon (mpg)",  # Label for the y-axis
       x = "Cylinders") +  # Label for the x-axis
  facet_grid(am ~ .,
             scales = "free_y",
             space = "free_y") +  # Create facets for 'am' (Transmission)
  theme_minimal()  # Use a minimal theme for the plot
```



14 Bivariate Continuous data (Part 3 of 4)

Chapter 14.

14.1 Exploring bivariate Continuous x Continuous data

This chapter explores how to summarize and visualize the interaction between *bivariate continuous data* using correlation analysis, scatter plots, scatter plot matrices and other such techniques. [1]

Data: Suppose we run the following code to prepare the `mtcars` data for subsequent analysis and save it in a tibble called `tb`.

```
# Load the required libraries, suppressing annoying startup messages
library(dplyr, quietly = TRUE, warn.conflicts = FALSE)
library(tibble, quietly = TRUE, warn.conflicts = FALSE)
library(knitr, quietly = TRUE, warn.conflicts = FALSE)
library(ggplot2, quietly = TRUE, warn.conflicts = FALSE)
library(car, quietly = TRUE, warn.conflicts = FALSE)
library(psych, quietly = TRUE, warn.conflicts = FALSE)
library(GGally, quietly = TRUE, warn.conflicts = FALSE)
```

```
Registered S3 method overwritten by 'GGally':
  method from
  +.gg   ggplot2
```

```
# Read the mtcars dataset into a tibble called tb
data(mtcars)
tb <- as_tibble(mtcars)

# Convert relevant columns into factor variables
tb$cyl <- as.factor(tb$cyl) # cyl = {4,6,8}, number of cylinders
tb$am <- as.factor(tb$am) # am = {0,1}, 0:automatic, 1: manual transmission
tb$vs <- as.factor(tb$vs) # vs = {0,1}, v-shaped engine, 0:no, 1:yes
tb$gear <- as.factor(tb$gear) # gear = {3,4,5}, number of gears
```

```r
# Directly access the data columns of tb, without tb$mpg
attach(tb)
```

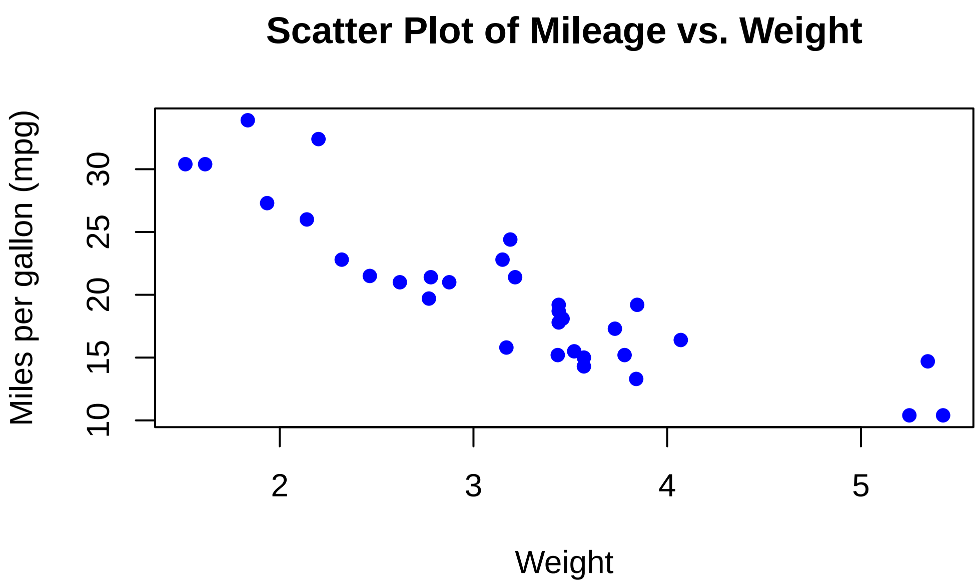
```
The following object is masked from package:ggplot2:

    mpg
```

14.2 Scatterplots

1. A scatter plot is used to display the relationship between **two continuous variables**. It is a graphical representation of a bivariate distribution, where the values of two variables are plotted as points on a two-dimensional coordinate system.
2. A scatter plot can be used to **identify trends, clusters**, and other patterns in the data. It is also useful for detecting the presence of any **outliers** or **influential observations** that may affect the analysis. [1]
3. To create a scatter plot of `mpg` (miles per gallon) against `wt` (weight) in the `mtcars` data set, we can use the following code:

14.2.1 Scatterplot using `plot()`

```r
# Create a scatter plot of Miles per Gallon (mpg) vs. Weight
plot(wt,
     mpg,
     main = "Scatter Plot of Mileage vs. Weight", # Set the title
     xlab = "Weight",                 # Label for the x-axis
     ylab = "Miles per gallon (mpg)",# Label for the y-axis
     pch = 16,                        # Use filled circles as data points
     col = "blue"                     # Set the color of the points to blue
     )
```



4. **Discussion**:

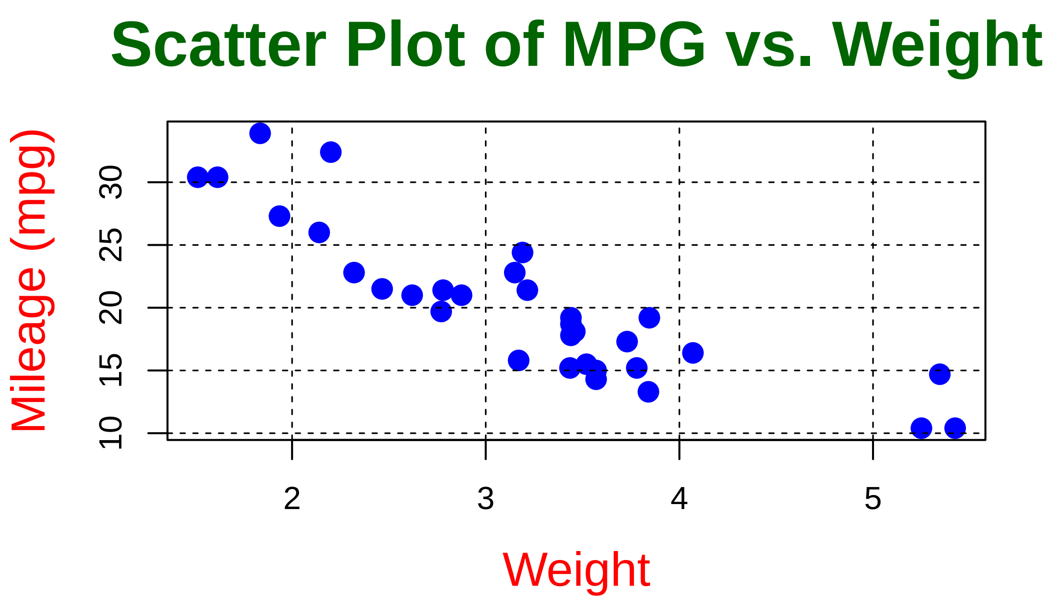
- This code will create a scatter plot of `mpg` against `wt` using the `plot()` function.
- The `main` argument adds a title to the plot, the `xlab` and `ylab` arguments add axis labels
- The `pch` argument sets the shape of the points to a solid circle. `pch = 15` gives a filled square. Recall other popular values: `pch = 16` gives a filled circle, `pch = 17` gives a filled triangle (pointing upwards), `pch = 18` gives a filled diamond, `pch = 19` gives a solid circle, `pch = 20` gives a filled bullet (smaller than `pch = 19`)
- the `col` argument specifies the color of the data points. Recall we can use any named color in R, or we can use hexadecimal color codes. For instance, `col = "#FF0000"` would give us red points. [1]

5. **Personalizing Scatter Plots**

- We can personalize the appearance of the scatterplot in a variety of additional ways.

```
# Create a scatter plot of Mileage (mpg) vs. Weight
plot(wt,
     mpg,
     main = "Scatter Plot of MPG vs. Weight",  # Set the plot title
     xlab = "Weight",          # Label for the x-axis
     ylab = "Mileage (mpg)", # Label for the y-axis
     pch = 16,                 # Use filled circles as data points
     cex = 1.5,                # Increase the size of data points
     col = "blue",             # Set the color of data points to blue
```

```r
     col.lab = "red",          # Set the color of axis labels to red
     cex.lab = 1.5,            # Increase the size of axis labels
     col.main = "darkgreen", # Set the color of the plot title to dark green
     cex.main = 2,             # Increase the size of the plot title
     bg = "gray"               # Set the background color to gray
     )
# Add a grid with black dashed lines of width 0.8
grid(col = "black",
     lty = "dashed",
     lwd = 0.8)
```



6. **Discussion**

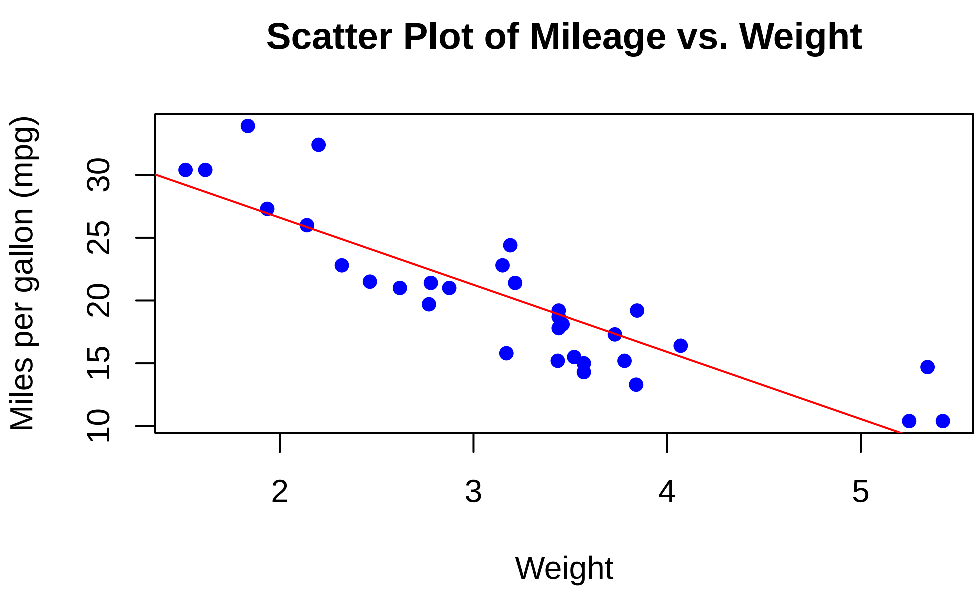
- **Point Size**: In the second plot, the size of the points is 1.5 times the default size (`cex = 1.5`), while in the first plot, the size of the points is the default size as `cex` is not specified.
- **Axis Labels' Color and Size**: The second plot has red-colored, larger size axis labels (`col.lab="red", cex.lab=1.5`), while the first plot uses the default color and size as these parameters are not specified.
- **Title's Color and Size**: The second plot has a dark green title that is twice the default size (`col.main="darkgreen", cex.main=2`), while the first plot uses the default color and size for the title as these parameters are not specified.
- **Background Color**: The second plot has a light gray background (`bg = "lightgray"`), while the first plot uses the default background color as the bg parameter is not specified.

- **Grid**: The second plot includes a grid with gray dotted lines (`grid(col = "gray", lty = "dotted", lwd = 0.5)`), while the first plot does not have a grid as the grid() function is not called. [2]

7. **Scatterplot with best fit line**

- We can add a line of best fit (a regression line) to your scatterplot using the `abline()` and `lm()` functions. `lm()` is used to fit linear models, and `abline()` adds a straight line to the plot. [3]

```
# Create a scatter plot of Miles per gallon (mpg) vs. Weight
# with blue data points
plot(tb$wt,
     tb$mpg,
     main = "Scatter Plot of Mileage vs. Weight",  # Set the plot title
     xlab = "Weight",                        # Label for the x-axis
     ylab = "Miles per gallon (mpg)",        # Label for the y-axis
     pch = 16,                               # Use filled circles as data points
     col = "blue"                            # Set the color of data points to blue
     )

# Fit a linear model to the data
fit <- lm(tb$mpg ~ tb$wt)

# Add a red regression line to the plot
abline(fit, col = "red")
```



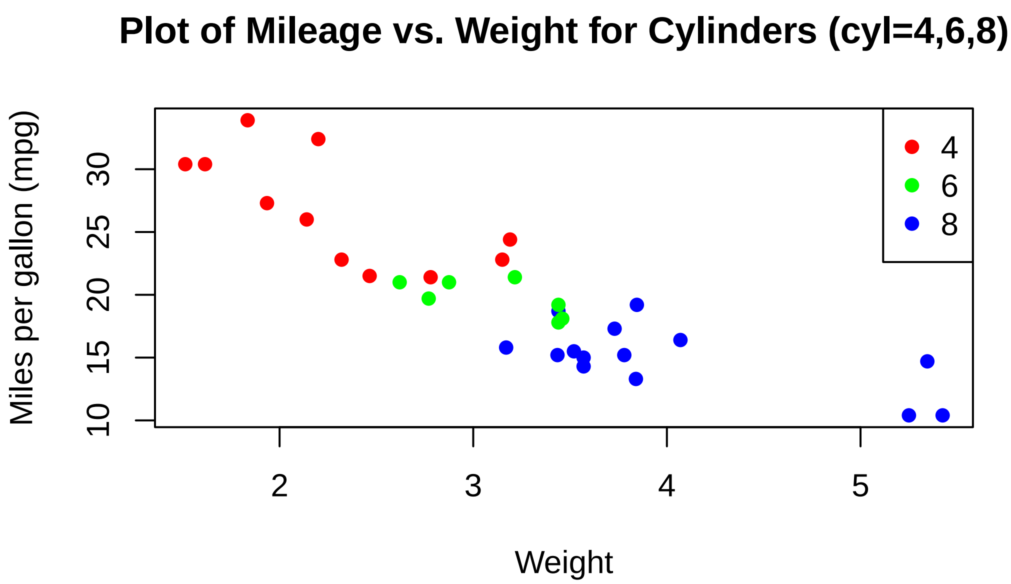
8. **Discussion**:

- `lm(tb$mpg ~ tb$wt)` fits a linear model predicting `mpg` from `wt`. The `~` operator is a formula operator in R that separates the response variable (on the left of `~`) from the predictor variables (on the right of `~`). `abline(fit, col = "red")` subsequently adds the regression line to the plot, drawn in red color. [3]

9. **Visualizing Continuous x Continuous x Categorical data**

- Consider the issue of visualizing two continuous variables `wt` (Weight and `mpg` (Miles per gallon) for different levels of a categorical variable `cyl` (Cylinders).

```
# Create a scatter plot of Miles per gallon (mpg) vs. Weight,
# where points are colored by the number of cylinders (cyl)
plot(tb$wt,
     tb$mpg,
     main = "Plot of Mileage vs. Weight for Cylinders (cyl=4,6,8)",
     xlab = "Weight",
     ylab = "Miles per gallon (mpg)",
     pch = 16,  # Use filled circles as data points
     col = c("red", "green", "blue")[tb$cyl]  # Color points on cyl
     )

# Add a legend to the top-right corner of the plot
legend("topright",
       legend = levels(tb$cyl),  # Display legend labels for each cylinder
       col = c("red", "green", "blue"),  # Define colors for the legend
       pch = 16)  # Use filled circles as legend symbols
```



10. Discussion

- In the snippet `col=c("red","green","blue")[tb$cyl]`, we assign the color of the data points according to the `cyl` variable. It translates to distinct colors for different `cyl` values in the plot.
- Moreover, when we incorporate the `legend()` function with parameters such as `"topright"`, we place a legend at the top-right corner of the plot. The identifiers and color scheme in the legend correspond to the unique values of the `cyl` variable. [2]

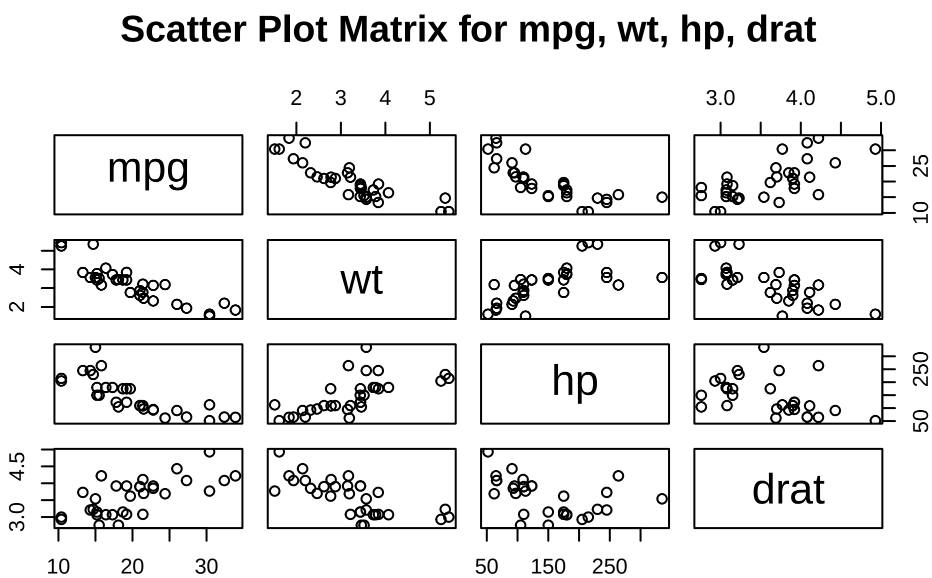
14.3 Scatterplot Matrix

1. A scatter plot matrix, also known as a pairs plot or **SPLOM**, is a powerful visual tool that helps us in depicting the **pair-wise relationships** among a group of variables. In this matrix, every distinct variable from the dataset is charted against each other in a grid-like structure. This enables us to delve into the associations between variable pairs and identify possible trends or patterns in the dataset.
2. When dealing with multivariate datasets, scatter plot matrices can be remarkably handy. They equip us with a rapid way to **discern potential correlations among variable pairs** – whether they are strong, weak, or non-existent. It's also a convenient method to spot non-linear relationships between variables. In addition, it's a beneficial tool to recognize outliers or peculiar data points and to observe clusters or collections of observations. [2], [3]

14.3.1 Scatterplot Matrix using `pairs()`

1. The following R code creates a scatter plot matrix, also known as a pairs plot, for four variables: `mpg` (miles per gallon), `wt` (weight), `hp` (horsepower), and `drat` (rear axle ratio). This is performed on the data stored in the `tb` data frame.

```
# Define the columns we want to include in the scatter plot matrix
columns <- c("mpg", "wt", "hp", "drat")

# Create a scatter plot matrix using the selected columns from the data frame
pairs(tb[, columns],
      main = "Scatter Plot Matrix for mpg, wt, hp, drat" # Set the title
)
```

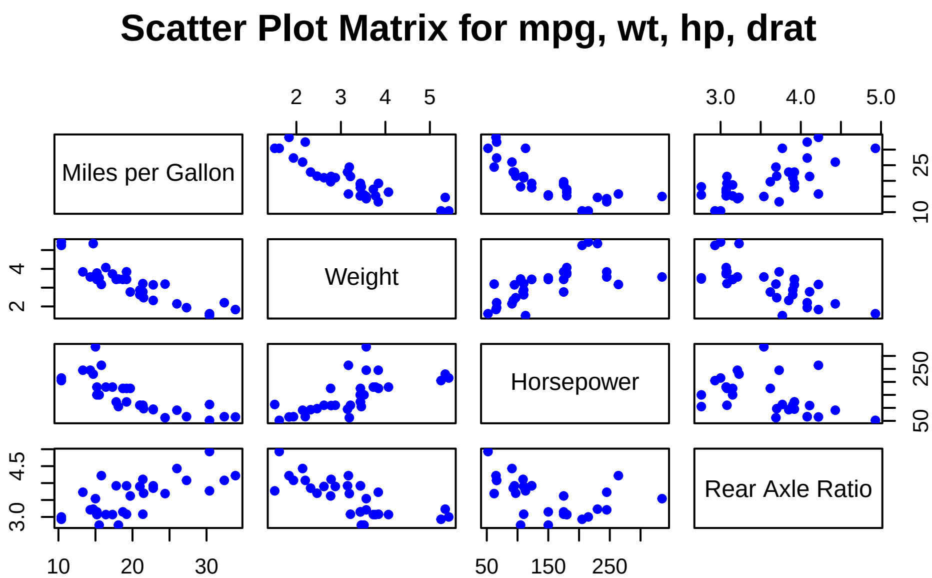


2. **Discussion**:

- The `pairs()` function in R is designed to take a subset of the `tb` data frame consisting of the columns specified in the columns vector and construct a matrix of scatter plots.
- The plots are arranged in a grid format, where the variable for each row is plotted against the variable for each column. [2], [3]

3. **Personalizing Scatterplot Matrix using `pairs()`**

```
# Create a scatter plot matrix for the columns "mpg," "wt," "hp," and "drat"
pairs(tb[, c("mpg", "wt", "hp", "drat")],  # Subset the data
      main = "Scatter Plot Matrix for mpg, wt, hp, drat",  # title
      pch = 19,  # Set the point character (solid circles)
      labels = c("Miles per Gallon", "Weight", "Horsepower", "Rear Axle Ratio"),
      col = "blue",  # Set point color to blue
      cex = 0.8  # Set the size of the points
      )
```



4. **Discussion**:

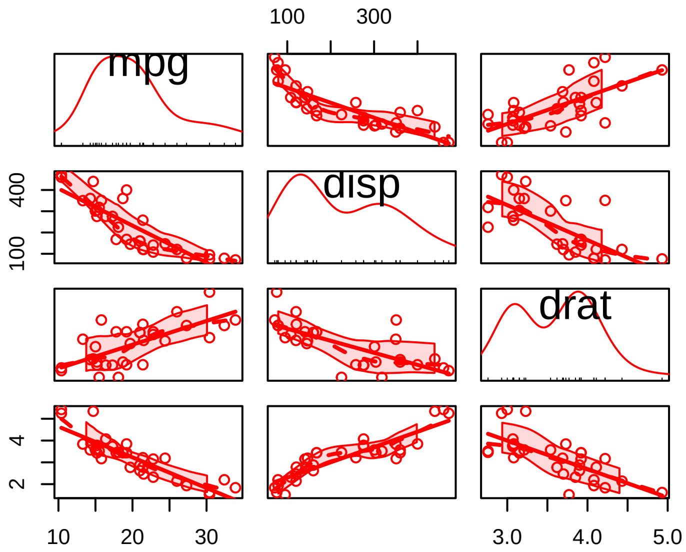
- **Symbol**: The `pch` parameter is used to specify the symbol that represents data points in a plot.
- **Labels**: The `labels` parameter lets us customize the variable labels that appear on the diagonal:
- **Color**: We can specify different colors for the points in each scatter plot with the `col` parameter.
- **Point Size**: The `cex` parameter controls the size of the points in the scatter plot. [2], [3]

14.3.2 Scatter Plot Matrix using `scatterplotMatrix()` from package `car`

1. The following code creates an alternate scatterplot matrix for each pair of the variables `mpg`, `disp`, `drat`, and `wt`, allowing us to explore the relationships among these four

variables. [6]

```r
# Load the car package
library(car)

# Create a scatterplot matrix using scatterplotMatrix()
scatterplotMatrix(data = tb,
                  ~ mpg + disp + drat + wt,  # variables
                  col = c("red"))  # Set the point color to red
```

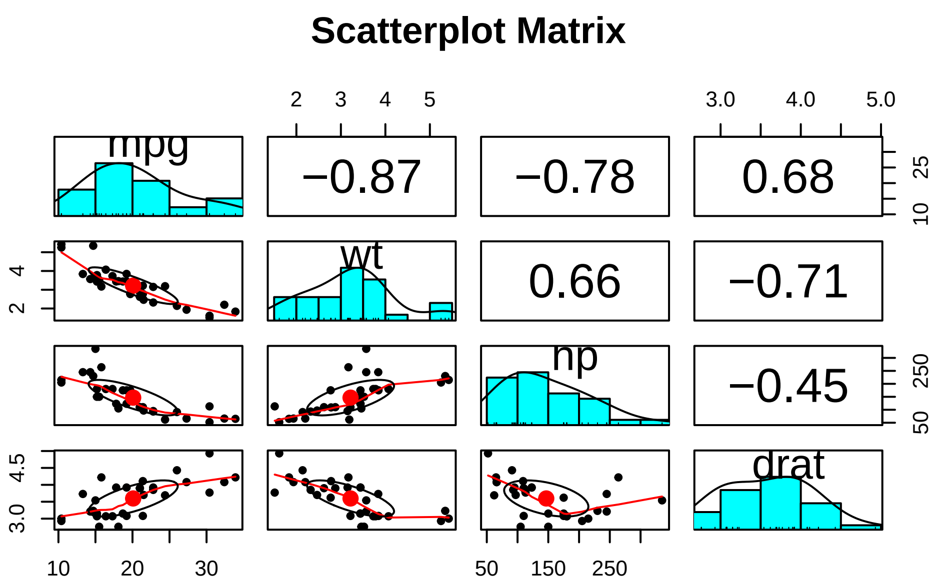


2. **Discussion**:

- The `car` package, short for Companion to Applied Regression, contains numerous functions and datasets that are helpful for regression analysis.
- For creating a scatterplot matrix„ we use the `scatterplotMatrix()` function.
- Here, the `data` argument specifies the data frame we are working with, `tb`.
- The formula `~ mpg + disp + drat + wt` indicates the variables to be included in the scatter plot matrix. Recall `mpg`, `disp`, `drat`, and `wt` represent miles per gallon, displacement, rear axle ratio, and weight, respectively.
- The `col` argument is used to set the color of the points in the scatter plots. Here, all points are colored red. [4]

14.3.3 Scatter Plot Matrix using `pairs.panels()` in package `psych`

```
# Load the 'psych' library, which contains the 'pairs.panels()' function
library(psych)

# Create a scatterplot matrix using the 'pairs.panels()' function
# Select the columns "mpg", "wt", "hp", and "drat"
# Set the main title of the scatterplot matrix to "Scatterplot Matrix"
pairs.panels(tb[, c("mpg", "wt", "hp", "drat")],
             main = "Scatterplot Matrix")
```



Discussion:

- The **psych** package contains a variety of functions useful for psychological, psychometric, and personality research. [4]
- We utilize the `pairs.panels()` function from the `psych` package to create a scatterplot matrix. This function is very helpful as it produces:
  - **Histograms**: The diagonal often contains histograms or density plots of the individual variables. This provides insights about the distribution of each variable.
  - **Scatterplots**: The lower triangle (below the diagonal) of the plot matrix contains scatterplots of each variable against the other. This helps to visualize the bivariate relationships between variables.

– **Correlation Coefficients**: The upper triangle (above the diagonal) often contains correlation coefficients or other statistics that describe the relationship between the two variables. The size (and sometimes color) of these numbers typically changes based on the strength of the correlation.

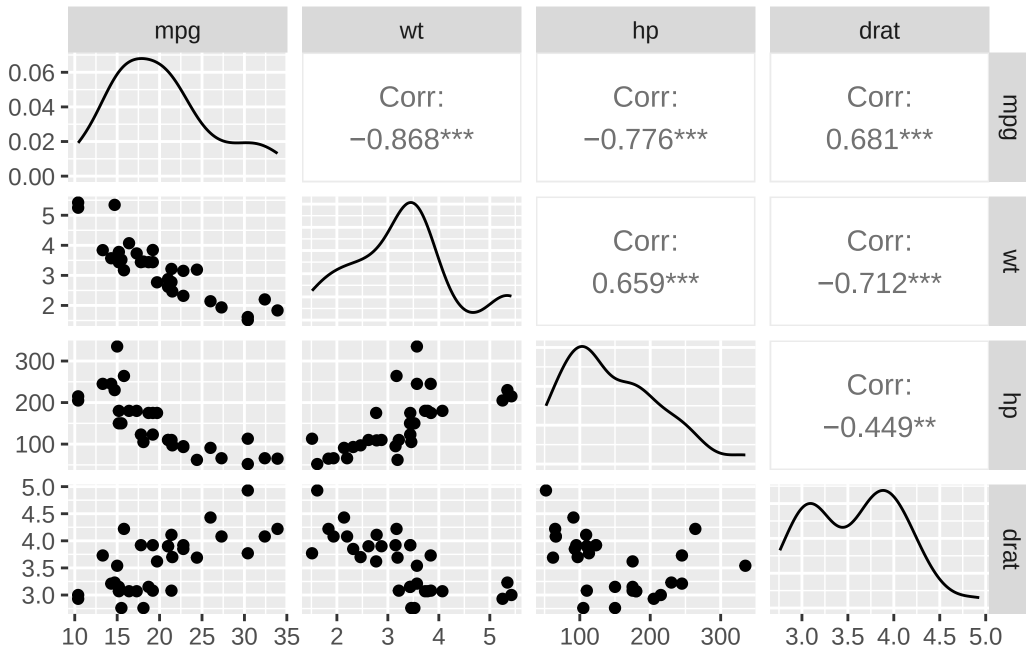
– **Red Dot**: In the scatterplots of the lower triangle, the red dot signifies the bivariate mean of the two variables being plotted. In other words, it represents the average value on the x-axis and the average value on the y-axis. The red dot helps you to quickly gauge where the center of the data is in relation to the scatter of the rest of the points. If the red dot (bivariate mean) is centrally located, it may suggest that the variables are symmetrically distributed around their means. If it's located towards one of the corners, it could suggest a skew in the data or a strong correlation between the variables.

– `tb[,c("mpg","wt","hp","drat")]` in the code specifies the subset of the `tb` data frame that we want to include in the scatterplot matrix. In this case, it refers to the columns `mpg` (miles per gallon), `wt` (weight), `hp` (horsepower), and `drat` (rear axle ratio).

– The `main` parameter sets the main title for the plot, which in this instance is "Scatterplot Matrix". [5]

14.3.4 Scatterplot Matrix Using `ggpairs()` in package `GGally`

```
# Load the 'GGally' package, which contains the 'ggpairs()' function
library(GGally)

# Create a scatterplot matrix using the 'ggpairs()' function
# Select the columns "mpg", "wt", "hp", and "drat"
ggpairs(tb[, c("mpg", "wt", "hp", "drat")])
```



Discussion:

- The GGally package is an extension of the renowned ggplot2 package in R. It offers enhanced functionalities to create specific types of plots, especially when dealing with relationships between multiple variables.
- The `ggpairs()` function is a standout tool from GGally. It is designed to generate scatterplot matrices (often known as pairs plots) in a visually appealing format, providing insights into the pairwise relationships between variables.
- In the given code, the data frame being operated on is denoted by `tb`.
- The selection `tb[,c("mpg","wt","hp","drat")]` specifically picks out the columns named "mpg", "wt", "hp", and "drat". This means that our scatterplot matrix will show the relationships among these four variables. As a reminder, assuming typical automotive datasets: "mpg" might signify miles per gallon, "wt" would be the weight of the vehicle, "hp" indicates horsepower, and "drat" could represent the drive axle ratio.
- The output plot, produced by `ggpairs()`, will display a 4x4 grid. The diagonal of this grid usually portrays histograms or density plots for the individual variables, while the off-diagonal segments demonstrate scatter plots between two variables, giving a comprehensive overview of how each variable relates to the others. [6]

14.4 Summary of Chapter 14 – Bivariate Continuous data (Part 3 of 4)

In this chapter, we explore the analysis of relationships between continuous variables. We begin by preparing a sample dataset and converting it to a structured format. As we move forward, we emphasize the importance of visual tools like scatter plots for understanding correlations and patterns between variables. These visual tools can reveal underlying trends, groupings, anomalies, and influential data points.

Using practical examples, we demonstrate how to create and customize these visualizations for clearer insights. Additionally, we introduce methods for enhancing scatter plots, including adding trend lines. The chapter also touches on the visualization of interactions between multiple variables, including the integration of categorical data, by means of color differentiation.

Transitioning from scatter plots, we explore the concept of a scatter plot matrix, or SPLOM. This tool is instrumental in showcasing pairwise relationships between a set of variables in a matrix format. Scatter plot matrices are incredibly helpful when navigating multivariate datasets, allowing quick visual recognition of potential correlations and relationships among variable pairs.

We then demonstrate how to create scatter plot matrices using R functions like `pairs()`, `scatterplotMatrix()`, and `pairs.panels()`. Each method offers a different visual experience and comes with its unique set of features. For example, using `pairs.panels()` from the psych package, the matrix not only contains scatter plots but also histograms on the diagonal, correlation coefficients, and other enlightening statistical visualizations.

Overall, this chapter equips us with the knowledge to visualize and analyze bivariate continuous data effectively, aiding in more profound data interpretation and insights.

14.5 References

Basic R Programming:

[1] Chambers, J. M. (2008). Software for Data Analysis: Programming with R (Vol. 2, No. 1). Springer, New York.

Crawley, M. J. (2012). The R Book. John Wiley & Sons.

Gardener, M. (2012). Beginning R: The Statistical Programming Language. John Wiley & Sons.

Grolemund, G. (2014). Hands-On Programming with R: Write Your Own Functions and Simulations. O'Reilly Media, Inc.

Kabacoff, R. (2022). R in Action: Data Analysis and Graphics with R and Tidyverse. Simon and Schuster.

Peng, R. D. (2016). R Programming for Data Science (pp. 86-181). Leanpub, Victoria, BC, Canada.

R Core Team. (2020). R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria. Retrieved from https://www.R-project.org/.

Tippmann, S. (2015). Programming Tools: Adventures with R. Nature, 517(7532), 109-110.

Wickham, H., Çetinkaya-Rundel, M., & Grolemund, G. (2023). R for Data Science. O'Reilly Media, Inc.

Statistics Using R:

[2] Braun, W. J., & Murdoch, D. J. (2021). A First Course in Statistical Programming with R. Cambridge University Press.

Cohen, Y., & Cohen, J. Y. (2008). Statistics and Data with R: An Applied Approach Through Examples. John Wiley & Sons.

Dalgaard, P. (2008). Introductory Statistics with R. Springer.

Davies, T. M. (2016). The Book of R: A First Course in Programming and Statistics. No Starch Press.

Everitt, B. S., & Hothorn, T. (2014). A Handbook of Statistical Analyses Using R. Chapman and Hall/CRC.

Field, A., Miles, J., & Field, Z. (2012). Discovering Statistics Using R. Sage Publications.

Fox, J., & Weisberg, S. (2018). An R Companion to Applied Regression. Sage Publications.

Hyndman, R. J., & Fan, Y. (1996). Sample Quantiles in Statistical Packages. The American Statistician, 50(4), 361-365.

Matloff, N. (2011). The Art of R Programming: A Tour of Statistical Software Design. No Starch Press.

R Core Team. (2020). R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria. Retrieved from https://www.R-project.org/.

Schumacker, R. E. (2014). Learning Statistics Using R. Sage Publications.

Schumacker, R., & Tomek, S. (2013). Understanding Statistics Using R. Springer Science & Business Media.

Regression:

[3] Fox, J., & Weisberg, S. (2011). An R Companion to Applied Regression (2nd ed.). Sage, Thousand Oaks, CA.

car:

[4] Fox, J., Weisberg, S., Adler, D., Bates, D., Baud-Bovy, G., Ellison, S., Heiberger, R., et al. (2012). Package 'car'. R Foundation for Statistical Computing, Vienna.

psych:

[5] Revelle, W. (2020). psych: Procedures for Psychological, Psychometric, and Personality Research. Northwestern University, Evanston, Illinois. R Package Version 2.0.12. Retrieved from https://CRAN.R-project.org/package=psych.

GGally:

[6] Schloerke, B., Crowley, J., & Cook, D. (2018). Package 'GGally': Extension to 'ggplot2'.

15 Bivariate Continuous data (Part 4 of 4)

Chapter 15.

15.1 Exploring bivariate Continuous x Continuous data, using `ggplot2`

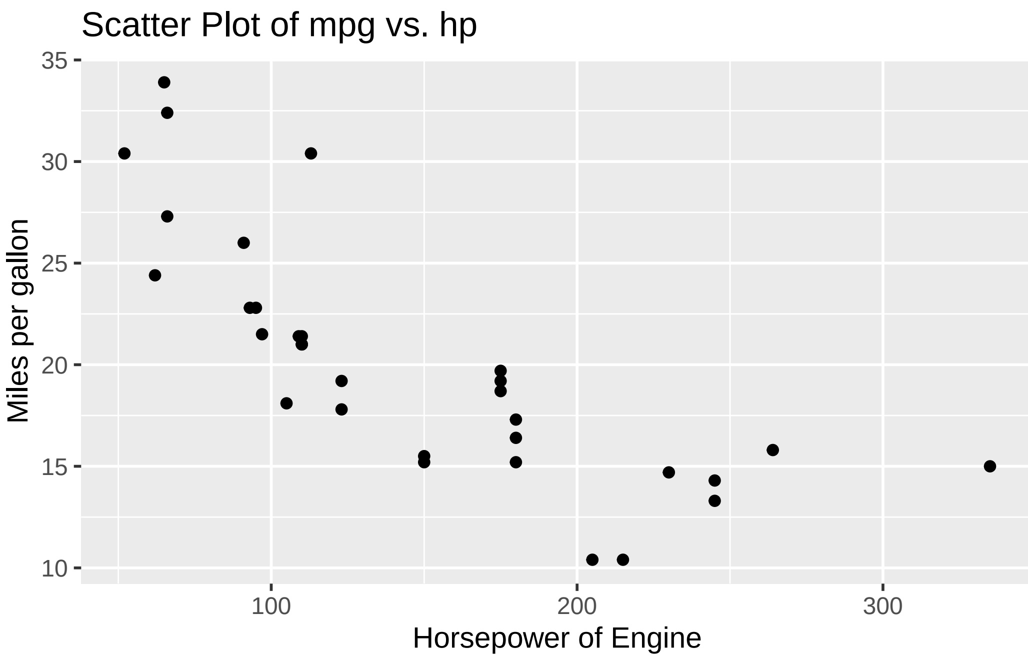
This chapter covers exploring bivariate continuous data using `ggplot2` and `ggpubr` in R. It demonstrates creating scatter plots to analyze relationships between continuous variables, utilizing ggplot2's layering approach for plot construction. The chapter includes examples of customizing plots with labels, titles, point shapes, color gradients, and regression lines using `ggplot2` and `ggpubr` functions. It also explores visualizations involving categorical variables, demonstrating faceted plots and bubble charts with varying point sizes. The chapter emphasizes enhancing scatter plots with annotations, color coding based on categories, and various plotting techniques for insightful data presentation.

Data: Suppose we run the following code to prepare the `mtcars` data for subsequent analysis and save it in a tibble called `tb`. [1]

```
# Load the required libraries, suppressing annoying startup messages
library(dplyr, quietly = TRUE, warn.conflicts = FALSE)
library(tibble, quietly = TRUE, warn.conflicts = FALSE)
library(ggplot2, quietly = TRUE, warn.conflicts = FALSE)
library(ggpubr, quietly = TRUE, warn.conflicts = FALSE)

# Read the mtcars dataset into a tibble called tb
data(mtcars)
tb <- as_tibble(mtcars)

# Convert relevant columns into factor variables
tb$cyl <- as.factor(tb$cyl) # cyl = {4,6,8}, number of cylinders
tb$am <- as.factor(tb$am) # am = {0,1}, 0:automatic, 1: manual transmission
tb$vs <- as.factor(tb$vs) # vs = {0,1}, v-shaped engine, 0:no, 1:yes
tb$gear <- as.factor(tb$gear) # gear = {3,4,5}, number of gears
```

15.1.1 Scatterplot using ggplot2

```
# Create a scatter plot using ggplot2 library  [2]
ggplot(data = tb,           # Specify the data frame 'tb' as the data source
       aes(x = hp,
           y = mpg)) +  # x-axis as 'hp' and y-axis as 'mpg'
  geom_point() +              # Add points to the plot
  xlab("Horsepower of Engine") +  # Label the x-axis
  ylab("Miles per gallon") +     # Label the y-axis
  ggtitle("Scatter Plot of mpg vs. hp")  # Set the title of the plot
```

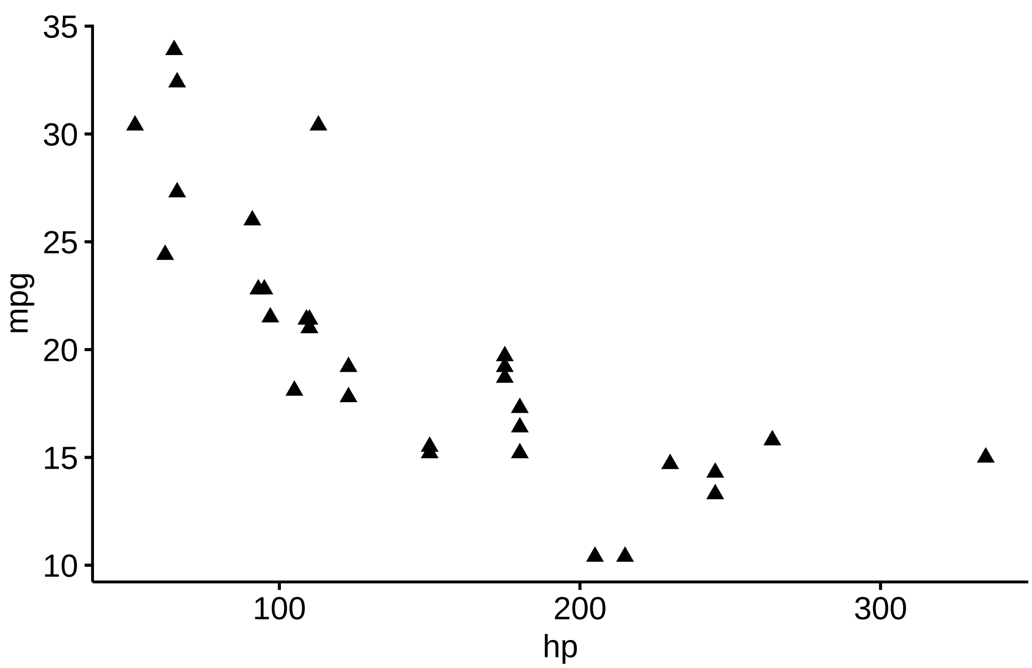


Discussion:

- The ggplot2 package uses a layering approach, enabling users to build plots incrementally, piece by piece, using a combination of data, aesthetics, and geometric objects.
- The function `ggplot()` initializes the plotting system. It requires a dataset to operate on and an aesthetic mapping to determine how data variables will be plotted. Here, the dataset is represented by `tb`.
- Inside the `aes()` function, which stands for aesthetics, the code specifies that the variable `hp` from the `tb` data frame will be plotted on the x-axis and the variable `mpg` will be plotted on the y-axis. Hence, the resulting plot will display a relationship between horsepower (`hp`) and miles per gallon (`mpg`).

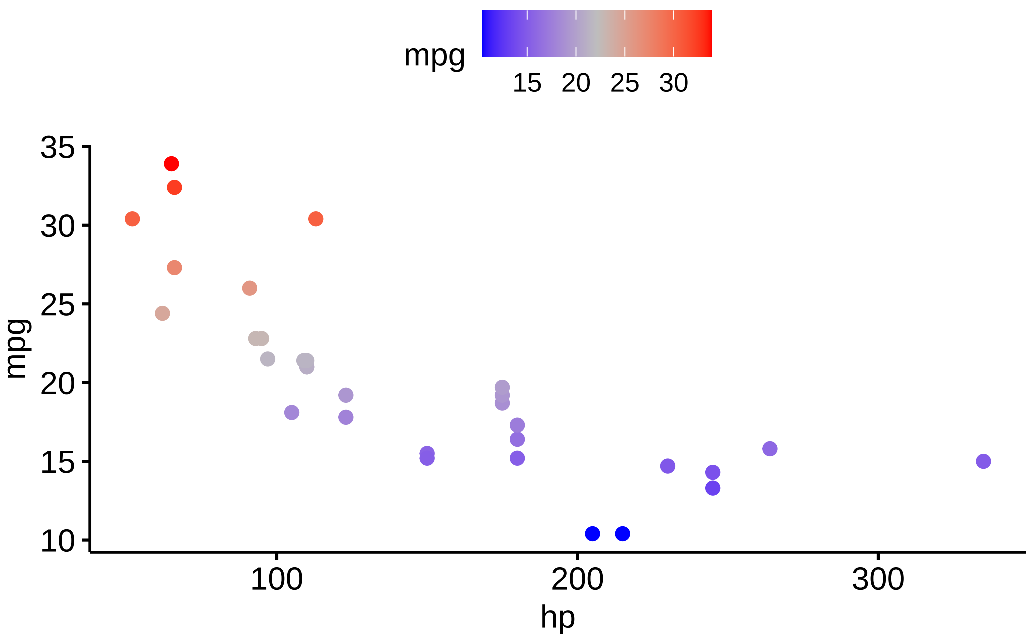
- The `geom_point()` function is an added layer, instructing `ggplot2` to render the relationship between `hp` and `mpg` as a scatter plot, with individual data points being represented as points.
- The functions `xlab()` and `ylab()` are used to set custom labels for the x and y axes, respectively. In this code, the x-axis is labeled as "Horsepower of Engine" and the y-axis is labeled as "Miles per gallon".
- Finally, the `ggtitle()` function is used to assign a title to the entire plot. In this instance, the title is set as "Scatter Plot of mpg vs. hp", clearly indicating the purpose and content of the visualization. [2]

```
# Load the 'ggpubr' library, which contains 'ggscatter()'
library(ggpubr)

# Create a scatter plot using ggscatter()
ggscatter(data = tb,         # Specify the data frame 'tb' as the data source
          x = "hp",          # Define the x-axis variable as "hp"
          y = "mpg",         # Define the y-axis variable as "mpg"
          shape = 17)        # Set the point shape to 17 (a filled circle)
```

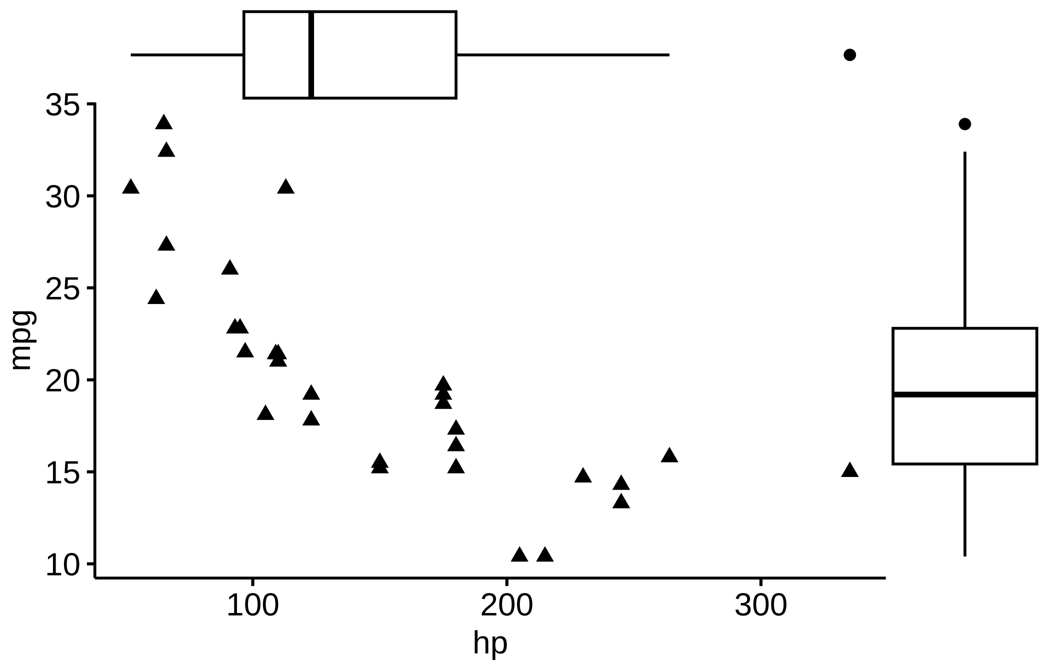



```
# Create a scatter plot using the 'ggscatter()'
# Set the x-axis variable to "hp" and the y-axis variable to "mpg"
# Additionally, specify the color aesthetic to "mpg" (color points based on "mpg")
```

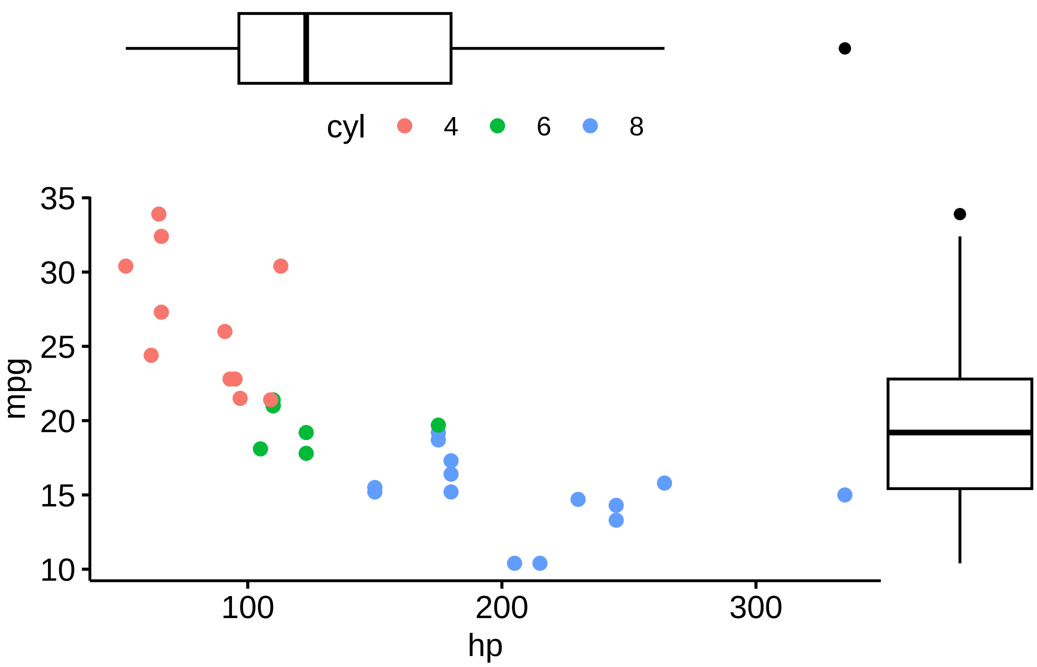
```
ggscatter(data = tb,
          x = "hp", y = "mpg",
          color = "mpg") +
  gradient_color(c("blue", "gray", "red")) # Add a gradient color scheme [3]
```




```
# Load the 'ggExtra' library, which contains the 'ggMarginal()' function
library("ggExtra")

# Create a scatter plot using the 'ggscatter()' function from 'ggpubr'
p <- ggscatter(data = tb,
               x = "hp", y = "mpg",
               shape = 17)

# Change the marginal plot type to a boxplot using 'ggMarginal()'
ggMarginal(p, type = "boxplot")
```

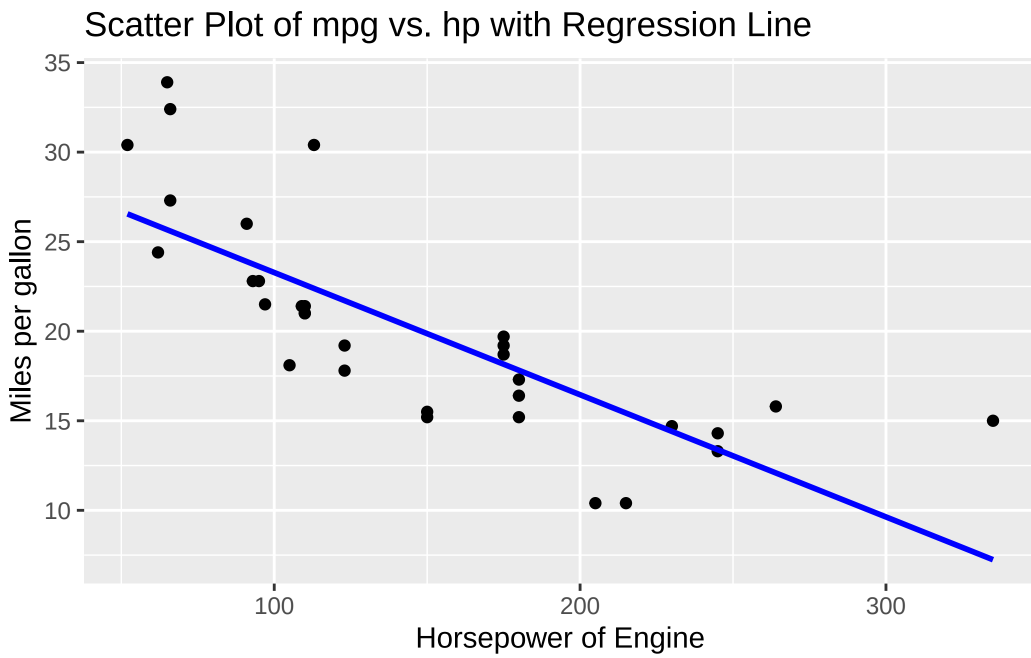



```
# Load the 'ggExtra' library, which contains the 'ggMarginal()' function
library("ggExtra")

# Create a scatter plot using 'ggscatter()'
# Set x-axis as "hp", y-axis as "mpg", and color points as "cyl"
p <- ggscatter(data = tb,
               x = "hp", y = "mpg",
               color = "cyl")

# Change the marginal plot type to a boxplot using 'ggMarginal()'
ggMarginal(p,
           type = "boxplot")
```

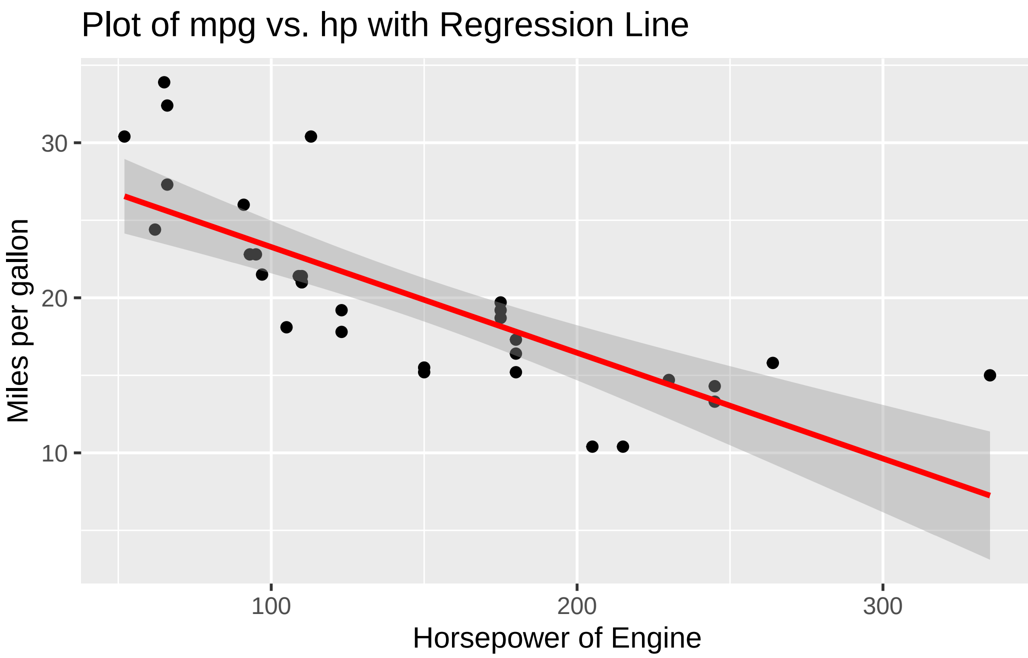


15.1.2 Scatterplot with Regression line using `ggplot2`

```
# Create a scatter plot using the 'ggplot()' function [2],[5]
ggplot(data = tb, aes(x = hp, y = mpg)) +
  geom_point() + # Add points to the plot
  geom_smooth(method = "lm",      # Add a linear regression line
              se = FALSE,         # Disable confidence intervals
              color = "blue") +   # set line color to blue
  xlab("Horsepower of Engine") + # Label the x-axis
  ylab("Miles per gallon") +     # Label the y-axis
  ggtitle("Scatter Plot of mpg vs. hp with Regression Line")
```




```r
ggplot(tb, aes(x = hp, y = mpg)) +
  geom_point() +                      # Add points to the plot  [2], [5]
  geom_smooth(method = "lm",          # Add a linear regression line
              se = TRUE,              # Enable confidence intervals
              color = "red") +        # Set line color to red
  xlab("Horsepower of Engine") +      # Label the x-axis
  ylab("Miles per gallon") +          # Label the y-axis
  ggtitle("Plot of mpg vs. hp with Regression Line")
```

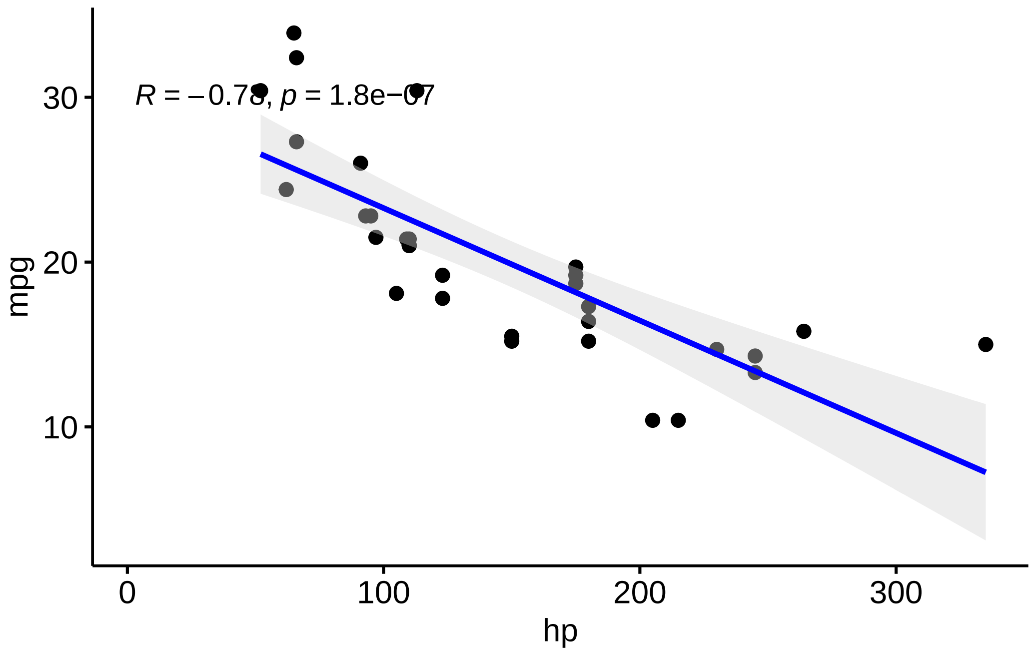


Discussion:

- `geom_smooth(method = "lm", se = FALSE, color = "blue")`: This function adds a smoothed conditional mean.
  - The `method = "lm"` argument indicates that a linear model (i.e., a regression line) should be used for smoothing. This line will depict the overall trend in the data.
  - If `se = FALSE` then the standard error bands (which show the uncertainty around the regression line) aren't plotted. This determines whether or not the standard error bands (or confidence interval bands) are displayed around the smoothing line. In the case of linear regression (method = "lm"), these bands represent the 95% confidence interval around the predicted values. This means that if you were to repeatedly sample from the population and fit a regression model each time, you'd expect about 95% of the confidence intervals to contain the true regression line. [3],[5]

```
# Create a scatter plot using ggscatter()
# Set the x-axis variable to "hp" and the y-axis variable to "mpg"
ggscatter(tb,
          x = "hp",
          y = "mpg",
          add = "reg.line",         # Add regression line
          conf.int = TRUE,          # Include confidence interval
          add.params = list(color = "blue",  # Customize color and fill
                            fill = "lightgray")) +
```

```r
# Add the Pearson correlation coefficient to the plot
# Specify the label coordinates for the correlation coefficient
stat_cor(method = "pearson",
         label.x = 3, label.y = 30)
```



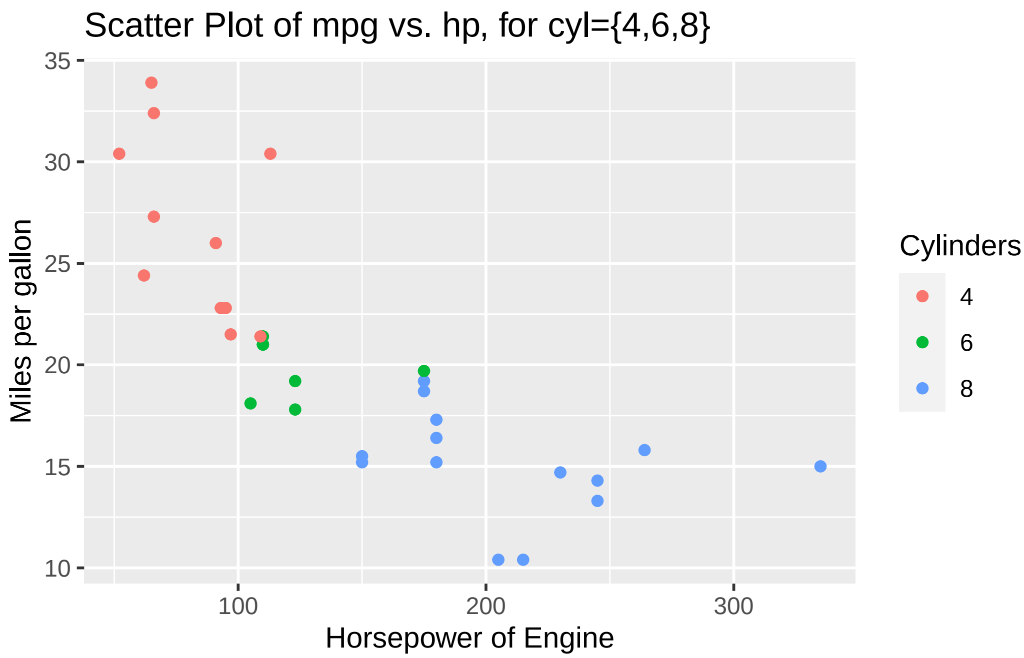
15.2 Scatterplots with Categorical Variables

15.2.1 Scatterplot colored by a Categorical variable, using `ggplot()`

This will create a scatterplot of miles per gallon (`mpg`) against horsepower (`hp`), with each point colored according to the number of cylinders (`cyl`) in the engine.

```r
# Create a scatter plot using the 'ggplot()'
# Set x-axis as "hp", y-axis as "mpg", and color points as "cyl"
ggplot(tb, aes(x = hp,
               y = mpg,
               color = factor(cyl))) +
# Add points to the plot
geom_point() +

labs(x = "Horsepower of Engine",
```

```
    y = "Miles per gallon") + # Label x-axis and y-axis
scale_color_discrete(name = "Cylinders")  +  # Customize  legend title
ggtitle("Scatter Plot of mpg vs. hp, for cyl={4,6,8}")
```



Discussion:

- The `aes()` function, short for aesthetics, designates the variables and their roles in the plot. In this code:

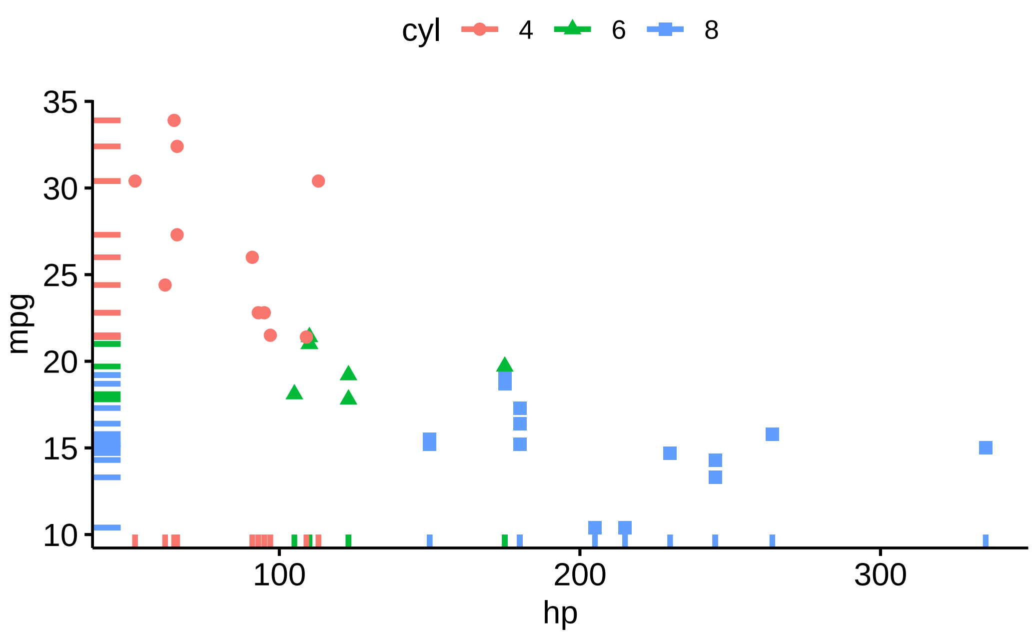
  - The `hp` variable is plotted on the x-axis.
  - The `mpg` variable is mapped to the y-axis.
  - The `color` attribute is set based on the `cyl` variable, which presumably indicates the number of cylinders in a car engine. The use of `factor(cyl)` ensures that the `cyl` variable is treated as a discrete factor rather than a continuous variable, which is essential for color differentiation.

- `geom_point()` introduces a scatter plot layer, meaning that the relationship between `hp` and `mpg` will be represented using individual points, with each point's color reflecting the number of cylinders as specified in the aesthetic mapping.

- The `labs()` function provides a convenient way to label the axes. Here, the x-axis receives the label "Horsepower" and the y-axis is labeled "Miles per gallon".

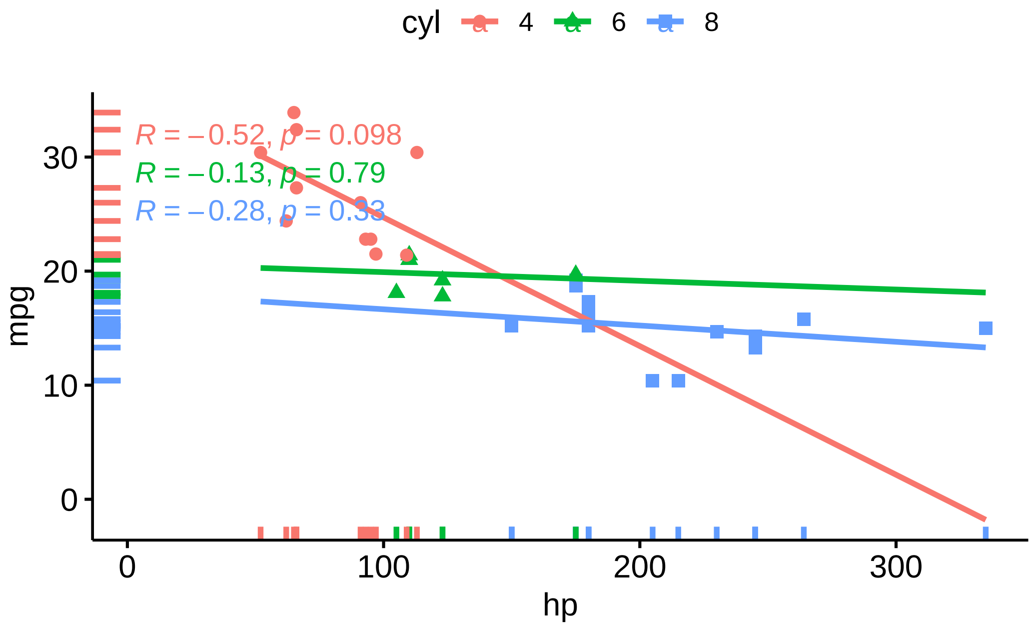
- The `scale_color_discrete()` function customizes the color scale for discrete variables. By specifying the `name` argument as "Cylinders", it ensures that the legend accompanying the color scale in the plot will be labeled as "Cylinders", making it clear to viewers that the colors of the points represent different cylinder counts. [3]

```
# Load the 'ggpubr' library, which contains 'ggscatter()'
library(ggpubr)

# Create a scatter plot using the 'ggscatter()' function
ggscatter(tb,
          x = "hp", y = "mpg",
          color = "cyl",  # Color points by "cyl"
          shape = "cyl",  # Change point shape by "cyl"
          rug = TRUE)    # Add marginal rugs
```

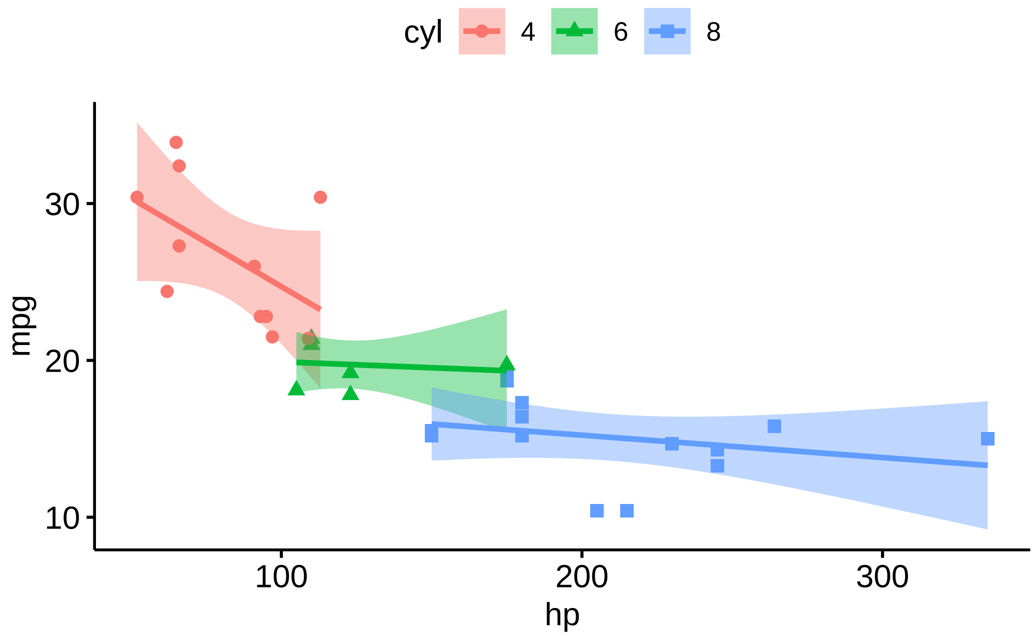



```
library(ggpubr)
# Create a scatter plot using the 'ggscatter()' function   [3]
ggscatter(tb,
          x = "hp", y = "mpg",
          add = "reg.line",     # Add regression line
          color = "cyl",        # Color by groups "cyl"
          shape = "cyl",        # Change point shape by "cyl"
          fullrange = TRUE,     # Extend the regression line
          rug = TRUE            # Add marginal rug (marginal density)
          ) +
# Add the Pearson correlation coefficient
```

```r
# Color the label by groups "cyl" and specify the label coordinates
stat_cor(aes(color = cyl),
         label.x = 3)
```



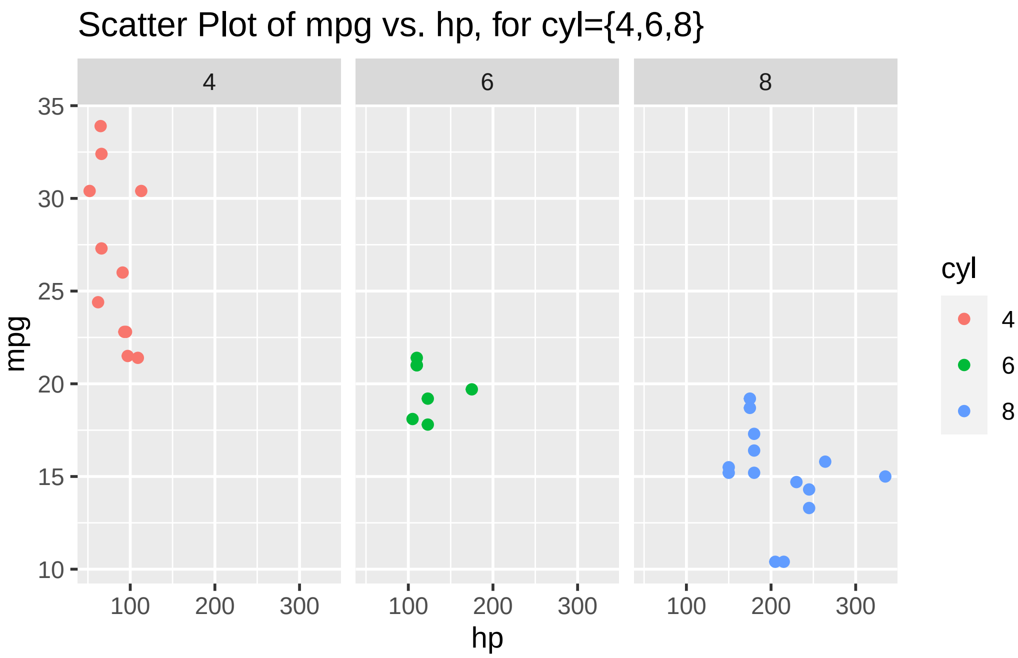

```r
# Create a scatter plot using the 'ggscatter()' function  [3], [5]
# Set the x-axis variable to "hp" and the y-axis variable to "mpg"
ggscatter(tb,
          x = "hp", y = "mpg",
          add = "reg.line",  # Add regression line
          conf.int = TRUE,   # Include confidence interval
          color = "cyl",     # Color by groups "cyl"
          shape = "cyl"      # Change point shape by groups "cyl"
          )
```



15.2.2 Scatterplot faceted by a Categorical variable, using `ggplot()`

This will create a scatterplot of miles per gallon (mpg) against weight, with each plot faceted by the number of cylinders in the engine (cyl).

```
# Create a scatter plot using the 'ggplot()'
# Set x-axis as "hp", y-axis as "mpg", and color as "cyl"
ggplot(tb,
       aes(x = hp,
           y = mpg,
           color = cyl)) +
# Add points to the plot
geom_point() +
# Facet the plot by the variable "cyl"
facet_grid(. ~ cyl) +
ggtitle("Scatter Plot of mpg vs. hp, for cyl={4,6,8}")
```



Discussion:

- The foundational layer is initialized with the `ggplot()` function. This function takes in a dataset, `tb`, and aesthetic mappings that determine how variables are displayed. In this piece of code:

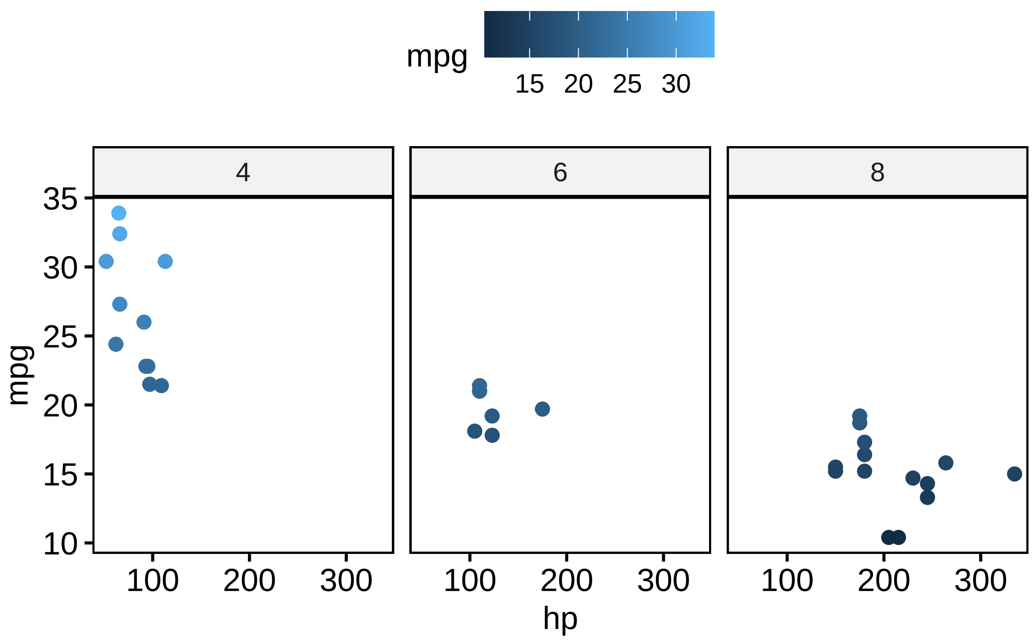
  - `hp` is chosen to be plotted on the x-axis.
  - `mpg` is selected for the y-axis.
  - The color of the points will be determined by the `cyl` variable.

- The addition of the `geom_point()` layer ensures that a scatter plot will represent the relationship between `hp` and `mpg`. Each point's color will correspond to the value of the `cyl` variable.

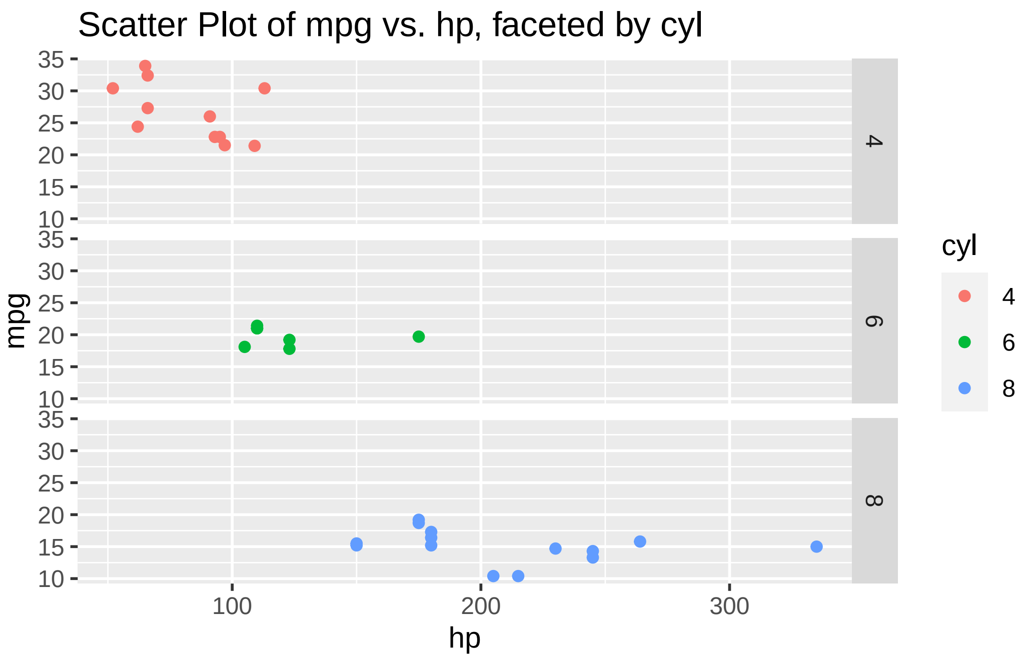
- The `facet_grid()` function introduces the concept of faceting. Faceting divides a plot into multiple panels based on the levels of one or more factors. In this case, the plot is faceted horizontally (`~ cyl`), meaning that separate panels are created for each unique value of `cyl`. The `.` before the `~` indicates that there's no faceting vertically.

- Finally, the `ggtitle()` function provides the entire plot with a title, which is "Scatter Plot of mpg vs. hp, for cyl={4,6,8}". This title clearly communicates the main theme of the plot and indicates that it showcases relationships for cars with 4, 6, or 8 cylinders.

```r
# Create a scatter plot using the 'ggscatter()' function
ggscatter(tb,
          x = "hp",
          y = "mpg",
          color = "mpg", # Color points by "mpg"
          facet.by = "cyl") # Facet the plot by "cyl"
```



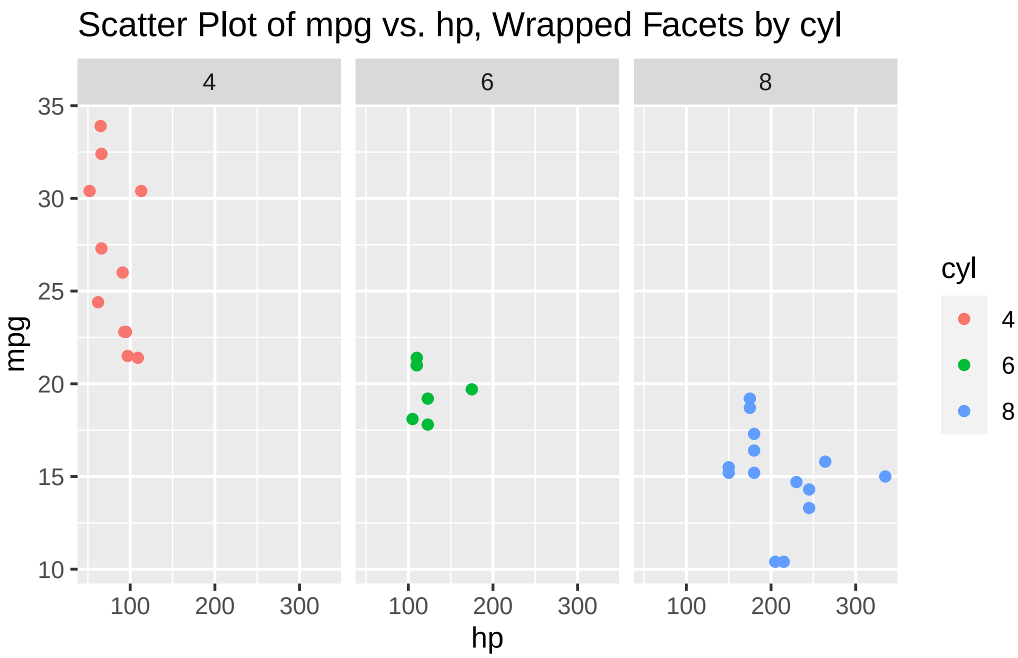

```r
# Create a scatter plot using the 'ggplot()' function
# Set x-axis as "hp", y-axis as "mpg", and color as "cyl"
ggplot(tb,
       aes(x = hp,
           y = mpg,
           color = cyl)) +
# Add points to the plot
geom_point() +
# Facet the plot vertically by "cyl"
facet_grid(cyl ~ .) +
ggtitle("Scatter Plot of mpg vs. hp, faceted by cyl")
```



Discussion:

- The primary difference between the two code snippets lies in how the faceting is implemented using the `facet_grid()` function.
- In the original code, `facet_grid(. ~ cyl)` is used, which means the scatter plots are faceted horizontally based on the unique values of the `cyl` variable; each unique cylinder count gets its own column.
- Conversely, in the updated code with `facet_grid(cyl ~ .)`, the scatter plots are faceted vertically based on the unique values of the `cyl` variable; each unique cylinder count gets its own row.

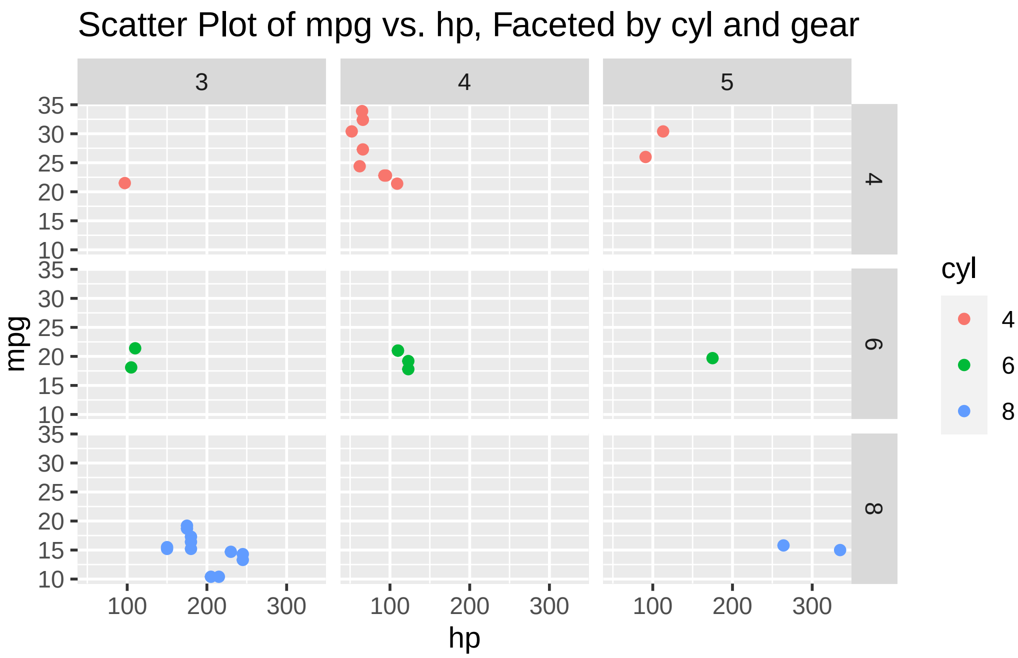
```
# Create a scatter plot using the 'ggplot()'
# Set x-axis as "hp", y-axis as "mpg", and color as "cyl"
ggplot(tb,
       aes(x = hp,
           y = mpg,
           color = cyl)) +
# Add points to the plot
geom_point() +
# Wrap facets by the variable "cyl" with 3 columns
facet_wrap(~ cyl,
           ncol = 3) +
ggtitle("Scatter Plot of mpg vs. hp, Wrapped Facets by cyl")
```



Discussion:

- This approach creates a wrapped grid of facets based on `cyl`.
- The `ncol = 3` argument specifies that up to three facets will be placed in a row before wrapping to the next row. You can adjust this as needed based on the number of levels in the faceting variable and the desired layout.

```
# Create a scatter plot using the 'ggplot()'
# Set x-axis as "hp", y-axis as "mpg", and color as "cyl"
ggplot(tb,
       aes(x = hp,
           y = mpg,
           color = cyl)) +
# Add points to the plot
geom_point() +
# Facet the plot by the variables "cyl" and "gear"
facet_grid(cyl ~ gear) +
ggtitle("Scatter Plot of mpg vs. hp, Faceted by cyl and gear")
```



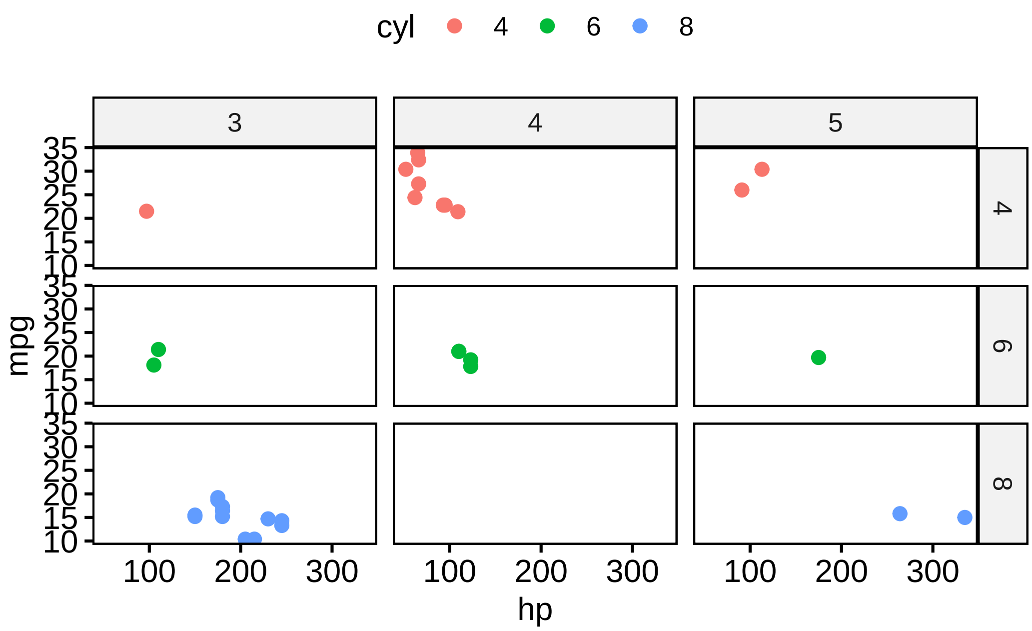
Discussion:

- In this code, within the `aes()` aesthetics function:
  - The variable `hp` is mapped to the x-axis.
  - The variable `mpg` is mapped to the y-axis.
  - The color of individual points is determined by the `cyl` variable, which probably represents the number of cylinders in an engine.

- The `geom_point()` function is introduced to represent the relationship between `hp` and `mpg` as a scatter plot. The colors of the individual points will correspond to the values of the `cyl` variable.

- The `facet_grid(cyl ~ gear)` function is the standout feature in this code. Here, the plots are faceted based on two categorical variables:
  - `cyl`, which is mapped to rows. Each unique value of `cyl` will generate a new row of plots.
  - `gear`, which is mapped to columns. Each unique value of `gear` will generate a new column of plots.
  - The resultant grid will represent combinations of `cyl` and `gear` values, with each cell in the grid showing the relationship between `hp` and `mpg` for a specific combination of `cyl` and `gear`. [4]

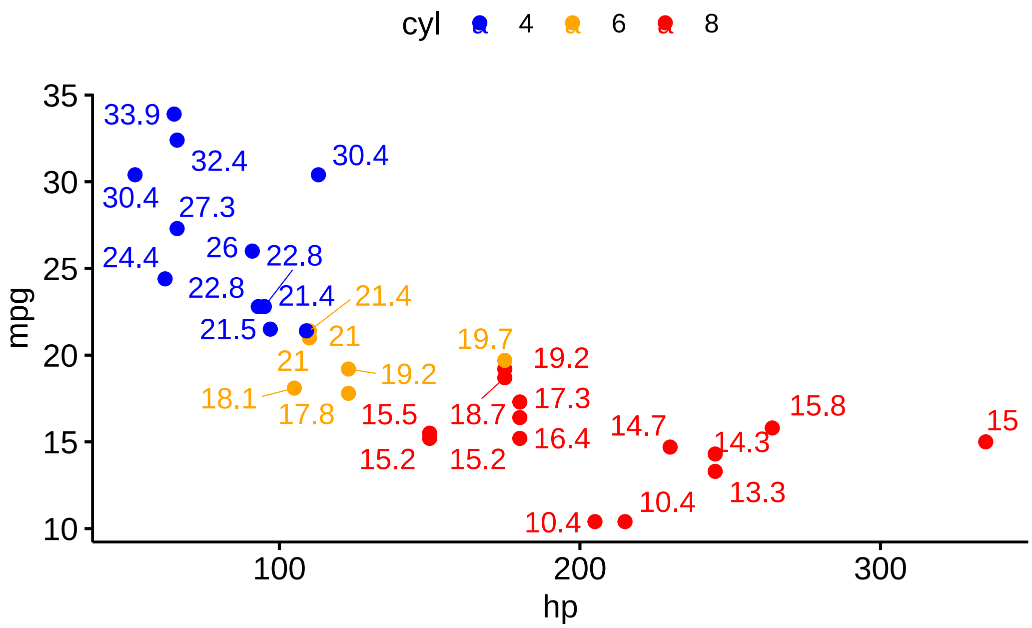
```r
# Create a scatter plot using the 'ggscatter()' function
# Set the x-axis variable to "hp" and the y-axis variable to "mpg"
# Color points by the categorical variables "cyl" and "gear"
# Facet the plot by the combinations of "cyl" and "gear"
ggscatter(tb,
          x = "hp", y = "mpg",
          color = "cyl",
          facet.by = c("cyl", "gear"))
```



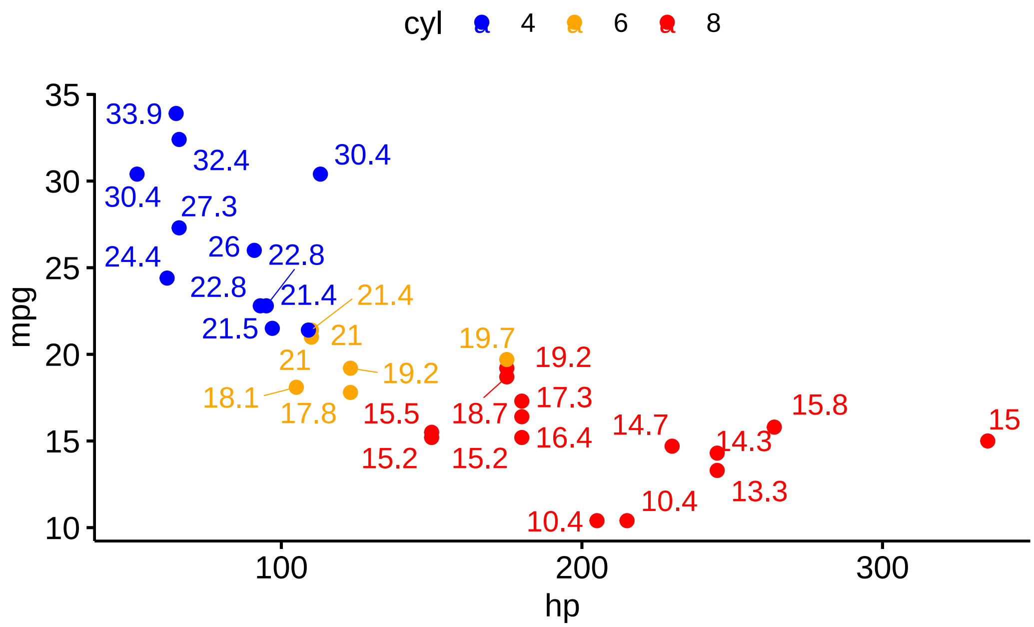
15.2.3 Scatterplot colored by a Categorical variable, with textual annotation, using `ggpubr()`

```r
# Create a scatter plot using the 'ggscatter()' function    [3]
# Set the x-axis variable to "hp" and the y-axis variable to "mpg"
# Color points by the categorical variable "cyl" using the specified palette
# Add textual annotations using the "mpg" variable and enable point repelling
ggscatter(tb,
          x = "hp",
          y = "mpg",
          color = "cyl",
          palette = c("blue", "orange", "red"),
```

```
        label = "mpg",
        repel = TRUE)
```

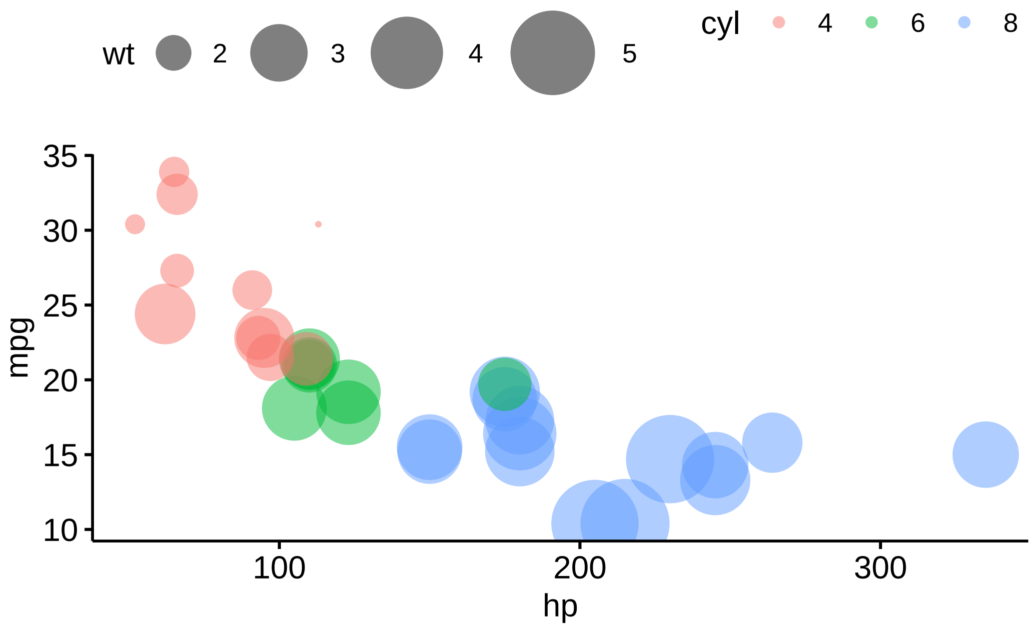



```
# Create a scatter plot using the 'ggscatter()' function    [3]
# Set the x-axis variable to "hp" and the y-axis variable to "mpg"
# Color points by the categorical variable "cyl" using the specified palette
# Add textual annotations using the "mpg" variable and enable point repelling
ggscatter(tb,
          x = "hp",
          y = "mpg",
          color = "cyl",
          palette = c("blue", "orange", "red"),
          label = "mpg",
          repel = TRUE) +

# Set limits to fit the entire plot inside a box
coord_cartesian(ylim = c(min(tb$mpg), max(tb$mpg)),
                xlim = c(min(tb$hp), max(tb$hp)))
```



15.3 Bubble Chart

```
# Create a scatter plot using the 'ggscatter()' function    [3]
# Set the x-axis variable to "hp" and the y-axis variable to "mpg"
# Color points by the categorical variable "cyl"
# Adjust point size based on the continuous variable "wt"
# and set transparency (alpha) to 0.5
ggscatter(tb,
          x = "hp", y = "mpg",
          color = "cyl",
          size = "wt",
          alpha = 0.5) +

# Adjust the range of point sizes for better visualization
scale_size(range = c(0.5, 15))
```



15.4 Summary of Chapter 15 – Bivariate Continuous data (Part 4 of 4)

This chapter provides a comprehensive guide to exploring bivariate continuous data using R's `ggplot2` and `ggpubr` packages. Initially, the data is formatted into a tibble, and key variables are transformed into factors for nuanced analysis. The chapter emphasizes scatterplot creation using `ggplot2`, illustrating the relationship. Techniques such as custom labeling, adding regression lines, and layering are demonstrated to enhance plot interpretability. `ggplot2`'s layering approach is highlighted for its effectiveness in building complex plots incrementally.

Further, the `ggpubr` package's `ggscatter()` function is introduced for advanced scatterplot customization. This includes altering point shapes, adding marginal rugs, regression lines, and correlation coefficients. The integration of categorical variables into scatter plots is extensively discussed. This involves coloring points by categories like the number of cylinders and employing various faceting techniques to compare data across categories. Advanced topics like point annotations, point repelling for clarity, and bubble charts where point sizes vary with a continuous variable are also covered. These techniques provide a multi-dimensional view of the data, showcasing the versatility of `ggplot2` and `ggpubr` in visual data analysis. Overall, the chapter equips readers with the skills to create and enhance scatter plots in R, offering insightful methods for detailed bivariate data analysis.

15.5 References

Basic R Programming

[1] Chambers, J. M. (2008). *Software for data analysis: Programming with R* (Vol. 2, No. 1). Springer.

Crawley, M. J. (2012). *The R book*. John Wiley & Sons.

Gardener, M. (2012). *Beginning R: The statistical programming language*. John Wiley & Sons.

Grolemund, G. (2014). *Hands-on programming with R: Write your own functions and simulations*. O'Reilly Media, Inc.

Kabacoff, R. (2022). *R in action: Data analysis and graphics with R and Tidyverse*. Simon and Schuster.

Peng, R. D. (2016). *R programming for data science*. Leanpub.

R Core Team (2020). *R: A language and environment for statistical computing*. R Foundation for Statistical Computing. Retrieved from https://www.R-project.org/

Tippmann, S. (2015). Programming tools: Adventures with R. *Nature*, 517(7532), 109-110.

Wickham, H., Çetinkaya-Rundel, M., & Grolemund, G. (2023). *R for data science*. O'Reilly Media, Inc.

ggplot2

[2] Wickham, H. (2016). *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York. Retrieved from https://ggplot2.tidyverse.org

Wickham, H., & Grolemund, G. (2016). *R for Data Science: Import, Tidy, Transform, Visualize, and Model Data*. O'Reilly Media.

Wickham, H. (2020). *ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics* (Version 3.3.2) [Computer software]. Retrieved from https://CRAN.R-project.org/package=ggplot2

Wickham, H., et al. (2020). *dplyr: A Grammar of Data Manipulation* (Version 1.0.2) [Computer software]. Retrieved from https://CRAN.R-project.org/package=dplyr

Wilkinson, L. (2005). *The Grammar of Graphics* (2nd ed.). Springer-Verlag.

Wickham, H., et al. (2020). *tibble: Simple Data Frames* (Version 3.0.3) [Computer software]. Retrieved from https://CRAN.R-project.org/package=tibble

ggpubr

[3] Kassambara, A. (2023). *ggpubr: 'ggplot2' Based Publication Ready Plots.* R package version 0.6.0. Retrieved from https://rpkgs.datanovia.com/ggpubr/

Statistics using R

[4] Braun, W. J., & Murdoch, D. J. (2021). *A first course in statistical programming with R.* Cambridge University Press.

Cohen, Y., & Cohen, J. Y. (2008). *Statistics and Data with R: An applied approach through examples.* John Wiley & Sons.

Dalgaard, P. (2008). *Introductory statistics with R.* Springer.

Davies, T. M. (2016). *The book of R: A first course in programming and statistics.* No Starch Press.

Everitt, B. S., & Hothorn, T. (2014). *A Handbook of Statistical Analyses Using R.* Chapman and Hall/CRC.

Field, A., Miles, J., & Field, Z. (2012). *Discovering statistics using R.* Sage Publications.

Hyndman, R. J., & Fan, Y. (1996). Sample quantiles in statistical packages. *The American Statistician*, 50(4), 361-365.

Matloff, N. (2011). *The art of R programming: A tour of statistical software design.* No Starch Press.

R Core Team (2020). *R: A language and environment for statistical computing.* R Foundation for Statistical Computing. Retrieved from https://www.R-project.org/

Schumacker, R. E. (2014). *Learning statistics using R.* Sage Publications.

Schumacker, R., & Tomek, S. (2013). *Understanding statistics using R.* Springer Science & Business Media.

Regression

[5] Fox, J., & Weisberg, S. (2011). *An R Companion to Applied Regression* (2nd ed.). Sage.

16 Three Dimensional (3D) Data

Chapter 16.

16.1 Overview of R packages for 3D Plots

This chapter demonstrates how to create 3D plots in R. This chapter demonstrates creating 3D plots in R, focusing on ggplot2 and scatterplot3d packages. It begins by discussing the ggplot2 package's limitations in producing true 3D plots, though it can simulate 3D effects using size, color, and transparency. The chapter then introduces the scatterplot3d package for creating genuine 3D scatter plots. This package is highlighted for its simplicity and effectiveness in visualizing data in three dimensions.

`ggplot2`

The ggplot2 package in R is primarily designed for creating 2D plots. However, we can simulate 3D effects, even though it won't provide true 3D plots. [1]

`scatterplot3d`

The scatterplot3d package in R can be used to create 3D scatter plots. It is quite simple to use and can be a good choice for quick visualizations. [2]

`rgl`

The rgl package in R is a powerful tool for creating interactive 3D plots. It allows real-time interaction with the plots, where users can zoom, rotate, and pan the visualization to explore the data in three dimensions. This package provides various functionalities to create a variety of 3D plots, including scatter plots, surface plots, line plots, bar plots. [3]

In this chapter, we focus our attention on ggplot2 and scatterplot3d packages.

```r
# Load the required libraries for 3D plots, suppressing startup messages
library(scatterplot3d, quietly = TRUE, warn.conflicts = FALSE)
```

Data: Suppose we run the following code to prepare the `mtcars` data for subsequent analysis and save it in a tibble called `tb`.

```r
# Load the required libraries, suppressing annoying startup messages
library(dplyr, quietly = TRUE, warn.conflicts = FALSE)
library(tibble, quietly = TRUE, warn.conflicts = FALSE)
library(ggplot2, quietly = TRUE, warn.conflicts = FALSE) # For data visualization
library(ggpubr, quietly = TRUE, warn.conflicts = FALSE) # For data visualization

# Read the mtcars dataset into a tibble called tb
data(mtcars)
tb <- as_tibble(mtcars)

# Convert relevant columns into factor variables
tb$cyl <- as.factor(tb$cyl) # cyl = {4,6,8}, number of cylinders
tb$am <- as.factor(tb$am) # am = {0,1}, 0:automatic, 1: manual transmission
tb$vs <- as.factor(tb$vs) # vs = {0,1}, v-shaped engine, 0:no, 1:yes
tb$gear <- as.factor(tb$gear) # gear = {3,4,5}, number of gears
```

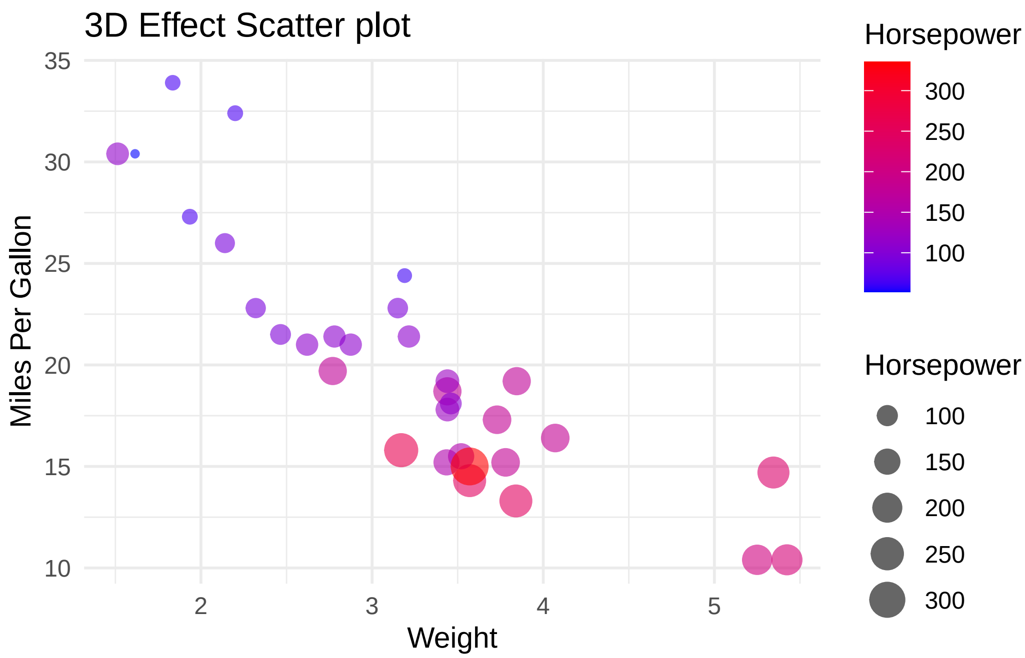
16.2 3D Plots using `ggplot2`

The ggplot2 package in R cannot create true 3D plots. Here is how to simulate a 3D plot using ggplot2. [1]

16.2.1 Simulating 3D using `ggplot2`

```r
# Load necessary library
library(ggplot2)

# Create a scatter plot with the perception of depth using size and color
ggplot(tb,
       aes(x = wt,
           y = mpg,
           color = hp,
           size = hp)) +  # Define the data and aesthetics for the plot
  geom_point(alpha = 0.6) +  # Add points to the plot with transparency
  scale_color_gradient(low = "blue", high = "red") +  # Define color gradient
  theme_minimal() +  # Set the plot theme to minimal
  labs(title = "3D Effect Scatter plot",
       x = "Weight",
       y = "Miles Per Gallon",
```

```
       color = "Horsepower",
       size = "Horsepower")  # Set plot labels and title
```



Discussion:

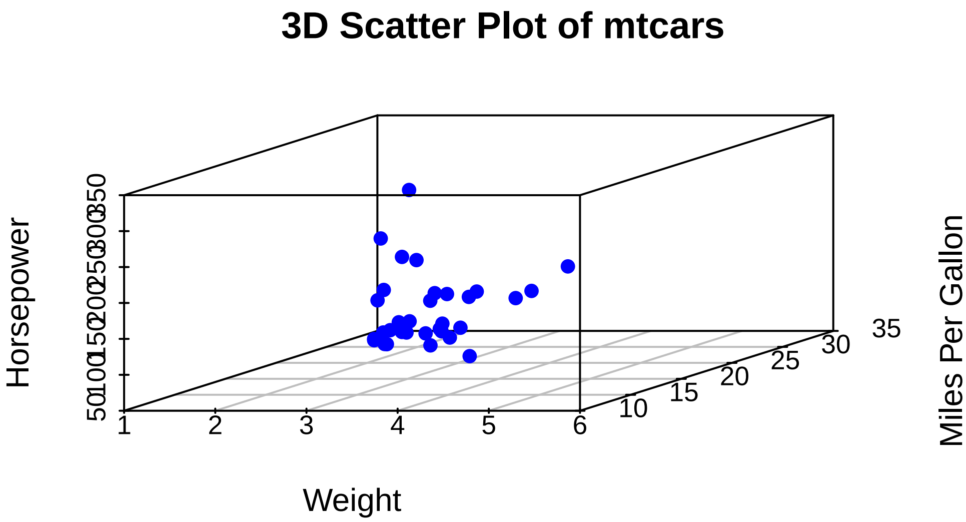
- The code provided simulates a 3D plot by exploiting visual cues and perceptual aspects of human vision. It uses the following strategies to simulate depth and give the perception of a three-dimensional plot on a two-dimensional surface:
- **Size Encoding**: The `size = hp` aesthetic in the `aes()` function maps the `hp` (horsepower) variable to the size of the points in the scatter plot. Points representing cars with higher horsepower appear larger, while those with lower horsepower appear smaller. This varying point size provides a depth cue, simulating the z-axis on a 2D surface.
- **Color Encoding**: The `color = hp` aesthetic assigns different colors to the points based on their horsepower values. The `scale_color_gradient(low = "blue", high = "red")` function further specifies that lower horsepower should be represented with blue, transitioning to red for higher horsepower. The gradient color scale serves as another depth cue, indicating that points of different colors are at different 'depths'.
- **Alpha Transparency**: The `alpha = 0.6` parameter in the `geom_point()` function adjusts the transparency of the points. This transparency allows for better visibility of overlapping points, giving a sense of depth where points are clustered.

- **Axis Mapping**: The `x = wt` and `y = mpg` within the `aes()` function map the weight of the cars to the x-axis and the miles per gallon to the y-axis. These two axes represent the spatial dimensions on the plotting surface.
- This plot has similarities to the classic bubble plot. The application of a color gradient gives the illusion of 3D.
- By combining these elements, the plot employs size, color gradient, and transparency to introduce visual cues of depth, effectively simulating a 3D effect on a 2D plotting surface. This enables the viewer to infer relationships among `wt`, `mpg`, and `hp`, even though the plot itself is inherently two-dimensional. [1]

16.3 3D Plots using `scatterplot3d`

The *`scatterplot3d`** package can be used on the mtcars data to demonstrate visualization in 3 dimensions.

```
# Load the scatterplot3d library for 3D plotting
library(scatterplot3d)

# Create a 3D scatter plot using the tb tibble
sp3 <- scatterplot3d(x = tb$wt,
                     y = tb$mpg,
                     z = tb$hp,
                     xlab = "Weight",
                     ylab = "Miles Per Gallon",
                     zlab = "Horsepower",
                     pch = 16, color = "blue",
                     main = "3D Scatter Plot of mtcars")
```



Discussion:

- `x = tb$wt`, `y = tb$mpg`, and `z = tb$hp` are used to map the `wt`, `mpg`, and `hp` columns from the `tb` tibble to the x, y, and z axes of the 3D scatter plot respectively.
- The `xlab`, `ylab`, and `zlab` parameters are used to label the x, y, and z axes.
- `pch = 16` specifies the type of point to be used in the plot.
- `color = "blue"` sets the color of the points to blue.
- `main` is used to set the title of the plot.
- Overall, this code visualizes a 3D scatter plot demonstrating the relationships between `wt`, `mpg`, and `hp` in the `mtcars` dataset. [2]

16.3.1 Variations of 3D Plots using `scatterplot3d`

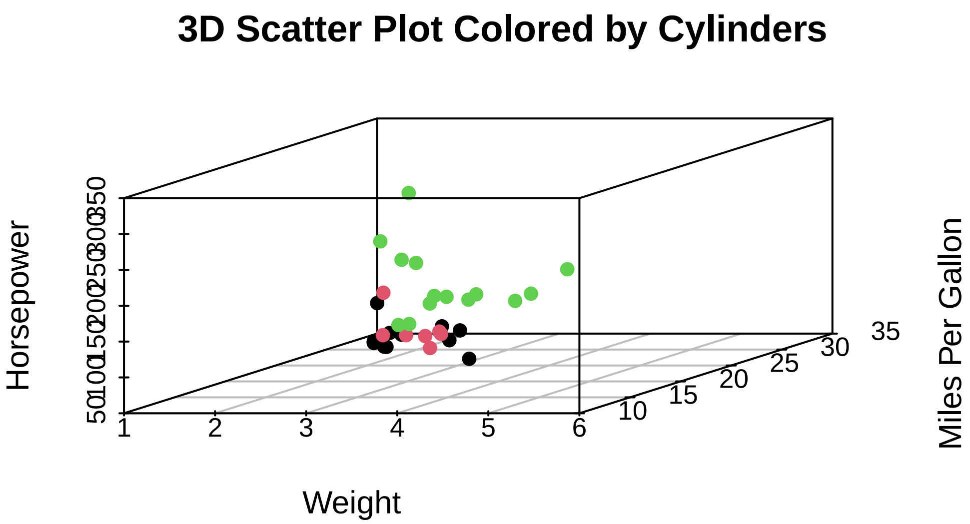
Here are some variations and extensions of the original scatterplot3d code to demonstrate the versatility of 3D visualizations:

Color by a Variable:

We can color points by a variable, for example, by the number of cylinders (cyl), which can reveal additional patterns in the data.

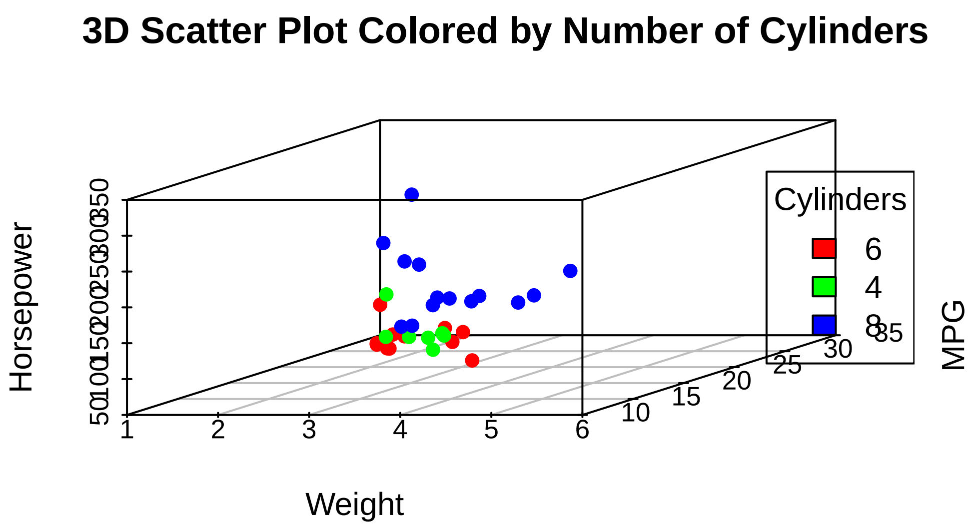
```
# Load the scatterplot3d library for 3D plotting
library(scatterplot3d)

# Create a 3D scatter plot, with points colored by cylinders
sp32 <- scatterplot3d(x = tb$wt,
                      y = tb$mpg,
                      z = tb$hp,
```

```
                          color = as.numeric(tb$cyl),  # Color by cyl
                          pch = 16,  # Use a specific point symbol
                          xlab = "Weight",
                          ylab = "Miles Per Gallon",
                          zlab = "Horsepower",
                          main = "3D Scatter Plot Colored by Cylinders")
```



- We can add a legend for the color in the `scatterplot3d` plot by using a combination of the `legend` function and the `colors` argument. [2]

```
library(scatterplot3d)
# Define a color palette
color_palette <- rainbow(length(unique(tb$cyl)))

# Create a scatterplot3d and specify colors by the number of cylinders
sp33 <- scatterplot3d(x = tb$wt, y = tb$mpg, z = tb$hp,
                      color = color_palette[as.numeric(tb$cyl)],
                      pch = 16,
                      xlab = "Weight", ylab = "MPG", zlab = "Horsepower",
                      main = "3D Scatter Plot Colored by Number of Cylinders")
# Add a color legend
legend("right",
       legend = unique(as.character(tb$cyl)),
       fill = color_palette,
       title = "Cylinders")
```



Discussion:

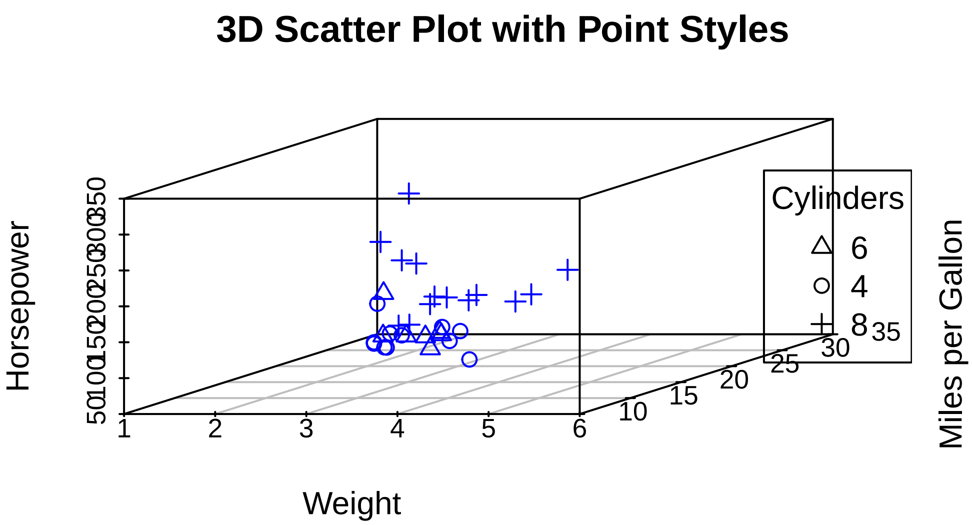
- We defined a color palette using the `rainbow` function based on the number of unique cylinder values in `tb$cyl`.
- We then used this color palette to color the points in the scatter plot.
- Finally, we used the `legend` function to manually add a color legend to the right of the plot. The `legend` function takes the unique cylinder values as the labels for the legend and the colors from the color palette as the fill colors. The title of the legend is set to "Cylinders". [2]

Change Point Style:

We can change the point style to better distinguish between data points. [2]

```
# Create a scatterplot3d and specify point shapes by the number of cylinders
sp34 <- scatterplot3d(x = tb$wt,
                      y = tb$mpg,
                      z = tb$hp,
                      pch = as.numeric(tb$cyl),  # point shapes
                      color = "blue",  # point color to blue
                      xlab = "Weight",
                      ylab = "Miles per Gallon",
                      zlab = "Horsepower",
                      main = "3D Scatter Plot with Point Styles")

# Get unique cylinder values and corresponding point shapes
cylinders <- unique(tb$cyl)
shapes <- as.numeric(cylinders)
```

```
# Add a legend for point shapes
legend("right",
       legend = as.character(cylinders),  # cylinders
       pch = shapes,  # Use point shapes as legend
       title = "Cylinders")  # Set legend title
```



Discussion:

- This R code visualizes relationships among the `wt`, `mpg`, and `hp` columns from the `tb` tibble, with points having different shapes based on the number of cylinders (`cyl`).
- **Creation of 3D Scatter Plot**
  - `sp34 <- scatterplot3d(...)`: This line initializes the creation of a 3D scatter plot and stores the result in the `s3d` variable.
  - `x = tb$wt, y = tb$mpg, z = tb$hp`: These arguments define the variables that will be plotted on the x, y, and z axes, respectively.
  - `pch = as.numeric(tb$cyl)`: This argument specifies the point shape (`pch`) to vary based on the `cyl` column, with different cylinders represented by different shapes.
  - `color = "blue"`: All points are colored blue.
  - `xlab`, `ylab`, `zlab`: These arguments label the axes of the plot.
  - `main`: This argument provides the main title of the plot.
- **Determination of Unique Cylinder Values and Shapes**
  - `cylinders <- unique(tb$cyl)`: This line identifies the unique values in the `cyl` column and stores them in the `cylinders` variable.

- `shapes <- as.numeric(cylinders)`: The unique cylinder values are converted to numeric form to determine the corresponding point shapes, which are stored in `shapes`.

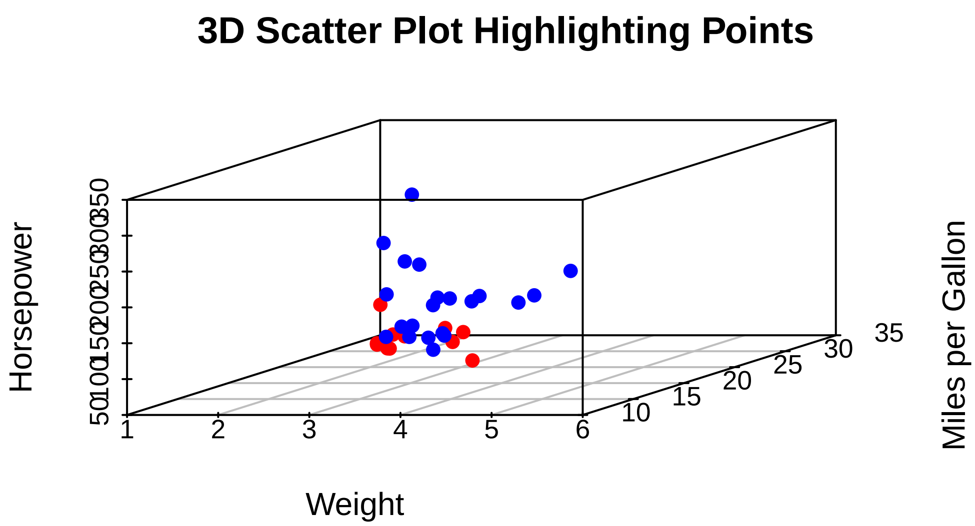
- **Addition of Legend**
  - `legend("right", ...)`: This function is used to add a legend to the right of the plot.
  - `legend = as.character(cylinders)`: This argument defines the labels of the legend, which are the unique cylinder values converted to character strings.
  - `pch = shapes`: The point shapes corresponding to the unique cylinder values are used in the legend.
  - `title = "Cylinders"`: The title of the legend is set as "Cylinders".

- By executing this code, we get a 3D scatter plot with points having different shapes based on the number of cylinders, and a corresponding legend clarifying the representation of each shape. [2]

Highlight Specific Points:

We can can highlight specific points, for example, cars with 4 cylinders, by changing their color or size. [2]

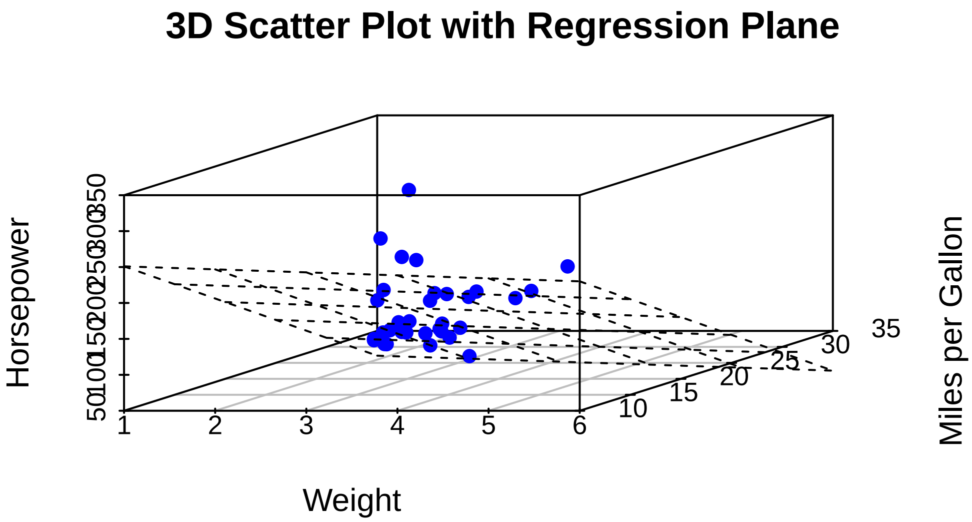
```
# Create a scatterplot3d with specific point highlighting
sp35 <- scatterplot3d(x = tb$wt,
                      y = tb$mpg,
                      z = tb$hp,
                      pch = 16,  # Use a specific point shape
                      color = ifelse(tb$cyl == 4, "red", "blue"),
                      # Color points based on the number of cylinders
                      xlab = "Weight",
                      ylab = "Miles per Gallon",
                      zlab = "Horsepower",
                      main = "3D Scatter Plot Highlighting Points")
```



Add Regression Plane:

We can fit a linear regression model and add the regression plane to the scatter plot. [2]

```
library(scatterplot3d)

# Create a 3D scatter plot
sp36 <- scatterplot3d(x = tb$wt, y = tb$mpg, z = tb$hp,
                      pch = 16, color = "blue",
                      xlab = "Weight",
                      ylab = "Miles per Gallon",
                      zlab = "Horsepower",
                      main = "3D Scatter Plot with Regression Plane")

# Fit a linear model and add a regression plane to 3D plot
model <- lm(hp ~ wt + mpg, data = tb)
sp36$plane3d(model)
```



Discussion:

- This code snippet is used to create a 3D scatter plot and then fit a linear regression plane to the plotted data points.
- **Fit a Linear Model**
  - `model <- lm(hp ~ wt + mpg, data = tb)`: A linear model is fitted using the `lm` function with `hp` as the dependent variable and `wt` and `mpg` as the independent variables. The resulting model object is stored in the variable `model`.
- **Add a Regression Plane to the Plot**
  - `sp36$plane3d(model)`: The `plane3d` function is accessed from the `s3d` object and is used to add a regression plane to the existing 3D scatter plot. The plane is based on the coefficients of the linear model stored in `model`.
- The resulting output will be a 3D scatter plot visualizing the relationships among weight (`wt`), miles per gallon (`mpg`), and horsepower (`hp`), with a regression plane fitted to these points, representing the linear relationship between the dependent and independent variables. [2], [4]

16.4 Summary of Chapter 16 – Three Dimensional (3D) Data

This chapter provides an insightful exploration into the creation of 3D plots using R, focusing on the ggplot2 and scatterplot3d packages.

ggplot2: The chapter begins by highlighting that `ggplot2`, primarily known for 2D plots, can be used to simulate 3D effects. This is demonstrated through a scatter plot example where visual cues like point size and color gradients are employed to give an illusion of depth. By mapping variables such as horsepower to these visual elements in the 'mtcars' dataset, the plot achieves a 3D-like effect on a 2D plane.

scatterplot3d: The `scatterplot3d` package, specifically designed for 3D scatter plots, is explored next. The chapter illustrates its straightforward application in creating 3D plots with the mtcars data, considering variables like weight, miles per gallon, and horsepower. It delves into various enhancements like coloring points based on categorical variables (e.g., the number of cylinders), changing point styles to distinguish data points, highlighting specific points for emphasis, and adding legends for clarity. The chapter also demonstrates how to incorporate a linear regression plane into the scatter plot, adding an analytical dimension to the visual representation.

Throughout the chapter, code examples and discussions are provided to guide the reader on how to effectively use these packages for 3D plotting in R. The focus is on practical application, offering readers the tools to transform standard 2D visualizations into more engaging and informative 3D plots. This approach not only enhances the visual appeal of the data but also allows for a more nuanced understanding of complex relationships within the dataset.

16.5 References

ggplot2

Wickham, H. (2016). *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York. Retrieved from https://ggplot2.tidyverse.org

Wickham, H., & Grolemund, G. (2016). *R for Data Science: Import, Tidy, Transform, Visualize, and Model Data*. O'Reilly Media.

Wickham, H. (2020). *ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics* (Version 3.3.2) [Computer software]. Retrieved from https://CRAN.R-project.org/package=ggplot2

Wickham, H., et al. (2020). *dplyr: A Grammar of Data Manipulation* (Version 1.0.2) [Computer software]. Retrieved from https://CRAN.R-project.org/package=dplyr

Wilkinson, L. (2005). *The Grammar of Graphics* (2nd ed.). Springer-Verlag.

Wickham, H., et al. (2020). *tibble: Simple Data Frames* (Version 3.0.3) [Computer software]. Retrieved from https://CRAN.R-project.org/package=tibble

Ligges, U., Maechler, M., Schnackenberg, S., & Ligges, M. U. (2018). *Package 'scatterplot3d'*. Retrieved from https://cran.rproject.org/web/packages/scatterplot3d/scatterplot3d.pdf

Ligges, U., & Mächler, M. (2002). *Scatterplot3d - an R package for visualizing multivariate data*. Technical report, 2002, 22.

3D Visualization

Adler, D., Nenadic, O., & Zucchini, W. (2003, March). *Rgl: A R-library for 3D visualization with OpenGL*. In Proceedings of the 35th Symposium of the Interface: Computing Science and Statistics, Salt Lake City (Vol. 35, pp. 1-11).

Regression

Fox, J., & Weisberg, S. (2011). *An R Companion to Applied Regression* (2nd ed.). Sage.

17 Case (1 of 2): An Overview of the S&P500

Chapter 17.

17.1 S&P 500

The S&P 500, also called the Standard & Poor's 500, is a stock market index that tracks the performance of 500 major publicly traded companies listed on U.S. stock exchanges. It serves as a widely accepted benchmark for assessing the overall health and performance of the U.S. stock market.

S&P Dow Jones Indices, a division of S&P Global, is responsible for maintaining the index. The selection of companies included in the S&P 500 is determined by a committee, considering factors such as market capitalization, liquidity, and industry representation.

The S&P is a float-weighted index, meaning the market capitalizations of the companies in the index are adjusted by the number of shares available for public trading. [1]

The performance of the S&P 500 is frequently used to gauge the broader stock market and is commonly referenced by investors, analysts, and financial media. It provides a snapshot of how large-cap U.S. stocks are faring and is considered a reliable indicator of overall market sentiment.

Aside: Typically, the S&P 500 index consists of 500 stocks. However, in reality, there are actually 503 stocks included. This discrepancy arises because three of the listed companies have multiple share classes, and each class is considered a separate stock that needs to be included in the index. [1]

Strengths:

1. **Diverse Representation**: The S&P 500 isn't fixated on a single industry. From technology to healthcare, it offers a panoramic view of various economic sectors, making it an inclusive representation of the U.S. corporate sector.
2. **Benchmark for Investors**: For many fund managers, outperforming the S&P 500 stands as a golden standard. It's a yardstick, establishing it as a critical touchstone for gauging investment success.

3. **Liquidity and Visibility**: Constituent companies enjoy high liquidity and are subject to rigorous screening processes, ensuring that the index represents financially viable entities.

Critiques:

1. **Market Capitalization Weighting**: The index is weighted by market capitalization, meaning companies with higher market values have a more pronounced effect on its performance. Critics argue this approach can skew perceptions, especially during market bubbles when certain sectors are overvalued.
2. **Exclusivity**: Despite its broad purview, 500 companies cannot encapsulate the entire U.S. economy. Many sectors, especially emerging industries or smaller businesses, might not be adequately represented.
3. **Potential for Complacency**: The prominence of the S&P 500 has led many investors to adopt passive investment strategies, tracking the index rather than actively managing portfolios. Detractors argue this might lead to market inefficiencies and reduced capital allocation efficacy.

While the S&P 500 remains an influential and pivotal tool for investors, its dominance prompts a double-edged sword of advantages and critiques. In a constantly evolving economic landscape, understanding both its power and limitations is essential for informed financial decision-making. [2]

The broad purpose of this Case Study is to review and analyze the different sectors and stocks within the S&P500.

17.2 S&P 500 Data

17.2.1 Load some useful R packages

```
# Load the required libraries, suppressing annoying startup messages
library(dplyr, quietly = TRUE, warn.conflicts = FALSE) # data manipulation
library(tibble, quietly = TRUE, warn.conflicts = FALSE) # data manipulation
library(ggplot2, quietly = TRUE, warn.conflicts = FALSE) # data visualization
library(ggpubr, quietly = TRUE, warn.conflicts = FALSE) # data visualization

library(gsheet, quietly = TRUE, warn.conflicts = FALSE) # Google Sheets
library(rmarkdown, quietly = TRUE, warn.conflicts = FALSE) # writing
library(knitr, quietly = TRUE, warn.conflicts = FALSE) # tables
library(kableExtra, quietly = TRUE, warn.conflicts = FALSE) # tables
library(scales)  # For formatting currency
```

17.2.2 Read the S&P500 data from a Google Sheet into a tibble

1. We will analyze a real-world, recent dataset containing information about the S&P500 stocks, sourced from TradingView.com. [3]
2. The dataset is located in a Google Sheet and periodically updated.
3. The complete URL of the Google Sheet that has the data is

   https://docs.google.com/spreadsheets/d/14mUlNNpeuV2RouT9MKaAWKUpvjRijzQu40DdWJgyKPQ/
4. Its Google Sheet ID is: `14mUlNNpeuV2RouT9MKaAWKUpvjRijzQu40DdWJgyKPQ`.

17.3 Loading the data into R

1. We can use the function `gsheet2tbl` in package `gsheet` to read the Google Sheet into a tibble , as demonstrated in the following code.

```
# Read S&P500 stock data present in a Google Sheet.
library(gsheet)
prefix <- "https://docs.google.com/spreadsheets/d/"
sheetID <- "14mUlNNpeuV2RouT9MKaAWKUpvjRijzQu40DdWJgyKPQ"
url500 <- paste(prefix,sheetID) # Form the URL to connect to
sp500Data <- gsheet2tbl(url500) # Read it into a tibble
```

2. **Note**: This data is current, as of **Fri, Jan 5, 2024**

17.4 S&P Global Industry Classification Standard (GICS®)

1. In this case study, we will classify and analyze the S&P 500 stocks based on the GICS standard!
2. The Global Industry Classification Standard (GICS®) was developed in 1999 by S&P Dow Jones Indices and MSCI. The GICS methodology aims to enhance the investment research and asset management process for financial professionals worldwide. The GICS methodology has been widely accepted as an industry analysis framework for investment research, portfolio management and asset allocation. [4]
3. The GICS classification consists of **11** sectors, – {Communication Services, Consumer Discretionary, Consumer Staples, Energy, Financials, Health Care, Industrials, Information Technology, Materials, Real Estate, Utilities}. The classification of each stock in the S&P 500 according to GICS is available at the following Google Sheet:

https://docs.google.com/spreadsheets/d/1WrVA8dPYvQsc_mXVctgTntRLS02qd7ubzcdAsw03Lgk/

4. For this file, the Google Sheet ID is `1WrVA8dPYvQsc_mXVctgTntRLS02qd7ubzcdAsw03Lgk`and we read this classification data into a tibble, we name `gics`, using similar code.

```
# Read GICS classificaiton of S&P 500 stocks from a Google Sheet.
library(gsheet)
prefix2 <- "https://docs.google.com/spreadsheets/d/"
sheetID2 <- "1WrVA8dPYvQsc_mXVctgTntRLS02qd7ubzcdAsw03Lgk"
urlgics <- paste(prefix2, sheetID2) # Form the URL to connect to
gics <- gsheet2tbl(urlgics) # Read it into a tibble called gics
```

5. Next, we join the two tibbles, using "Stock" as the key and name our joint tibble `sp500`, as follows.

```
# Merging dataframes
sp500 <- merge(sp500Data,
               gics ,
               id = "Stock")
```

17.5 Review the S&P 500 data

1. The data corresponds to **503** companies that are part of the S&P500 and includes 39 data columns, as of **Fri, Jan 5, 2024**

```
dim(sp500)
```

```
[1] 503  39
```

2. The first ten stocks in the S&P500 data, their GICS Sector and their recent prices are as follows:

```
sp500 %>%
  select(Stock, Description, GICSSector) %>%
  head(10) %>%
  kable("html",
        caption = "The first 10 companies in the S&P500") %>%
  kable_styling()
```

Table 17.1: The first 10 companies in the S&P500

Stock	Description	GICSSector
A	Agilent Technologies, Inc.	Health Care
AAL	American Airlines Group, Inc.	Industrials
AAPL	Apple Inc.	Information Technology
ABBV	AbbVie Inc.	Health Care
ABNB	Airbnb, Inc.	Consumer Discretionary
ABT	Abbott Laboratories	Health Care
ACGL	Arch Capital Group Ltd.	Financials
ACN	Accenture plc	Information Technology
ADBE	Adobe Inc.	Information Technology
ADI	Analog Devices, Inc.	Information Technology

3. Data Columns

- The data comprises of the following 39 columns:

```r
colnames(sp500)
```

```
 [1] "Stock"
 [2] "Date"
 [3] "Description"
 [4] "Sector"
 [5] "Industry"
 [6] "Market Capitalization"
 [7] "Price"
 [8] "52 Week Low"
 [9] "52 Week High"
[10] "Return on Equity (TTM)"
[11] "Return on Assets (TTM)"
[12] "Return on Invested Capital (TTM)"
[13] "Gross Margin (TTM)"
[14] "Operating Margin (TTM)"
[15] "Net Margin (TTM)"
[16] "Price to Earnings Ratio (TTM)"
[17] "Price to Book (FY)"
[18] "Enterprise Value/EBITDA (TTM)"
[19] "EBITDA (TTM)"
[20] "EPS Diluted (TTM)"
[21] "EBITDA (TTM YoY Growth)"
[22] "EBITDA (Quarterly YoY Growth)"
```

```
[23] "EPS Diluted (TTM YoY Growth)"
[24] "EPS Diluted (Quarterly YoY Growth)"
[25] "Price to Free Cash Flow (TTM)"
[26] "Free Cash Flow (TTM YoY Growth)"
[27] "Free Cash Flow (Quarterly YoY Growth)"
[28] "Debt to Equity Ratio (MRQ)"
[29] "Current Ratio (MRQ)"
[30] "Quick Ratio (MRQ)"
[31] "Dividend Yield Forward"
[32] "Dividends per share (Annual YoY Growth)"
[33] "Price to Sales (FY)"
[34] "Revenue (TTM YoY Growth)"
[35] "Revenue (Quarterly YoY Growth)"
[36] "Technical Rating"
[37] "Security"
[38] "GICSSector"
[39] "GICSSubIndustry"
```

- The names of the data columns are self-explanatory. The Financial terms are explained in depth on multiple external websites such as www.Investopedia.com

17.5.1 Rename Data Columns

4. The names of the data columns are lengthy and confusing. We will rename the data columns to make it easier to work with the data.

```
# Define a mapping of new column names
new_names <- c(
  "Stock", "Date", "StockName", "Sector", "Industry",
  "MarketCap", "Price", "Low52Wk", "High52Wk",
  "ROE", "ROA", "ROIC", "GrossMargin",
  "OperatingMargin", "NetMargin", "PE",
  "PB", "EVEBITDA", "EBITDA", "EPS",
  "EBITDA_YOY", "EBITDA_QYOY", "EPS_YOY",
  "EPS_QYOY", "PFCF", "FCF",
  "FCF_QYOY", "DebtToEquity", "CurrentRatio",
  "QuickRatio", "DividendYield",
  "DividendsPerShare_YOY", "PS",
  "Revenue_YOY", "Revenue_QYOY", "Rating",
  "Security", "GICSSector", "GICSSubIndustry"
)
```

```r
# Rename the columns using the new_names vector
colnames(sp500)<-new_names
```

5. We review the column names again after renaming them, using the `colnames()` function.

```r
colnames(sp500)
```

```
 [1] "Stock"                "Date"                 "StockName"
 [4] "Sector"               "Industry"             "MarketCap"
 [7] "Price"                "Low52Wk"              "High52Wk"
[10] "ROE"                  "ROA"                  "ROIC"
[13] "GrossMargin"          "OperatingMargin"      "NetMargin"
[16] "PE"                   "PB"                   "EVEBITDA"
[19] "EBITDA"               "EPS"                  "EBITDA_YOY"
[22] "EBITDA_QYOY"          "EPS_YOY"              "EPS_QYOY"
[25] "PFCF"                 "FCF"                  "FCF_QYOY"
[28] "DebtToEquity"         "CurrentRatio"         "QuickRatio"
[31] "DividendYield"        "DividendsPerShare_YOY" "PS"
[34] "Revenue_YOY"          "Revenue_QYOY"         "Rating"
[37] "Security"             "GICSSector"           "GICSSubIndustry"
```

17.5.2 Understand the Data Columns

6. Our next goal is to gain a deeper understanding of what the data columns mean. We reorganize the column names into eight tables, labeled Table 1a, 1b.. 1h.

a. The column names described in Table 1a. concern basic **Company Information** of each stock.

Table 1a: Data Columns giving basic Company Information

ColumnName	Description
Stock	Stock Ticker (e.g. AAL)
Date	Date (e.g. ”7/15/2023”)
StockName	Name of the company (e.g ”American Airlines Group, Inc.”)
GICSSector	Sector, as per GICS Classification
GICSSubIndustry	Sub-Industry, as per GICS Classification
MarketCap	Market capitalization of the company
Price	Recent Stock Price

b. The column names described in Table 1b. are related to **Technical Analysis**, including the 52-Week High and Low prices.

Table 1b: Data Columns related to Pricing and Technical Analysis

ColumnName	Description
Low52Wk	52-Week Low Price
High52Wk	52-Week High Price
Rating	Technical Rating

c. The column names described in Table 1c. are related to the **Profitability** of each stock.

Table 1c: Data Columns related to Profitability

ColumnName	Description
ROE	Return on Equity
ROA	Return on Assets
ROIC	Return on Invested Capital
GrossMargin	Gross Profit Margin
OperatingMargin	Operating Profit Margin
NetMargin	Net Profit Margin

d. The column names described in Table 1d are related to the **Earnings** of each stock.

Table 1d: Data Columns related to Earnings

ColumnName	Description
PE	Price-to-Earnings Ratio
PB	Price-to-Book Ratio
EVEBITDA	Enterprise Value to EBITDA Ratio
EBITDA	EBITDA
EPS	Earnings per Share
EBITDA_YOY	EBITDA Year-over-Year Growth
EBITDA_QYOY	EBITDA Quarterly Year-over-Year Growth
EPS_YOY	EPS Year-over-Year Growth
EPS_QYOY	EPS Quarterly Year-over-Year Growth

e. The column names described in Table 1e are related to the **Free Cash Flow** of each stock.

Table 1e: Data Columns related to Free Cash Flow

ColumnName	Description
PFCF	Price-to-Free Cash Flow
FCF	Free Cash Flow
FCF_QYOY	Free Cash Flow Quarterly Year-over-Year Growth

f. The column names described in Table 1f concern the **Liquidity** of each stock.

Table 1f: Data Columns related to Liquidiy

ColumnName	Description
DebtToEquity	Debt-to-Equity Ratio
CurrentRatio	Current Ratio
QuickRatio	Quick Ratio

g. The column names described in Table 1g are related to the **Revenue** of each stock.

Table 1g: Data Columns related to Revenue

ColumnName	Description
PS	Price-to-Sales Ratio
Revenue_YOY	Revenue Year-over-Year Growth
Revenue_QYOY	Revenue Quarterly Year-over-Year Growth

h. The column names described in Table 1h are related to the **Dividends** of each stock.

Table 1h: Data Columns related to Dividends

ColumnName	Description
DividendYield	Dividend Yield
DividendsPerShare_YOY	Annual Dividends per Share Year-over-Year Growth

17.5.3 Stock Prices, 52-Week Low, High; Market Cap in Billions

We want to analyze stock prices relative to their 52 Week Low and 52 Week High respectively, to understand their relative price attractiveness.

Hence, a new column named `Low52WkPerc` is being added. The column contains the percentage change between the current price (`Price`) and its 52-week low (`Low52Wk`). The formula used is:

$$Low52WkPerc = \frac{(CurrentPrice - 52WeekLow) * 100}{52WeekLow}$$

Another column named `High52WkPerc` represents the percentage change between the 52-week high (`High52Wk`) and the current price (`Price`). We round off the data to two decimal places for clarity.

```
library(dplyr)
sp500 <- sp500 %>%
  mutate(Low52WkPerc = round((Price - Low52Wk) * 100 / Low52Wk, 2),
         High52WkPerc = round((High52Wk - Price) * 100 / High52Wk, 2),
         MarketCapBillions = round(MarketCap / 1e9, 3) # MarketCap to billions
         )
```

For convenience, we format the Prices.

```
library(dplyr)
library(scales)  # For formatting currency

sp500 <- sp500 %>%
  mutate(
    Price = scales::dollar(round(Price, 2)),  # format the Price as $
    High52Wk = scales::dollar(round(High52Wk, 2)),  # format 52 Week High
    Low52Wk = scales::dollar(round(Low52Wk, 2))  # format 52 Week Low
  )
```

17.5.4 Analysis of Stock Ratings

1. In the data, the S&P500 shares have Technical Ratings such as {Strong Buy, Buy, Neutral, Sell, Strong Sell}. Since each Stock has a unique Technical Rating, it makes sense to model the data column Rating as a `factor()` variable.

```
sp500$Rating <- as.factor(sp500$Rating)
```

2. We confirm that Rating is now modelled as a factor variable, using `str()` and use `levels()` to review the different levels it can take.

```
str(sp500$Rating)
```

```
 Factor w/ 5 levels "Buy","Neutral",..: 1 1 3 4 1 3 3 4 2 3 ...
```

```r
levels(sp500$Rating)
```

```
[1] "Buy"          "Neutral"     "Sell"        "Strong Buy"  "Strong Sell"
```

3. The `table()` function allows us to count how many stocks have each Rating. see how many stocks have ratings ranging from "Strong Sell" to "Strong Buy". This completes our review of Rating.

```r
table(sp500$Rating)
```

```

        Buy     Neutral        Sell  Strong Buy Strong Sell 
        192          51         178          58          24 
```

17.5.5 Analysis of GICS Sectors in the S&P500

- The S&P 500 comprises a wide array of sectors, reflecting the diverse American corporate landscape.
- The data showcases the S&P500 divided across 11 Sectors. Each stock belongs to a unique sector and it makes sense to model `GICSSector` as a `factor`.

```r
sp500$GICSSector <- as.factor(sp500$GICSSector)
```

2. We confirm that `GICSSector` is now modelled as a factor variable and review the different levels it can take.

```r
str(sp500$GICSSector)
```

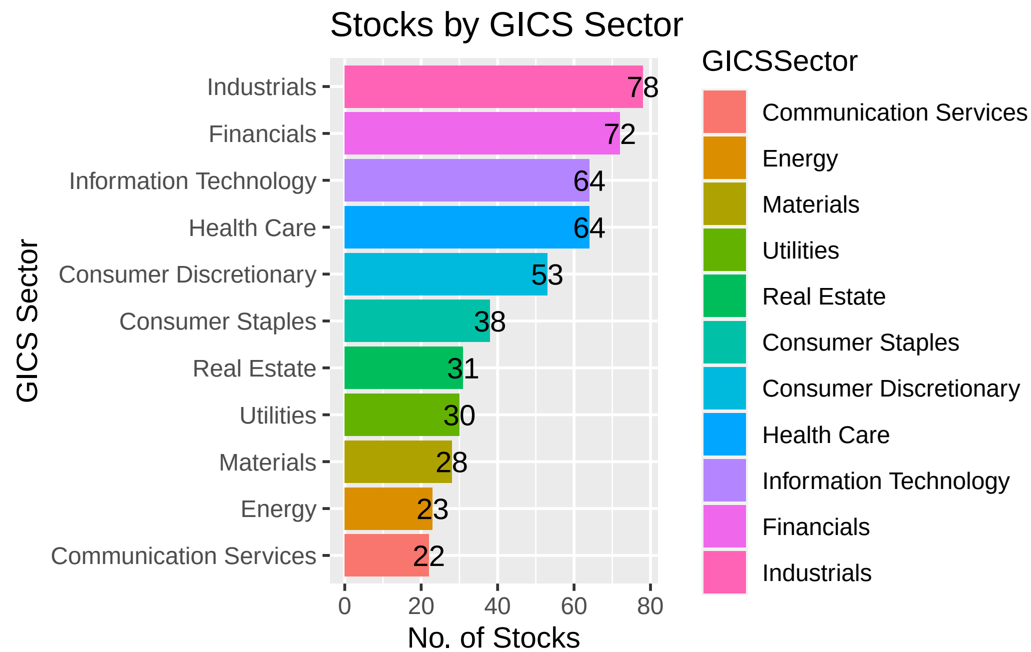
```
 Factor w/ 11 levels "Communication Services",..: 6 7 8 6 2 6 5 8 8 8 ...
```

```r
levels(sp500$GICSSector)
```

```
 [1] "Communication Services" "Consumer Discretionary" "Consumer Staples"      
 [4] "Energy"                 "Financials"             "Health Care"           
 [7] "Industrials"            "Information Technology" "Materials"             
[10] "Real Estate"            "Utilities"             
```

- We note that the S&P500 consists of 503 stocks, divided across 11 sectors.

```r
library(ggplot2)  # For creating plots
library(dplyr)    # For data manipulation

sp500 %>%
  mutate(
    # Reorder the 'GICSSector' factor levels based on each sector
    # 'table(GICSSector)[GICSSector]' calculates the frequency of each sector
    # 'reorder' reorders the levels of 'GICSSector' based on these frequencies
    GICSSector = reorder(GICSSector,
                         table(GICSSector)[GICSSector])
  ) %>%
  # Start a ggplot with 'GICSSector' on the y-axis
  ggplot(aes(y = GICSSector)) +
    # Create a bar plot; 'geom_bar' counts the frequency for each sector
    # 'fill = GICSSector' colors the bars based on the sector
    geom_bar(aes(fill = GICSSector)) +
    # Add text labels on the bars showing the count of stocks in each sector
    # 'stat = "count"' calculates the count for each sector
    # 'label = after_stat(count)' adds these counts as labels on the bars
    geom_text(stat = 'count',
              aes(label = after_stat(count))) +
    labs(title = "Stocks by GICS Sector",  # Title of the plot
         x = "No. of Stocks",              # Label for the x-axis
         y = "GICS Sector")                # Label for the y-axis
```



- Thus, we can see how many stocks are part of each sector. We can sum them to confirm that they add up to 503 stocks.

17.5.6 MarketCap by GICS Sector

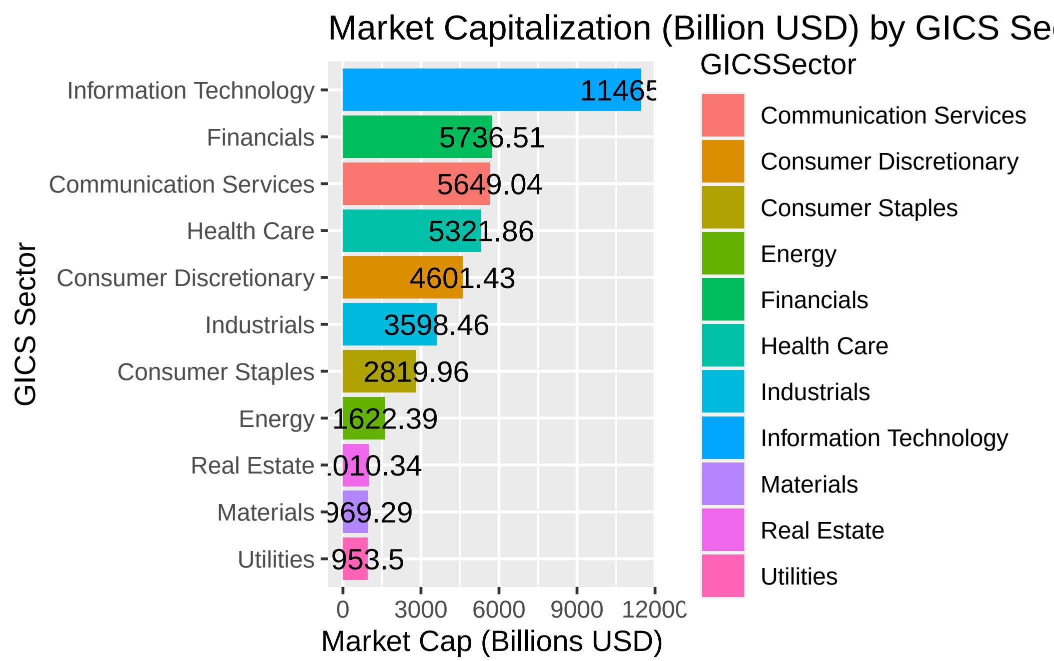
1. We review the Market Cap of S&P500 stocks across GICS Sectors. We summarize the total Market Cap for each GICS Sector, using the following code.

```
# Calculate Market Cap by Sector

MarketCapbySector <- sp500 %>%
  mutate(Market_Cap_Billions = round(MarketCap / 1000000000, 2)) %>%
  group_by(GICSSector) %>%
  summarise(MarketCapBillions = sum(Market_Cap_Billions, na.rm = TRUE)) %>%
  arrange(-MarketCapBillions)
```

2. We create a bar plot of Market Cap by GICS Sector

```
# Create a bar plot of Market Cap by GICS Sector,
ggplot(MarketCapbySector,
       aes(y = reorder(GICSSector,
                       MarketCapBillions),  # Y-axis: GICSSector reordered
           x = MarketCapBillions,  # X-axis: Market Capitalization in billions
```

```
            fill = GICSSector)) +  # Fill color of the bars based on GICSSector
  geom_bar(stat = "identity") +  # 'stat = "identity"' to use MarketCapBillions for bars
  labs(title = "Market Capitalization (Billion USD) by GICS Sector",
       y = "GICS Sector",  # Label for the y-axis
       x = "Market Cap (Billions USD)") +  # Label for the x-axis
  geom_text(aes(label = MarketCapBillions))  # Add text labels to the bars
```



3. The S&P500 has a combined Market Cap of 43748.27 Billion USD.

17.5.7 Highest Market Cap Stocks in each GICS Sector

1. Suppose we wanted to find the top two stocks with the highest market capitalization in each GICS Sector.
2. We could group the data by **`GICSSector`**; arrange the data in descending order of **`MarketCap`** within each sector; slice the top 2 entries for each group. Here's the R code to accomplish this:

```
# Find the top two stocks by MarketCap in each GICS Sector
top_stocks_by_sector <- sp500 %>%
  group_by(GICSSector) %>%
  arrange(desc(MarketCap)) %>%
  slice_head(n = 2) %>%
```

```
  ungroup() %>%
  arrange(GICSSector)  # Arrange the final data by GICSSector

# Select only the specified columns and create a table using kable
top_stocks_by_sector %>%
  select(GICSSector, Stock, StockName, MarketCapBillions) %>%
  kable("html",
        caption = "Top Two Stocks by Market Cap, for each Sector") %>%
  kable_styling()
```

Table 17.10: Top Two Stocks by Market Cap, for each Sector

GICSSector	Stock	StockName	MarketCapBillions
Communication Services	GOOG	Alphabet Inc.	1723.840
Communication Services	GOOGL	Alphabet Inc.	1723.430
Consumer Discretionary	AMZN	Amazon.com, Inc.	1504.950
Consumer Discretionary	TSLA	Tesla, Inc.	751.020
Consumer Staples	WMT	Walmart Inc.	421.765
Consumer Staples	PG	Procter & Gamble Company (The)	349.243
Energy	XOM	Exxon Mobil Corporation	412.872
Energy	CVX	Chevron Corporation	284.572
Financials	BRK.B	Berkshire Hathaway Inc. New	796.444
Financials	V	Visa Inc.	523.588
Health Care	LLY	Eli Lilly and Company	583.748
Health Care	UNH	UnitedHealth Group Incorporated	499.797
Industrials	BA	Boeing Company (The)	149.284
Industrials	CAT	Caterpillar, Inc.	147.054
Information Technology	AAPL	Apple Inc.	2831.530
Information Technology	MSFT	Microsoft Corporation	2750.200
Materials	LIN	Linde plc	198.572
Materials	SHW	Sherwin-Williams Company (The)	76.078
Real Estate	PLD	Prologis, Inc.	123.424
Real Estate	AMT	American Tower Corporation (REIT)	100.440
Utilities	NEE	NextEra Energy, Inc.	126.878
Utilities	SO	Southern Company (The)	78.045

17.5.8 Prices relative to 52-Week-Low and 52-Week-High of each GICS Sector

We analyze the distribution of stock prices relative to their 52 Week Low, measured as `Low52WkPerc = (Price - 52-Week-Low)/(52-Week-Low)`. In fact, we review this distribution for each GICS Sector, to understand which sector has stocks priced relatively closest to their 52-Week Low prices.

We also do this for stock prices relative to their 52 Week High, measured as `High52WkPerc = (52-Week-High - Price)/(52-Week-High)`.

Table 2

```r
library(dplyr)
library(kableExtra)

# Q1, Median, Q3 for Low52WkPerc, High52WkPerc within each sector
sector_summary_stats <- sp500 %>%
  group_by(GICSSector) %>%
  summarise(
    Low_Q1 = round(quantile(Low52WkPerc, 0.25, na.rm = TRUE), 1),
    Low_Median = round(median(Low52WkPerc, na.rm = TRUE), 1),
    Low_Q3 = round(quantile(Low52WkPerc, 0.75, na.rm = TRUE), 1),
    High_Q1 = round(quantile(High52WkPerc, 0.25, na.rm = TRUE), 1),
    High_Median = round(median(High52WkPerc, na.rm = TRUE), 1),
    High_Q3 = round(quantile(High52WkPerc, 0.75, na.rm = TRUE), 1)
  ) %>%
  ungroup()

# Sort by Low Q1
sector_summary_stats <- sector_summary_stats %>%
  arrange(Low_Q1)

# Reapply formatting with percentage symbol
sector_summary_stats <- sector_summary_stats %>%
  mutate(
    across(ends_with("Q1"), ~paste0(., "%")),
    across(ends_with("Median"), ~paste0(., "%")),
    across(ends_with("Q3"), ~paste0(., "%"))
  )

# Create a formatted table using kable
sector_summary_stats %>%
  kable("html",
        caption = "Q1, Median, Q3 of Low52WkPerc, High52WkPerc") %>%
  kable_styling()
```

Table 17.11: Q1, Median, Q3 of Low52WkPerc, High52WkPerc

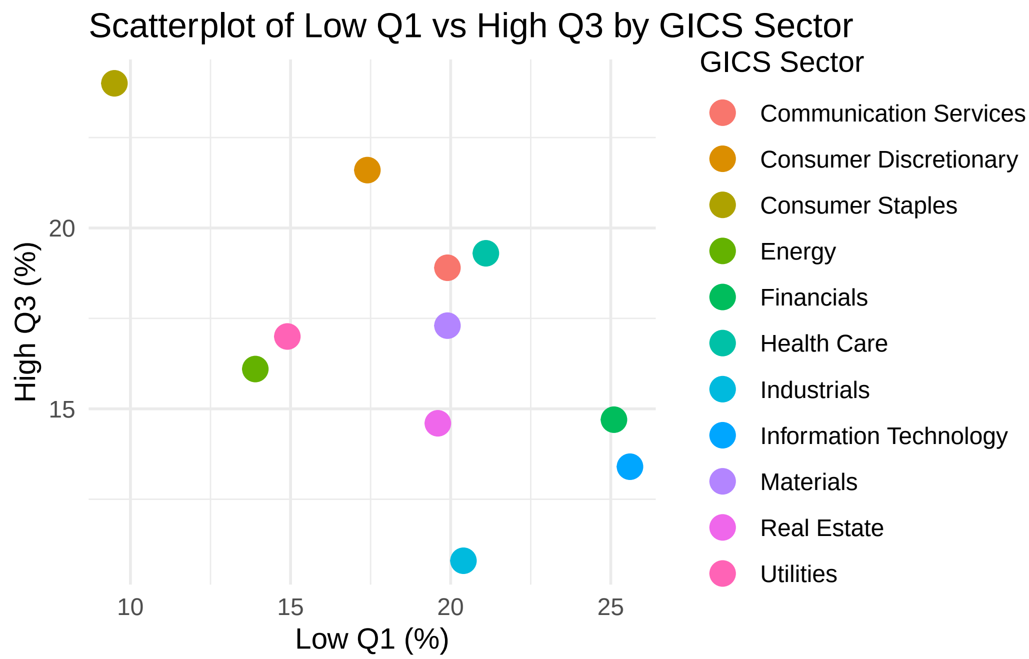
GICSSector	Low_Q1	Low_Median	Low_Q3	High_Q1	High_Median	High_Q3
Consumer Staples	9.5%	18%	22.1%	9.2%	15.5%	24%
Energy	13.9%	27.4%	30.2%	7.1%	12.7%	16.1%
Utilities	14.9%	18.2%	22.1%	7.3%	9.5%	17%
Consumer Discretionary	17.4%	30.4%	47.9%	4.3%	12.7%	21.6%
Real Estate	19.6%	27.8%	36.1%	5.5%	8.1%	14.6%
Communication Services	19.9%	30.1%	56.5%	4.1%	10.8%	18.9%
Materials	19.9%	27.5%	35.1%	4.6%	7.7%	17.3%
Industrials	20.4%	29.5%	42%	3.9%	6.7%	10.8%
Health Care	21.1%	29.7%	39.7%	3.9%	11.2%	19.3%
Financials	25.1%	34%	41.5%	1.8%	7.4%	14.7%
Information Technology	25.6%	37.6%	56.5%	5.8%	8.4%	13.4%

Notice that Low_Q1 represents the 25th percentile of the distribution of Low52WkPerc, while High_Q3 represents the 75th percentile of the distribution of High52WkPerc.

We want to identify the GICS Sector(s) whose stocks are closest to their 52-Week Low and relatively furthest from their 52-Week High prices. Once way of doing this is to create a scatterplot, as follows.

```
library(ggplot2)
library(dplyr)

# Convert percentage strings back to numeric for plotting
plot_data <- sector_summary_stats %>%
  mutate(
    Low_Q1_numeric = as.numeric(gsub("%", "", Low_Q1)),
    High_Q3_numeric = as.numeric(gsub("%", "", High_Q3))
  )

# Create a scatterplot with larger circle sizes
ggplot(plot_data,
       aes(x = Low_Q1_numeric, y = High_Q3_numeric,
           color = GICSSector)) +
  geom_point(size = 4) +  # Increase the size of the points
  labs(
    title = "Scatterplot of Low Q1 vs High Q3 by GICS Sector",
    x = "Low Q1 (%)",
    y = "High Q3 (%)",
    color = "GICS Sector"
```

```
  ) +
  theme_minimal()  # Use a minimal theme for better aesthetics
```

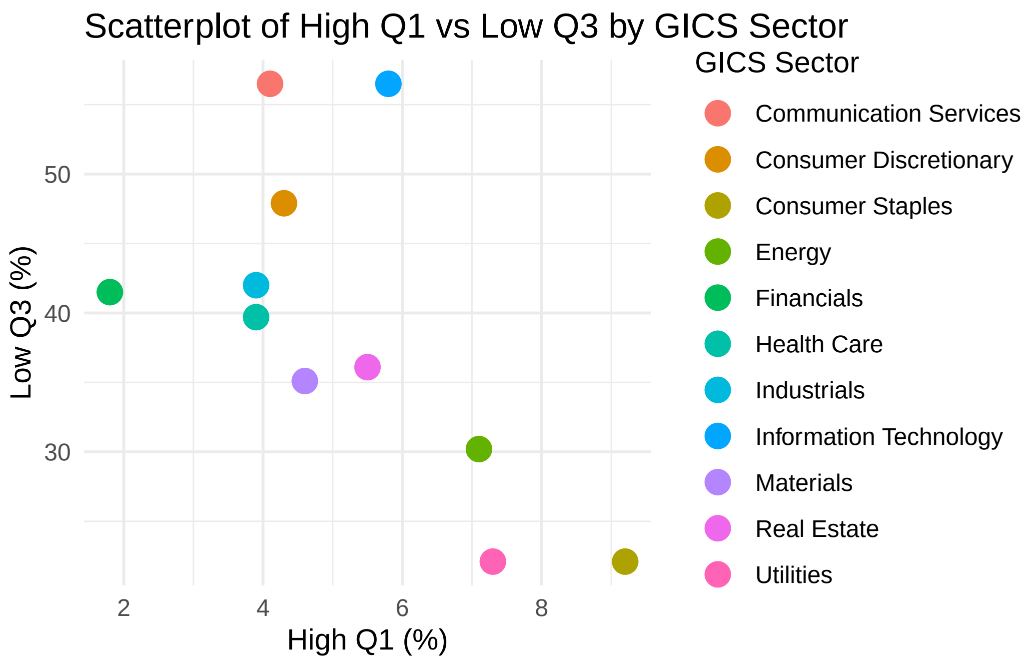


- **This scatterplot suggests that the Consumer Staples sector is closest to 52-Week Low and furthest from 52-Week High.**

Next, we can do the reverse to determine which Sector is furthest away from 52-Week Low and closest to 52-Week High prices.

```
library(ggplot2)
library(dplyr)

# Convert percentage strings back to numeric for plotting
plot_data <- sector_summary_stats %>%
  mutate(
    High_Q1_numeric = as.numeric(gsub("%", "", High_Q1)),
    Low_Q3_numeric = as.numeric(gsub("%", "", Low_Q3))
  )

# Create a scatterplot with larger circle sizes
ggplot(plot_data, aes(x = High_Q1_numeric, y = Low_Q3_numeric, color = GICSSector)) +
  geom_point(size = 4) +  # Increase the size of the points
  labs(
```

```r
    title = "Scatterplot of High Q1 vs Low Q3 by GICS Sector",
    x = "High Q1 (%)",
    y = "Low Q3 (%)",
    color = "GICS Sector"
  ) +
  theme_minimal()  # Use a minimal theme for better aesthetics
```



- **This scatterplot suggests that the Communication Services sector is closest to 52-Week Low and furthest from 52-Week High.**

17.5.9 Profitability of each GICS Sector

The following table gives us the Median of different profitability metrics {ROE, ROA, ROIC, OPM, NPM}, for each GICS sector.

```r
library(dplyr)
library(kableExtra)

# Calculate the median values for Profitability metrics
median_financials <- sp500 %>%
  group_by(GICSSector) %>%
  summarise(
```

```
    Med_ROE = round(median(ROE, na.rm = TRUE), 1),
    Med_ROA = round(median(ROA, na.rm = TRUE), 1),
    Med_ROIC = round(median(ROIC, na.rm = TRUE), 1),
    Med_OPM = round(median(OperatingMargin, na.rm = TRUE), 1),
    Med_NPM = round(median(NetMargin, na.rm = TRUE), 1)
  ) %>%
  arrange(desc(Med_ROE)) # Sort the table based on Median ROE

# Create a formatted table using kable
median_financials %>%
  kable("html",
        caption = "Median Financial Metrics by GICS Sector") %>%
  kable_styling()
```

Table 17.12: Median Financial Metrics by GICS Sector

GICSSector	Med_ROE	Med_ROA	Med_ROIC	Med_OPM	Med_NPM
Information Technology	25.9	10.8	16.0	21.6	17.5
Consumer Discretionary	24.5	12.5	18.1	13.8	9.6
Industrials	23.4	8.3	13.0	16.5	11.2
Energy	22.7	11.1	15.2	25.4	16.2
Consumer Staples	21.3	6.5	9.9	14.6	8.2
Materials	16.5	5.7	9.1	13.9	8.5
Financials	14.5	2.2	8.4	23.1	17.0
Communication Services	12.9	4.7	5.7	17.7	8.6
Health Care	12.6	5.7	8.6	15.1	10.0
Utilities	9.4	2.4	3.7	19.9	11.6
Real Estate	7.2	3.4	3.9	31.2	19.7

- **We notice that the Information Technology GICS Sector has the relatively largest profitability numbers.**
- **We also note that the Consumer Staples has fairly good profitability numbers as well, relative to all the other sectors.**

17.5.10 Earnings of each GICS Sector

We review the distribution of Price to Earnings (PE) and Price to Book (PB) ratios, across different sectors. Obviously, the smaller the PE and PB ratio, the better it is.

We also review how the Earnings Per Share have increased or decreased across the different GICS sectors.

```
library(dplyr)
library(kableExtra)

# Calculate the median values for Earnings related metrics
median_earnings <- sp500 %>%
  group_by(GICSSector) %>%
  summarise(
    Med_PE = round(median(PE, na.rm = TRUE), 1),
    Med_PB = round(median(PB, na.rm = TRUE), 1),
    Med_EPS_QYOY = round(median(EPS_QYOY, na.rm = TRUE), 1),
    Med_EPS_YOY = round(median(EPS_YOY, na.rm = TRUE), 1)
  ) %>%
  ungroup() # Removed the sorting to keep the table unsorted

# Create a formatted table using kable
median_earnings %>%
  kable("html",
        caption = "Earnings Metrics (PE, PB, EPS_QYOY, EPS_YOY)") %>%
  kable_styling()
```

Table 17.13: Earnings Metrics (PE, PB, EPS_QYOY, EPS_YOY)

GICSSector	Med_PE	Med_PB	Med_EPS_QYOY	Med_EPS_YOY
Communication Services	25.7	2.0	20.1	2.1
Consumer Discretionary	21.9	3.7	14.0	11.3
Consumer Staples	23.4	4.4	7.6	-1.4
Energy	11.1	2.5	-26.5	-16.0
Financials	15.3	2.0	9.8	4.5
Health Care	29.5	3.7	0.1	0.5
Industrials	26.0	5.3	11.2	12.5
Information Technology	30.6	6.4	4.4	8.6
Materials	21.7	2.7	-27.4	-29.4
Real Estate	41.6	2.3	-17.1	-20.4
Utilities	19.2	1.9	5.5	-0.9

- **We conlcude that from a Price to Earnings ratio perspective, the PE ratio is relatively lowest for the Energy sector.**
- **From an Earnings per Share Growth perspective, the Consumer Discre-**

tionary GICS sector has performed the best.

Thus, in this survey, we have analyzed the distribution of prices and also the distribution of profitability and earnings growth metrics for the different GICS sector. With this, we conclude our brief descriptive survey of the S&P500 stocks.

17.6 Summary of Chapter 17 – Case (1 of 2): An Overview of the S&P500

Chapter Summary: S&P 500 Case Study

This chapter presents a comprehensive case study of the S&P 500, a crucial index in the U.S. stock market, comprising 500 major publicly traded companies. The chapter begins by introducing the S&P 500, including its management by S&P Dow Jones Indices and its significance as a benchmark for U.S. stock market health. Key strengths of the S&P 500 are highlighted, such as its diverse representation across various economic sectors and its role as a benchmark for investors. However, the chapter also discusses critiques, including the potential for market misrepresentation due to its market capitalization weighting and the issue of exclusivity, as it doesn't fully represent the entire U.S. economy.

The chapter then delves into a detailed data analysis using R programming, starting with loading necessary R packages and reading S&P 500 data from a Google Sheet. The data, sourced from TradingView.com, is organized into a tibble and classified according to the Global Industry Classification Standard (GICS). Further, the chapter categorizes data columns into various aspects like company information, profitability, earnings, and others. It also includes renaming and understanding these data columns for more accessible analysis.

Significant analyses in the chapter include: 1. Stock Prices Analysis: Comparison of stock prices relative to their 52-Week Low and High. 2. GICS Sectors Analysis: Examination of the distribution of stocks and market capitalization across different GICS sectors. 3. Profitability and Earnings Analysis: Evaluation of the profitability and earnings of companies in each GICS sector.

Each analysis is supported with R code snippets and explanations, providing a clear understanding of the methodologies and results. The chapter concludes by summarizing insights from the analyses, such as sector-wise profitability and earnings growth, offering a nuanced understanding of the S&P 500 index's composition and performance.

This chapter serves as an in-depth review and analysis of the S&P 500, blending theoretical knowledge with practical data analysis skills, making it a valuable resource for those interested in financial markets and data analytics.

17.7 References

S&P 500

[1] https://www.investopedia.com/terms/s/sp500.asp

[2] S&P Global: S&P Global. (n.d.). S&P 500. Retrieved September 14, 2023, from https://www.spglobal.com/spdji/en/indices/equity/sp-500/

MarketWatch: MarketWatch. (n.d.). S&P 500 Index. Retrieved September 14, 2023, from https://www.marketwatch.com/investing/index/spx

Bloomberg: Bloomberg. (n.d.). S&P 500 Index (SPX:IND). Retrieved September 14, 2023, from https://www.bloomberg.com/quote/SPX:IND

[3] TradingView.com https://www.tradingview.com/screener/

[4] GICS: Global Industry Classification Standard: https://www.spglobal.com/spdji/en/landing/topic/gics/

18 Case (2 of 2): S&P500 Sector Analysis

Chapter 18.

18.1 Targeted Analysis of a Focal S&P 500 GICS Sector

We are set to conduct a comprehensive analysis focused on a specific GICS sector within the S&P 500. This will involve a close examination of the individual companies comprising the sector we select, allowing us to gain deeper insights into their collective and individual performance dynamics.

```
focalSector = "Consumer Staples"
```

18.1.1 Organize the data

For our upcoming analysis, it's essential to structure and arrange the data effectively, mirroring the methodical approach we successfully implemented in our previous case study

1. Load Packages:

```
# Load the required libraries, suppressing annoying startup messages
library(dplyr, quietly = TRUE, warn.conflicts = FALSE) # For data manipulation
library(tibble, quietly = TRUE, warn.conflicts = FALSE) # For data manipulation
library(ggplot2, quietly = TRUE, warn.conflicts = FALSE) # For data visualization
library(ggpubr, quietly = TRUE, warn.conflicts = FALSE) # For data visualization

library(gsheet, quietly = TRUE, warn.conflicts = FALSE) # For Google Sheets
library(rmarkdown, quietly = TRUE, warn.conflicts = FALSE) # For writing
library(knitr, quietly = TRUE, warn.conflicts = FALSE) # For tables
library(kableExtra, quietly = TRUE, warn.conflicts = FALSE) # For tables
library(scales)  # For formatting currency
```

2. Read the data from a Google Sheet into a tibble:

```r
# Read S&P500 stock data present in a Google Sheet.
library(gsheet)
prefix <- "https://docs.google.com/spreadsheets/d/"
sheetID <- "14mUlNNpeuV2RouT9MKaAWKUpvjRijzQu40DdWJgyKPQ"
url500 <- paste(prefix,sheetID) # Form the URL to connect to
sp500Data <- gsheet2tbl(url500) # Read it into a tibble called sp500Data
```

3. Read GICS classificaiton of S&P 500 stocks:

```r
# Read GICS classificaiton of S&P 500 stocks from a Google Sheet.
library(gsheet)
prefix2 <- "https://docs.google.com/spreadsheets/d/"
sheetID2 <- "1WrVA8dPYvQsc_mXVctgTntRLS02qd7ubzcdAsw03Lgk"
urlgics <- paste(prefix2, sheetID2) # Form the URL to connect to
gics <- gsheet2tbl(urlgics) # Read it into a tibble called gics
```

4. Join the data:

```r
# Merging dataframes
sp500 <- merge(sp500Data,
               gics ,
               id = "Stock")
```

5. Rename the data columns:

```r
# Define a mapping of new column names
new_names <- c(
  "Stock", "Date", "StockName", "Sector", "Industry",
  "MarketCap", "Price", "Low52Wk", "High52Wk",
  "ROE", "ROA", "ROIC", "GrossMargin",
  "OperatingMargin", "NetMargin", "PE",
  "PB", "EVEBITDA", "EBITDA", "EPS",
  "EBITDA_YOY", "EBITDA_QYOY", "EPS_YOY",
  "EPS_QYOY", "PFCF", "FCF",
  "FCF_QYOY", "DebtToEquity", "CurrentRatio",
  "QuickRatio", "DividendYield",
  "DividendsPerShare_YOY", "PS",
  "Revenue_YOY", "Revenue_QYOY", "Rating",
  "Security", "GICSSector", "GICSSubIndustry"
)
# Rename the columns using the new_names vector
colnames(sp500) <- new_names
```

6. Create new columns to track prices relative to 52-week low and high.

```
library(dplyr)
sp500 <- sp500 %>%
  mutate(Low52WkPerc = round((Price - Low52Wk) * 100 / Low52Wk, 2),
         High52WkPerc = round((High52Wk - Price) * 100 / High52Wk, 2),
         MarketCapBillions = round(MarketCap / 1e9, 3)
         # Convert MarketCap to billions
         )
```

7. Review the column names:

```
colnames(sp500)
```

```
 [1] "Stock"                "Date"                 "StockName"
 [4] "Sector"               "Industry"             "MarketCap"
 [7] "Price"                "Low52Wk"              "High52Wk"
[10] "ROE"                  "ROA"                  "ROIC"
[13] "GrossMargin"          "OperatingMargin"      "NetMargin"
[16] "PE"                   "PB"                   "EVEBITDA"
[19] "EBITDA"               "EPS"                  "EBITDA_YOY"
[22] "EBITDA_QYOY"          "EPS_YOY"              "EPS_QYOY"
[25] "PFCF"                 "FCF"                  "FCF_QYOY"
[28] "DebtToEquity"         "CurrentRatio"         "QuickRatio"
[31] "DividendYield"        "DividendsPerShare_YOY" "PS"
[34] "Revenue_YOY"          "Revenue_QYOY"         "Rating"
[37] "Security"             "GICSSector"           "GICSSubIndustry"
[40] "Low52WkPerc"          "High52WkPerc"         "MarketCapBillions"
```

8. Format the Prices.

```
library(dplyr)
library(scales)  # For formatting currency

sp500 <- sp500 %>%
  mutate(
    Price = scales::dollar(round(Price, 2)),  # format the Price as $
    High52Wk = scales::dollar(round(High52Wk, 2)),  # format the 52 Week High
    Low52Wk = scales::dollar(round(Low52Wk, 2))  # format the 52 Week Low
  )
```

9. Model the Rating as a `factor()` variable.

```
sp500$Rating <- as.factor(sp500$Rating)
table(sp500$Rating)
```

```

        Buy     Neutral        Sell  Strong Buy Strong Sell 
        192          51         178          58          24 
```

10. Model `GICSSector` as a `factor`.

```
sp500$GICSSector <- as.factor(sp500$GICSSector)
table(sp500$GICSSector)
```

```

Communication Services Consumer Discretionary       Consumer Staples 
                    22                     53                     38 
                Energy             Financials            Health Care 
                    23                     72                     64 
           Industrials Information Technology              Materials 
                    78                     64                     28 
           Real Estate              Utilities 
                    31                     30 
```

18.2 Analysis

Suppose our mission is to allocate `$1 Million` to the "best" stock within our focal sector. Which stock should we select?

18.2.1 The Consumer Staples GICS Sector in the S&P500

We focus on investment opportunities within the Consumer Staples GICS Sector in the S&P500. We want to determine the *fundamentally strongest* AND *most reasonably priced* shares for medium term investing.

For this purpose, we create a tibble named focalStocks, filtering the shares that belong to the Consumer Staples Sector in the S&P500.

```
focalStocks = sp500 %>%
  filter(GICSSector == focalSector)
```

18.2.2 A) Market Cap

We list the stocks. Specifically, the following code processes data from the focalStocks tibble, specifically for stocks in the focal GICS sector. It sorts these stocks by their market capitalization in descending order and selects columns for stock ticker, company name, market cap (in billions), and stock price.

```r
library(dplyr)
library(kableExtra)

# Select stocks with their Market Cap in billions
focalStocks %>%
  arrange(desc(MarketCapBillions)) %>%
  select(Stock, StockName, MarketCapBillions, Price) %>%
  kable("html",
        caption = "GISC Sector Stocks, with Market Cap in Billions") %>%
  kable_styling()
```

Table 18.1: GISC Sector Stocks, with Market Cap in Billions

Stock	StockName	MarketCapBillions	Price
WMT	Walmart Inc.	421.765	$156.74
PG	Procter & Gamble Company (The)	349.243	$148.14
COST	Costco Wholesale Corporation	291.372	$656.65
KO	Coca-Cola Company (The)	258.021	$59.66
PEP	PepsiCo, Inc.	234.827	$170.78
PM	Philip Morris International Inc	148.193	$95.45
MDLZ	Mondelez International, Inc.	99.291	$72.95
MO	Altria Group, Inc.	73.346	$41.46
CL	Colgate-Palmolive Company	65.969	$80.12
TGT	Target Corporation	64.845	$140.34
MNST	Monster Beverage Corporation	60.044	$57.72
EL	Estee Lauder Companies, Inc. (The)	49.247	$137.62
KHC	The Kraft Heinz Company	46.339	$37.78
STZ	Constellation Brands, Inc.	45.487	$248.30
KDP	Keurig Dr Pepper Inc.	44.817	$32.06
KMB	Kimberly-Clark Corporation	41.320	$122.32
KVUE	Kenvue Inc.	40.224	$21.00
HSY	The Hershey Company	38.528	$188.40
SYY	Sysco Corporation	38.009	$75.36
ADM	Archer-Daniels-Midland Company	37.950	$71.15
GIS	General Mills, Inc.	37.174	$65.43

Stock	StockName	MarketCapBillions	Price
KR	Kroger Company (The)	32.888	$45.72
DG	Dollar General Corporation	29.601	$134.86
DLTR	Dollar Tree, Inc.	29.513	$135.35
BF.B	Brown Forman Inc	26.703	$55.24
CHD	Church & Dwight Company, Inc.	23.186	$94.11
WBA	Walgreens Boots Alliance, Inc.	20.541	$23.81
K	Kellanova	19.424	$56.71
MKC	McCormick & Company, Incorporated	18.165	$67.64
CLX	Clorox Company (The)	17.700	$142.52
HRL	Hormel Foods Corporation	17.543	$32.08
TSN	Tyson Foods, Inc.	15.523	$54.21
LW	Lamb Weston Holdings, Inc.	15.400	$106.44
BG	Bunge Limited	14.381	$98.98
SJM	The J.M. Smucker Company	13.652	$128.62
CAG	ConAgra Brands, Inc.	13.577	$28.40
TAP	Molson Coors Beverage Company	13.277	$61.59
CPB	Campbell Soup Company	12.894	$43.24

Consider the summary statistics of the Market Capitalization of the focal stocks.

The following code performs a statistical summary of the market capitalization (in billions) of stocks within a specific GICS sector. It calculates various summary statistics: the count of stocks (`N`), mean, standard deviation (`SD`), median, first quartile (`Q1`), third quartile (`Q3`), minimum (`Min`), maximum (`Max`), and total sum (`Sum`) of market capitalizations. This provides a comprehensive overview of the financial size and spread of the companies in the selected sector.

```
focalStocks %>% summarise(
  N = n(),
  Mean = mean(MarketCapBillions),
  SD = sd(MarketCapBillions),
  Median = median(MarketCapBillions),
  Q1 = quantile(MarketCapBillions, 0.25),
  Q3 = quantile(MarketCapBillions, 0.75),
  Min = min(MarketCapBillions),
  Max = max(MarketCapBillions),
  Sum = sum(MarketCapBillions)
) %>%
  round(2) %>%
  kable("html",
```

```
      caption = "Summary Statistics of Market Cap (Billion USD)") %>%
  kable_styling()
```

Table 18.2: Summary Statistics of Market Cap (Billion USD)

N	Mean	SD	Median	Q1	Q3	Min	Max	Sum
38	74.21	100.2	37.98	18.48	63.64	12.89	421.76	2819.98

As can be seen, the S&P500 GICS sector Consumer Staples consists of 38 stocks. We want to determine the *fundamentally strongest* and *most reasonably priced* shares for medium-term investing.

18.2.3 B) Price Attractiveness – Price relative to 52-Week Low

The 52-week low price is an important metric for stocks on major indices like the S&P 500 for several reasons:

1. Historical Context: It shows the lowest price point of a stock in the past year, providing insights into its price trend.
2. Investment Opportunity: Stocks near their 52-week low may be seen as undervalued, presenting potential buying opportunities.
3. Support Level: This price can act as a psychological support level, indicating market valuation and resistance to further decline.
4. Technical Analysis Tool: For technical analysts, the 52-week low is key for identifying bearish trends or potential recoveries.
5. Dividend Yield Impact: For dividend stocks, a lower price near this level can mean higher yields, appealing to income-focused investors.

- Caution: Our analysis aims to examine stocks' prices in relation to their 52-week lows and highs to gauge their relative price attractiveness. It's crucial to consider the 52-week low alongside other financial indicators, as it alone doesn't necessarily signify a good buy or sell point.

Recall that a new data column named `Low52WkPerc` was created, measuring the percentage change between the current price (`Price`) and its 52-week low (`Low52Wk`), based on the simple formula:

$$Low52WkPerc = \frac{(CurrentPrice - 52WeekLow) * 100}{52WeekLow}$$

Similarly, `High52WkPerc` represents the percentage change between the 52-week high (`High52Wk`) and the current price (`Price`). We round off the data to two decimal places for clarity.

Summary Statistics of Price rel. to 52-Week Low (Low52WkPerc)

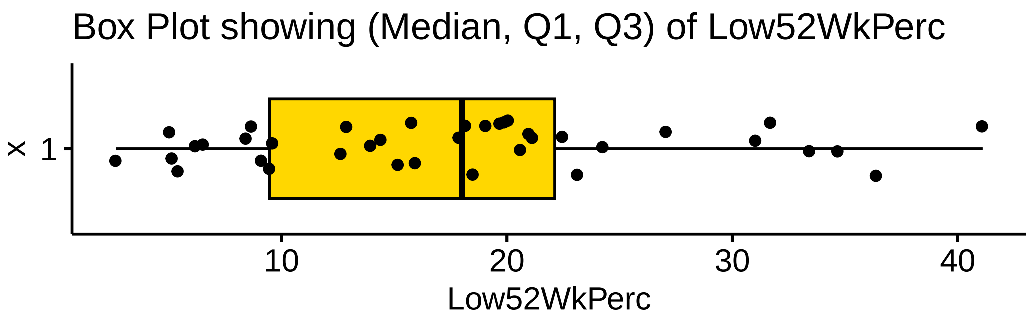
```
summaryStats <- focalStocks %>% summarise(
  N = n(),
  Mean = mean(Low52WkPerc),
  SD = sd(Low52WkPerc),
  Median = median(Low52WkPerc),
  Q1 = quantile(Low52WkPerc, 0.25),
  Q3 = quantile(Low52WkPerc, 0.75),
  Min = min(Low52WkPerc),
  Max = max(Low52WkPerc)
)

Low52WkPercQ1 <- summaryStats$Q1 # Save Q1 of Low52WkPerc

summaryStats %>%
  round(2) %>%
  kable("html",
        caption = "Summary Statistics of Price rel. to 52-Week Low (Low52WkPerc") %>%
  kable_styling()
```

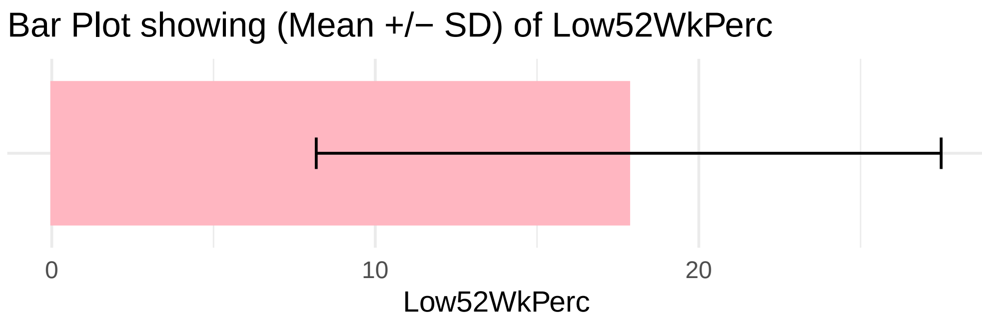
Table 18.3: Summary Statistics of Price rel. to 52-Week Low (Low52WkPerc

N	Mean	SD	Median	Q1	Q3	Min	Max
38	17.83	9.66	18.01	9.46	22.12	2.65	41.11

Visualizing the distribution of Low52WkPerc

The following code creates two visualizations for the `Low52WkPerc` data from the `focalStocks` dataset and combines them into one display. The first is a horizontal box plot (`BoxPlot`) with additional rug and jitter features for detailed data representation, highlighting median, Q1, and Q3 values. The second is a bar plot (`BarPlot`) showing the mean value of `Low52WkPerc` with an error bar representing its standard deviation.





Overall, this code effectively visualizes the distribution and central tendency (mean and standard deviation) of the `Low52WkPerc` metric from the focalStocks dataset. The combination of a box plot and a bar plot with error bars provides a comprehensive view of the data's spread and central measure.

Inexpensive Stocks Close to 52-Week Low (Low52WkPerc < Q1(Low52WkPerc))

The following R code snippet identifies and displays a subset of stocks from the focalStocks dataset that are considered 'inexpensive' based on their performance relative to their 52-week low. It emphasizes those stocks whose `Low52WeekPerc` is less than the 25th percentile (Q1) of the distribution within the focal sector.

```
lowPrice <- focalStocks %>%
  select(Stock, StockName, Price, Low52Wk, Low52WkPerc) %>%
  filter(Low52WkPerc < Low52WkPercQ1) %>%
  arrange(Low52WkPerc)

lowPrice %>%
  kable("html",
        caption = "Inexpensive Stocks (Low52WkPerc < Q1(Low52WkPerc))") %>%
  kable_styling()
```

Table 18.4: Inexpensive Stocks (Low52WkPerc < Q1(Low52WkPerc))

Stock	StockName	Price	Low52Wk	Low52WkPerc
ADM	Archer-Daniels-Midland Company	$71.15	$69.31	2.65
BF.B	Brown Forman Inc	$55.24	$52.59	5.04
KMB	Kimberly-Clark Corporation	$122.32	$116.32	5.16
HSY	The Hershey Company	$188.40	$178.82	5.36
MO	Altria Group, Inc.	$41.46	$39.06	6.13
HRL	Hormel Foods Corporation	$32.08	$30.12	6.51
GIS	General Mills, Inc.	$65.43	$60.33	8.45
KR	Kroger Company (The)	$45.72	$42.09	8.61
PG	Procter & Gamble Company (The)	$148.14	$135.83	9.06
PM	Philip Morris International Inc	$95.45	$87.23	9.42

The filter function is applied to keep only those stocks where `Low52WkPerc` (the percentage difference from the 52-week low) is less than `Low52WkPercQ1` (the first quartile of Low52WkPerc). This implies selecting stocks currently priced closer to their 52-week low than the majority of stocks in the dataset, indicating they are relatively 'inexpensive'.

In essence, this code is used to identify and showcase stocks within the focalSector that are currently trading near their 52-week low, potentially indicating they are undervalued or at an attractive entry point for investors. This analysis assumes that stocks trading below their 52-week low quartile are considered 'inexpensive.'

Next, we analyze the Profitability of stocks within the focal sector, as indicated by metrics such as Return on Equity (ROE).

18.2.4 C) Return on Equity (ROE)

We investigate the stocks in the focal sector that have relatively high Return on Equity (ROE). Return on Equity (ROE) is a crucial financial metric used to assess a company's profitability, specifically measuring how effectively a company uses its equity to generate profits. Here's why ROE is significant in measuring profitability:

1. **Efficiency Indicator**: ROE indicates how well a company is using its shareholders' equity to generate profit. A higher ROE suggests that the company is more efficient in using its equity base to produce earnings.
2. **Investor Attractiveness**: Investors often look at ROE as a sign of a company's ability to generate profit from equity investments. Companies with consistently high ROE are generally seen as well-managed and potentially more attractive to investors.

3. **Comparative Tool**: ROE is used to compare the profitability of companies in the same industry. It levels the playing field by measuring profit relative to shareholder equity, making it a useful tool for comparing companies of different sizes.
4. **Growth Potential**: A strong ROE can indicate a company's potential for growth and its ability to fund expansion without needing additional debt or equity financing. This self-funding capability is especially valuable for investors looking for companies with sustainable growth prospects.
5. **Risk Assessment**: While a high ROE can be desirable, it needs to be considered in the context of risk. A very high ROE might indicate excessive leverage or risk-taking. Thus, ROE should be analyzed alongside other financial metrics like debt-to-equity ratio to understand the risk profile of the company.
6. **Performance Benchmark**: ROE serves as a performance benchmark. It helps in assessing whether a company is improving its efficiency in using equity to generate profits over time.

Summary Statistics of Return on Equity (ROE)

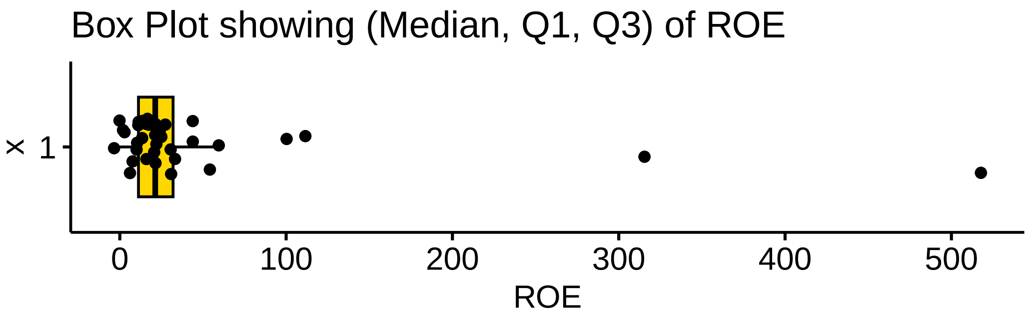
```
summaryStats <- focalStocks %>% summarise(
  N = n(),
  Mean = mean(ROE, na.rm = TRUE),
  SD = sd(ROE, na.rm = TRUE),
  Median = median(ROE, na.rm = TRUE),
  Q1 = quantile(ROE, 0.25, na.rm = TRUE),
  Q3 = quantile(ROE, 0.75, na.rm = TRUE),
  Min = min(ROE, na.rm = TRUE),
  Max = max(ROE, na.rm = TRUE)
)

ROE_Q3 <- summaryStats$Q3

summaryStats %>%
  round(2) %>%
  kable("html",
        caption = "Summary Statistics of Return on Equity (ROE)") %>%
  kable_styling()
```

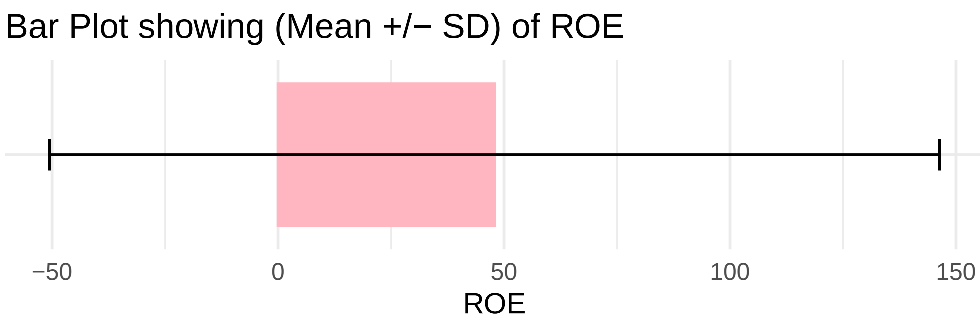
Table 18.5: Summary Statistics of Return on Equity (ROE)

N	Mean	SD	Median	Q1	Q3	Min	Max
38	47.88	98.44	21.32	11.22	32.04	-3.43	517.78

Visualizing the distribution of Return on Equity (ROE)

The following code creates visual representations of the Return on Equity (ROE) for stocks in the focal sector. It includes a BoxPlot that displays the median, first quartile (Q1), and third quartile (Q3), alongside a BarPlot which emphasizes the mean and standard deviation of the ROE. This dual approach offers a comprehensive view of the ROE distribution within the focal sector.





Identifying Stocks that have high ROE (particularly ROE > Q3(ROE))

The following R code identifies and displays a subset of stocks from the focalStocks dataset that have relatively high ROE, filtering stocks whose ROE is more than the 75th percentile (Q3) of the distribution of ROE within the focal sector.

```
highROE <- focalStocks %>%
  select(Stock, StockName, Price, ROE, Low52Wk, Low52WkPerc) %>%
  filter(ROE > ROE_Q3) %>%
  arrange(desc(ROE))

highROE %>%
  kable("html",
        caption = "Stocks with ROE > Q3(ROE)") %>%
  kable_styling()
```

Table 18.6: Stocks with ROE > Q3(ROE)

Stock	StockName	Price	ROE	Low52Wk	Low52WkPerc
CL	Colgate-Palmolive Company	$80.12	517.78140	$67.62	18.49
KMB	Kimberly-Clark Corporation	$122.32	315.48791	$116.32	5.16
SYY	Sysco Corporation	$75.36	111.56247	$62.24	21.09
LW	Lamb Weston Holdings, Inc.	$106.44	100.29901	$81.25	31.00
CLX	Clorox Company (The)	$142.52	59.51557	$114.68	24.27
HSY	The Hershey Company	$188.40	54.15166	$178.82	5.36
PEP	PepsiCo, Inc.	$170.78	43.88217	$155.83	9.59
KO	Coca-Cola Company (The)	$59.66	43.85101	$51.55	15.73
PG	Procter & Gamble Company (The)	$148.14	33.20330	$135.83	9.06

18.2.5 D) Return on Assets (ROA)

Next, we investigate the stocks in the focal sector that have relatively high Return on Assets (ROA).

Return on Assets (ROA) is a key metric for assessing a company's profitability, indicating how efficiently a company uses its assets to generate earnings. Key aspects of its significance include:

1. **Efficiency Indicator**: Higher ROA values signify more effective asset utilization to produce profits.
2. **Comparative Analysis**: ROA is useful for comparing profitability across companies with different sizes and industries, as it reflects income generated from all assets.
3. **Investment Decisions**: Investors use ROA to evaluate management efficiency and make informed investment choices.
4. **Operational Insight**: ROA offers a comprehensive view of a company's operational efficiency by including all assets in its calculation.
5. **Financial Health**: Consistent or improving ROA can signal strengthening financial health and effective operational strategies.
6. **Risk Assessment**: ROA can also indicate potential operational risks, with low values suggesting inefficiencies.
7. **Growth Sustainability**: A stable and robust ROA suggests that a company's growth is underpinned by efficient asset use.

ROA provides a holistic measure of how well a company leverages its total assets to generate profits, making it a crucial tool for financial analysis and decision-making.

Summary Statistics of Return on Assets (ROA)

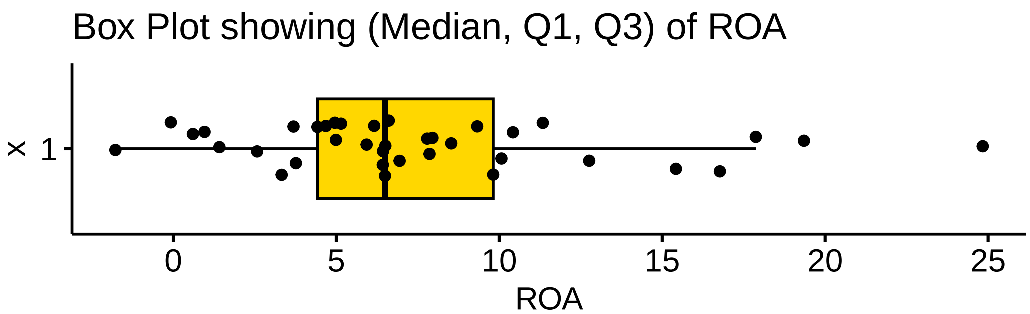
```r
summaryStats <- focalStocks %>% summarise(
  N = n(),
  Mean = mean(ROA, na.rm = TRUE),
  SD = sd(ROA, na.rm = TRUE),
  Median = median(ROA, na.rm = TRUE),
  Q1 = quantile(ROA, 0.25, na.rm = TRUE),
  Q3 = quantile(ROA, 0.75, na.rm = TRUE),
  Min = min(ROA, na.rm = TRUE),
  Max = max(ROA, na.rm = TRUE)
)

ROA_Q3 <- summaryStats$Q3

summaryStats %>%
  round(2) %>%
  kable("html",
        caption = "Summary Statistics of Return on Equity (ROA)") %>%
  kable_styling()
```

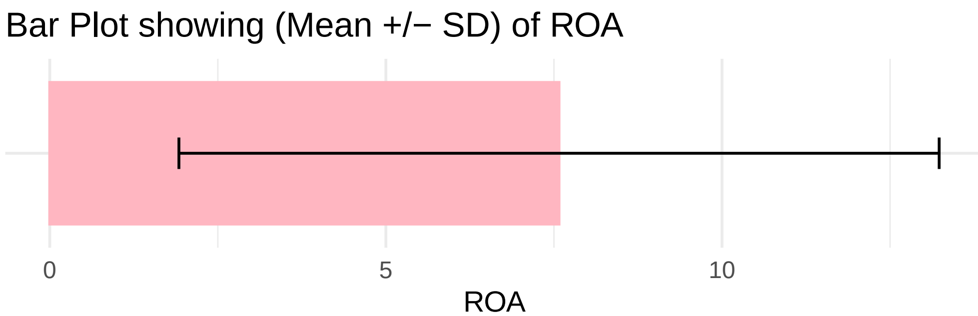
Table 18.7: Summary Statistics of Return on Equity (ROA)

N	Mean	SD	Median	Q1	Q3	Min	Max
38	7.58	5.65	6.49	4.42	9.82	-1.77	24.83

Visualizing the distribution of Return on Assets (ROA)

Similar to the ROE analysis, the following code creates visual representations of the Return on Assets (ROA) for stocks in the focal sector. It includes a BoxPlot that displays the median, first quartile (Q1), and third quartile (Q3), alongside a BarPlot which emphasizes the mean and standard deviation of the ROA. This dual approach offers a comprehensive view of the ROA distribution within the focal sector.





Identifying Stocks that have high ROA (particularly ROA > Q3(ROA))

Similar to the ROE analysis above, the following R code identifies and displays a subset of stocks from the focalStocks dataset that have relatively high ROA, filtering stocks whose ROA is more than the 75th percentile (Q3) of the distribution of ROA within the focal sector.

```
highROA <- focalStocks %>%
  select(Stock, StockName, Price, ROA, Low52Wk, Low52WkPerc) %>%
  filter(ROA > ROA_Q3) %>%
  arrange(desc(ROA))

highROA %>%
  kable("html",
        caption = "Stocks with ROA > Q3(ROA)") %>%
  kable_styling()
```

Table 18.8: Stocks with ROA > Q3(ROA)

Stock	StockName	Price	ROA	Low52Wk	Low52WkPerc
MO	Altria Group, Inc.	$41.46	24.83315	$39.06	6.13
LW	Lamb Weston Holdings, Inc.	$106.44	19.34928	$81.25	31.00
MNST	Monster Beverage Corporation	$57.72	17.87396	$47.13	22.47
HSY	The Hershey Company	$188.40	16.76908	$178.82	5.36
PM	Philip Morris International Inc	$95.45	15.42009	$87.23	9.42

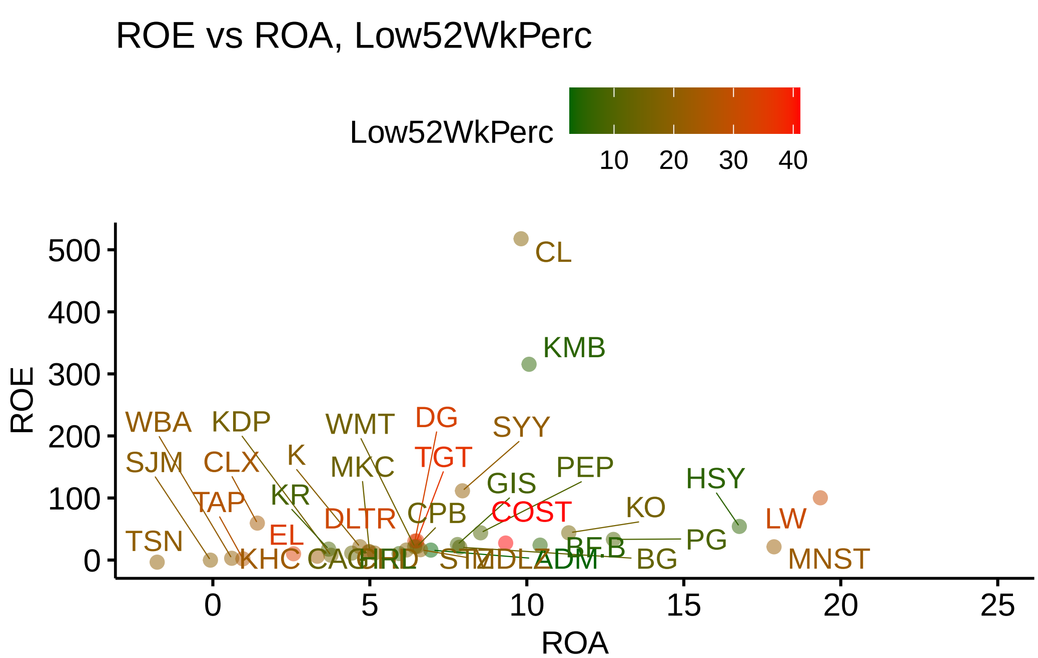
Stock	StockName	Price	ROA	Low52Wk	Low52WkPerc
PG	Procter & Gamble Company (The)	$148.14	12.75894	$135.83	9.06
KO	Coca-Cola Company (The)	$59.66	11.33602	$51.55	15.73
BF.B	Brown Forman Inc	$55.24	10.42293	$52.59	5.04
KMB	Kimberly-Clark Corporation	$122.32	10.07174	$116.32	5.16

18.2.6 E) ROE versus ROA and colored by Price rel. to 52-Week Low

Next, we visualize the joint distribution of ROE, ROA and Price rel. to 52-Week Low.

The following code creates a scatter plot that visualizes the relationship between ROA and ROE in the **focalStocks** dataset. It uses color coding based on the **Low52WkPerc** metric to provide further insights into the stocks' performance relative to their 52-week lows.

```
ggscatter(focalStocks,
          x = "ROA",
          y = "ROE",
          color = "Low52WkPerc",
          alpha = 0.5,
          label = "Stock",
          repel = TRUE,
          title = "ROE vs ROA, Low52WkPerc") +
  gradient_color(c("darkgreen",  "red"))
```



The objective of the provided code is to create a visual analysis of the relationship between Return on Assets (ROA) and Return on Equity (ROE) for stocks within a specific sector (focal sector). The scatter plot, enhanced with color-coding based on each stock's percentage difference from its 52-week low (`Low52WkPerc`), aims to reveal how these key profitability metrics interact and are influenced by the stock's relative position to its 52-week low. By incorporating stock labels and a color gradient, the plot not only maps out the financial performance (ROA and ROE) of these stocks but also integrates their recent price history, offering a multifaceted view of the sector's financial health and market perception.

18.2.7 F) Investment Insights

Let us review our analysis and generate some insights.

1. The following stocks within the focal sector have prices relatively close to their 52-Week low, within Q1 of the distribution of `Low52WkPerc`:

```
lowPrice$Stock
```

```
 [1] "ADM"  "BF.B" "KMB"  "HSY"  "MO"   "HRL"  "GIS"  "KR"   "PG"   "PM"
```

2. The following stocks within the focal sector are highly profitable, having Return on Equity more than the 75th percentile (Q3) of the distribution of the focal sector:

```
highROE$Stock
```

```
[1] "CL"  "KMB" "SYY" "LW"  "CLX" "HSY" "PEP" "KO"  "PG"
```

3. The following stocks within the focal sector are highly profitable, having Return on Assets more than the 75th percentile (Q3) of the distribution of the focal sector:

```
highROA$Stock
```

```
[1] "MO"   "LW"   "MNST" "HSY"  "PM"   "PG"   "KO"   "BF.B" "KMB"
```

4. Let us find the intersection between these three sets of stocks

```
# Find the intersection (common stocks) between the three sets
common_stocks <- intersect(intersect(highROA$Stock, highROE$Stock), lowPrice$Stock)

# View the result
common_stocks
```

```
[1] "HSY" "PG"  "KMB"
```

The following are the stock(s) that are most reasonably priced AND have relatively high Return on Equity and Return on Assets: HSY, PG, KMB

```r
library(dplyr)

# Filter and select specific columns from focalStocks where Stock is in common_stocks
filtered_stocks <- focalStocks %>%
  filter(Stock %in% common_stocks) %>%
  select(Stock, StockName, ROE, ROA, Price)

# View the result
filtered_stocks %>%
  kable("html",
        caption = "Most Resonably Priced and Highly Profitable Stocks") %>%
  kable_styling()
```

Table 18.9: Most Resonably Priced and Highly Profitable Stocks

Stock	StockName	ROE	ROA	Price
HSY	The Hershey Company	54.15166	16.76908	$188.40
KMB	Kimberly-Clark Corporation	315.48791	10.07174	$122.32
PG	Procter & Gamble Company (The)	33.20330	12.75894	$148.14

Conclusion: The best investment for USD 1 million, is in these stocks that are both reasonably priced and highly profitable.

18.3 Summary of Chapter 18 – Case (2 of 2): S&P500 Sector Analysis

Last updated on January 07, 2024, this chapter presents a Case Study of a detailed exploration of the Consumer Staples sector within the S&P 500. This analysis utilizes a range of data manipulation and visualization tools in R, leveraging libraries such as `dplyr`, `tibble`, `ggplot2`, `gsheet`, and `kableExtra`. The study begins by meticulously loading and structuring data from Google Sheets into tibbles. This data is then merged, renamed, and enhanced with new metrics for a more insightful analysis.

The study delves into several key areas of analysis for the focal sector. It starts with an examination of market capitalization, sorting stocks by their market cap and generating summary statistics to understand the financial magnitude of these companies. The significance of the 52-week low price is then explored, identifying stocks that are trading near these levels and potentially undervalued. The analysis further extends to profitability measures, particularly Return on Equity (ROE) and Return on Assets (ROA). Through visualizations and data filtering, the chapter highlights stocks that demonstrate high ROE and ROA, signaling their profitability.

An intersection analysis is conducted to identify stocks that are not only close to their 52-week low but also exhibit high profitability. This comprehensive approach pinpoints the most promising investment opportunities in the sector. The study utilizes scatter plots to illustrate the relationships between ROE, ROA, and the percentage change from 52-week lows, providing deeper insights into stock performance and price attractiveness.

In conclusion, the anlaysis identifies the 'best' investment options by filtering out stocks that meet both criteria of price attractiveness and high profitability. This rigorous analysis, combining both quantitative measures and visual insights, offers a nuanced perspective on the focal sector, guiding investment decisions based on a blend of market performance and financial metrics.

18.4 References

S&P 500

[1] https://www.investopedia.com/terms/s/sp500.asp

[2] S&P Global: S&P Global. (n.d.). S&P 500. Retrieved September 14, 2023, from https://www.spglobal.com/spdji/en/indices/equity/sp-500/

MarketWatch: MarketWatch. (n.d.). S&P 500 Index. Retrieved September 14, 2023, from https://www.marketwatch.com/investing/index/spx

Bloomberg: Bloomberg. (n.d.). S&P 500 Index (SPX:IND). Retrieved September 14, 2023, from https://www.bloomberg.com/quote/SPX:IND

[3] TradingView.com https://www.tradingview.com/screener/

[4] GICS: Global Industry Classification Standard: https://www.spglobal.com/spdji/en/landing/topic/gics/