DATA ANALYTICS 101

EXPLORATARY DATA ANALYSIS

Using

R

Sameer Mathur
Aryeman G. Mathur

Data Analytics 101 – Exploratory Data Analysis using R programming.

Sameer Mathur, Aryeman Gupta Mathur

2023-07-25

Table of contents

Welcome

July 25, 2023

Exploratory Data Analysis (EDA) is an approach to analyzing data sets to summarize their main characteristics, often using statistical graphics and other data visualization methods. EDA is primarily for seeing what the data can tell us beyond the formal modeling or hypothesis testing tasks.

The EDA approach can be broken down into the following steps:

Data Cleaning: This step includes handling missing data, removing outliers, and other data cleansing processes.

Univariate Analysis: Here, each field in the dataset is analyzed independently to better understand its distribution, outliers, and unique values. This could involve statistical plots for measuring central tendency like mean, median, mode, frequency distribution, quartiles, etc.

Bivariate Analysis: This step involves the analysis of two variables to determine the empirical relationship between them. It includes techniques such as scatter plots for continuous variables or crosstabs for categorical data.

Multivariate Analysis: This is an advanced step, involving analysis with more than two variables. It helps to understand the interactions between different fields in the dataset.

Data Visualization: This is the creation of plots such as histograms, box plots, scatter plots, etc., to identify patterns, relationships, or outliers within the dataset. This can be done using visualization tools or libraries.

Insight Generation: After visualizations and some statistical tests, analysts will generate insights that could lead to further questions, hypotheses, and model building.

The EDA process is an important precursor to more complex analyses because it allows us to confirm or invalidate some initial hypotheses and to formulate a more precise question or hypothesis that can lead to further statistical analysis and testing.

Our focus

- We ignore the Data Cleaning step, although we acknowledge it's practical relevance. We assume that we are working with a clean dataset.
- We emphasize Exploratory Univariate and Bivariate Analysis of data and the corresponding Data Visualization.

We illustrate all of the above using the R programming language.

We further illustrate how to use R programming in the form of a live project implemented on a real-world dataset. Our dataset concerns the S&P500 stocks. This will demonstrate a practical aspect of using this book. We have many sample codes regarding this, using real-world data.. We will explore financial metrics such as Return on Equity, Return on Assets and Return on Invested Capital of S&P500 shares.

1 Getting Started

July 25, 2023

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 your 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 your operating system (Windows, Mac, or Linux).
3. After choosing your 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 your 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 your local machine and looking to take advantage of IDE features, you 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, you can enhance your 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 you 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 you to choose the one that aligns best with your 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 your productivity and enhance your 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 your computer, you can follow these simple steps:

1. **Download RStudio**: Visit the RStudio download page and choose the version of RStudio that matches your operating system.
2. **Install RStudio**: Once the RStudio installer is downloaded, run it and follow the instructions provided to complete the installation process on your computer.
3. **Open RStudio:** After the installation is finished, you can open RStudio by double-clicking the RStudio icon on your desktop or in the Applications folder.
4. **Start an R session**: In RStudio, click on the Console tab to initiate an R session. You 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, you 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 sales by 2 2 * sales}`

```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 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. 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. 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

Wickham, H. (2014). Tidy data. Journal of Statistical Software, 59(10), 1-23.

Wickham, H. (2016). ggplot2: Elegant graphics for data analysis. Springer-Verlag.

Wickham, H., & Grolemund, G. (2017). R packages: Organize, test, document, and share your code. O'Reilly Media.

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

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

The R Project for Statistical Computing. (2021). Download R for Linux. 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./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./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. https://www.rstudio.com/

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

[5] RStudio. (2021). RStudio. 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. https://doi.org/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. https://doi.org/10.1016/j.jnca.2013.11.004

Cloud Spectator. (2021). Cloud Service Provider Pricing Models: A Comprehensive Guide. https://www.cloudspectator.com/cloud-service-provider-pricing-models-a-comprehensive-guide/

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

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

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

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

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

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

Microsoft Azure. (2021). Create a Windows virtual machine with the Azure portal. https://docs.microsoft.com/en-us/azure/virtual-machines/windows/quick-create-portal

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

2 R Packages

July 25, 2023

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. R packages 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, you would execute the following code:

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

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

```r
library(ggplot2)
```

6. By executing the provided code, you will enable access to the functions and datasets of the `ggplot2` package for use within your 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. `plyr`: Offers a set of functions for splitting, applying a function, and combining results, allowing for efficient data manipulation and summarization.
4. `reshape2`: Provides functions for transforming data between different formats, such as converting data from wide to long format and vice versa.
5. `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.
6. `lubridate`: Designed specifically for working with dates and times, it simplifies common tasks like parsing, manipulating, and formatting date-time data.
7. `stringr`: Offers a consistent and intuitive set of functions for working with strings, including pattern matching, string manipulation, and string extraction.
8. `magrittr`: Provides a simple and readable syntax for composing data manipulation and transformation operations, making code more readable and expressive.
9. `ggplot2`: A powerful and flexible package for creating beautiful and customizable data visualizations using a layered grammar of graphics approach.
10. `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()
```

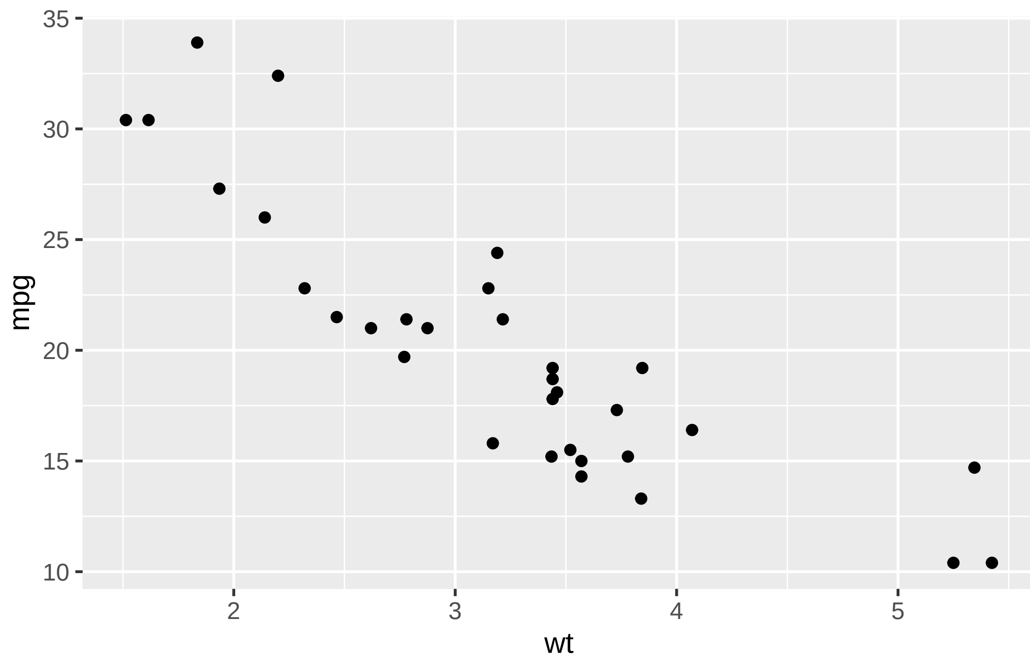


Figure 2.1: Scatterplot of Car Mileage with Car Weight

2.4.1 Getting help

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

1. **Documentation**: Most R packages include comprehensive documentation that covers functions, datasets, and usage examples. To access the documentation, you can use the `help()` function or type `?package_name` directly in the R console, replacing `package_name` with the specific package you 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, you can use the commands `help(topic)` or `?topic` in the R console, where `topic` represents the name of the function or package you 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 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. (2011). The Split-Apply-Combine Strategy for Data Analysis. Journal of Statistical Software, 40(1), 1-29.

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**

Bache, S. M., & Wickham, H. (2014). magrittr: A Forward-Pipe Operator for R. R package version 2.0.1. Retrieved from **https://CRAN.R-project.org/package=magrittr**

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

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

RStudio Team. (2020). RStudio: Integrated Development Environment for R. Retrieved from **https://www.rstudio.com/**

3 Data Structures

July 25, 2023

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 we believe that these are the three most useful data structures. [1]

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. [1]

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
```

```r
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
```

In the above code, we perform addition, subtraction, multiplication, division, and exponentiation on these vectors using the arithmetic operators +, -, *, /, and ^ respectively.

In addition to the common arithmetic operations (addition, subtraction, multiplication, division, and exponentiation), R also supports other arithmetic operations such as modulus, integer division, and absolute value. Let's demonstrate these operations

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

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

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

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

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

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

Let's explore a few additional arithmetic operations:

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

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

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

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

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

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

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

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

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

```
[1] 209
```

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

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

`log` calculates the natural logarithm of each element.

`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.

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

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

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

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

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

```r
print(female_and_income)
```

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

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

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

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

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

In the above code, we perform the following logical operations:

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 (!=): 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.

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 (|): 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 (!): 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.

```r
# Greater than comparison
age_greater_50 <- ages > 50
print(age_greater_50)
```

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

```r
# Less than or equal to comparison
weight_less_equal_54 <- weights <= 54
print(weight_less_equal_54)
```

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

```r
# Element-wise AND
age_and_weight_greater_50 <- (ages > 50) & (weights > 50)
print(age_and_weight_greater_50)
```

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

```r
# Element-wise OR
age_or_weight_less_equal_50 <- (ages > 50) | (weights <= 50)
print(age_or_weight_less_equal_50)
```

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

```r
# Element-wise XOR
age_xor_weight_greater_50 <- xor(ages > 50, weights > 50)
print(age_xor_weight_greater_50)
```

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

```r
# Any True
any_female <- any(females)
print(any_female)
```

```
[1] TRUE
```

```r
# All True
all_female <- all(females)
print(all_female)
```

```
[1] FALSE
```

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 (<=): 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.

Element-wise AND (&): It performs an element-wise logical AND operation between the conditions (ages > 50) and (weights > 50). The resulting vector, age_and_weight_greater_50, contains TRUE for elements that satisfy both conditions and FALSE otherwise.

Element-wise OR (|): It performs an element-wise logical OR operation between the conditions (ages > 50) and (weights <= 50). The resulting vector, age_or_weight_less_equal_50, contains TRUE for elements that satisfy either condition or both.

Element-wise XOR (xor()): It performs an element-wise exclusive OR operation between the conditions (ages > 50) and (weights > 50). The resulting vector, age_xor_weight_greater_50, contains TRUE for elements where exactly one condition is true and FALSE otherwise.

Any True (any()): It checks if at least one element in the females vector is TRUE. The result, any_female, is TRUE if there is at least one TRUE value in the vector and FALSE otherwise.

All True (all()): It checks if all elements in the females vector are TRUE. The result, all_female, is TRUE if all values in the vector are TRUE and FALSE otherwise.

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

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

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

```
[1] TRUE
```

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

```
[1] FALSE
```

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

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

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

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

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

```
[1] FALSE
```

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

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

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

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

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()): 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()): 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: 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: 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(): 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(): 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.

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:

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

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

```r
# 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.

```r
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.

```r
min(ages)
```

```
[1] 42
```

```r
max(ages)
```

```
[1] 72
```

```r
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. Additional Functionality:

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

```
[1] 6.737643
```

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

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

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

```
[1] -0.9770974
```

Standard Error of the Mean: It calculates the standard error of the mean for the ages vector. The result is stored in se_ages.

Cumulative Sum: It calculates the cumulative sum of the elements in the ages vector. The cumulative sum is stored in cumulative_sum_ages.

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

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.

3.1.4 Strings

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

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:

```r
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:

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

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

```
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:

```
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()` and `grepl()` functions, 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 use:

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

```
[1] 3 4
```

This returns a vector containing the indexes of the "names" vector elements that contain the letter "a."

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

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

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

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

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

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

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

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

%in% Operator: It checks if each element in the names vector is present in the specified set of names. The resulting vector, names_found, contains TRUE for elements that are found in the set and FALSE otherwise.

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.

3.2 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

Wickham, H., & Grolemund, G. (2016). R for data science: Import, tidy, transform, visualize, and model data. O'Reilly Media, Inc.

4 Reading Data

July 24, 2023

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. Using the `data.frame()` function or by importing data from external sources such as CSV files, Excel spreadsheets, or databases, dataframe objects can be created in R.
4. dataframe objects have many useful built-in methods and functions for manipulating and summarizing data, including subsetting, merging, filtering, and aggregation. [1]

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)

# Print the resulting dataframe
```

```r
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. 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. 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.
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. [2]

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.

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.

```
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.
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"

# 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.

4.3.5 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`.

4.4 Tibbles

1. A tibble is a contemporary and enhanced variant of a R data frame that is part of the tidyverse package collection.
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, with the use of [[always returning a vector or NULL, and [] always returning a tibble.
6. Here is an example of using the tibble() function in dplyr to construct a tibble:

```
library(dplyr, warn.conflicts = FALSE)

# Create a tibble
my_tibble <- tibble(
  name = c("Alice", "Bob", "Charlie"),
  age = c(25, 30, 35),
  gender = c("F", "M", "M")
)

# Print the tibble
my_tibble
```

```
# A tibble: 3 x 3
  name      age gender
  <chr>   <dbl> <chr>
1 Alice      25 F
2 Bob        30 M
3 Charlie    35 M
```

7. This will generate 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

```r
# Create a data frame
my_df <- data.frame(
  name = c("Alice", "Bob", "Charlie"),
  age = c(25, 30, 35),
  gender = c("F", "M", "M")
)

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

# Print the tibble
my_tibble
```

```
# A tibble: 3 x 3
  name      age gender
  <chr>   <dbl> <chr>
1 Alice      25 F
2 Bob        30 M
3 Charlie    35 M
```

8. This assigns the tibble representation of the data frame my_df to the variable my_tibble.
9. 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.

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   Alice  25      F
2     Bob  30      M
3 Charlie  35      M
```

10. 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: With [[always returning a vector or NULL and [] always returning a tibble, 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 provides more informative error messages that make it simpler to diagnose and resolve data-related problems.
- Fewer surprises: Tibbles have more stringent constraints than data frames, resulting in fewer surprises and unexpected behavior when manipulating data.

4.5 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

Wickham, H., & Grolemund, G. (2016). R for data science: Import, tidy, transform, visualize, and model data. O'Reilly Media, Inc.

R Core Team. (2022, March 2). Data Frames. R Documentation. https://www.rdocumentation.org/packages/ba

[2]

OpenIntro. (2022). 1.3 RStudio and working directory. In Introductory Statistics with Randomization and Simulation (1st ed.). https://www.openintro.org/book/isrs/

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. https://stat.ethz.ch/R-manual/R-devel/library/base/html/getwd.html

5 Exploring Dataframes

July 25, 2023

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.

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

```
data(mtcars)
```

5.1 Reviewing a dataframe

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

```
View(mtcars)
```

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
```

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
```

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

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

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

```r
dim(mtcars)
```

```
[1] 32 11
```

```r
nrow(mtcars)
```

```
[1] 32
```

```r
ncol(mtcars)
```

```
[1] 11
```

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

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

```r
names(mtcars)
```

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

```r
colnames(mtcars)
```

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

5.2 Accessing data within a dataframe

$: In R, the dollar sign $ is a unique operator that lets us retrieve specific columns from a dataframe or elements from a list.

For instance, consider the dataframe mtcars. If we wish to fetch the data from the mpg (miles per gallon) column, we would use mtcars$mpg. This action will yield a vector containing the data from the mpg column.

```r
# 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
```

This operator offers a simple and readable shortcut for accessing data.

[[: The usage of $ is limited since it doesn't support character substitution for dynamic column access inside functions. In such cases, we resort to using double square brackets [[or single square brackets [.

As an example, if we have a character string stored in a variable var as var <- "mpg", using mtcars$var will not return the mpg column. But if we use mtcars[[var]] or mtcars[, var], we will correctly get the mpg column.

```r
# 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

str(): This function displays the internal structure of an R object.

```
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 ...
```

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.

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

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

```
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.

```
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.

```
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

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."

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.

To change the data type of the am, cyl, vs, and gear variables in the mtcars dataset to factors, you 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 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 ...
```

When the cyl variable in the mtcars dataset 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), you will receive an output that reveals 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:

```
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.

Utilizing the levels() function on factor variables in R allows you to examine the particular categories or levels present within a factor, aiding in understanding the data's composition and facilitating operations that target specific levels if necessary.

To change the base level of a factor variable in R, you can use the relevel() function. This function allows you to reassign a new base level by rearranging the order of the levels in the factor variable.

Here's an example of how you can change the base level of a factor variable:

```
# 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.

Changing the base level can be particularly useful when conducting statistical modeling or interpreting the effects of categorical variables in regression models. By selecting a specific level as the base, we can compare the effects of the other levels relative to the chosen base level, facilitating more meaningful analysis and interpretation.

For convenience, we will change the base level back to "4".

```
# 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.

```
# Assuming 'cyl' is a factor variable with levels "4", "6", and "8"
# Check the levels of 'cyl' before removing unused levels
levels(mtcars$cyl)
```

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

```
# 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 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(): The cut() function allows you to convert a continuous variable into a factor variable by dividing it into intervals or bins. This is useful when you want to group numeric data into categories or levels.

```
# 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. 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.

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
```

```
# 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.

```
# 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.

```
# 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.

```
# 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.

```
# 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.

```
# 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
```

```
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.

```
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
```

```
mtcars[order(-mtcars$mpg), ] # descending order
```

```
                     mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Toyota Corolla      33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
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
Lotus Europa        30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
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
Merc 240D           24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
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
Toyota Corona       21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
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
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
Ferrari Dino        19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
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
Hornet Sportabout   18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
```

```
Valiant             18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
Merc 280C           17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
Merc 450SL          17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
Merc 450SE          16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
Ford Pantera L      15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
Dodge Challenger    15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
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
Maserati Bora       15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
Duster 360          14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
Camaro Z28          13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
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
```

5.6 Statistical functions

mean(): This function computes the arithmetic mean.

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

```
[1] 20.09062
```

median(): This function computes the median.

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

```
[1] 19.2
```

sd(): This function computes the standard deviation.

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

```
[1] 6.026948
```

var(): This function computes the variance.

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

```
[1] 36.3241
```

cor(): This function computes the correlation between variables.

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

```
[1] -0.8676594
```

unique(): This function extracts the unique elements of a vector.

```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

summary(): This function is a convenient tool to generate basic descriptive statistics for your dataset. It provides a succinct snapshot of the distribution characteristics of your data.

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

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

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.

```
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.

```
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:

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

  # Return the computed average
  return(average_val)
}

# Utilize the created function
average_mpg <- compute_average(mtcars, "mpg")
print(average_mpg)
```

```
[1] 20.09062
```

```
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) column. 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.

This demonstrates a simple way to create a custom function in R.

Function to calculate average mileage for cars with a specific number of cylinders:

```
avg_mileage_by_cyl <- function(data, cyl) {
  mean(data$mpg[data$cyl == cyl])
}
```

```r
# Usage

# Returns the average mileage of cars with 4 cylinders
avg_mileage_by_cyl(mtcars, 4)
```

```
[1] 26.66364
```

```r
# Returns the average mileage of cars with 6 cylinders
avg_mileage_by_cyl(mtcars, 6)
```

```
[1] 19.74286
```

6 Live Case: S&P500 (1 of 3)

July 25, 2023

6.1 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. https://www.investopedia.com/terms/s/sp500.asp

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.

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.

Among these 503 stocks, Apple, the technology giant, holds the top position with a market capitalization of $2.35 billion. Following Apple, Microsoft and Amazon.com rank as the second and third largest stocks in the S&P 500, respectively. The next positions are held by Nvidia Corp, Tesla, Berkshire Hathaway, and two classes of shares from Google's parent company, Alphabet..

6.2 S&P 500 Data - Preliminary Analysis

We will analyze a real-world, recent dataset containing information about the S&P500 stocks. The dataset is located in a Google Sheet

The data is disorganized and challenging to understand. We will review the data and proceed in a step-by-step manner.

6.2.1 Read the S&P500 data from a Google Sheet into a tibble dataframe.

1. The complete URL is https://docs.google.com/spreadsheets/d/11ahk9uWxBkDqrhNm7qYmiTwrlSC53N1zvXYfv7ttOCM/
2. The Google Sheet ID is: `11ahk9uWxBkDqrhNm7qYmiTwrlSC53N1zvXYfv7ttOCM`. We can use the function `gsheet2tbl` in package `gsheet` to read the Google Sheet into a tibble or 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"
url500 <- paste(prefix,sheetID) # Form the URL to connect to
sp500 <- gsheet2tbl(url500) # Read it into a tibble called sp500
```

6.3 Review the data

1. We want to understand the different data columns and their data structure. For this purpose, we run the `str()` function.

```
str(sp500)
```

```
spc_tbl_ [503 x 36] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
 $ Date                                    : chr [1:503] "7/25/2023" "7/25/2023" "7/25/2023" "
 $ Stock                                   : chr [1:503] "A" "AAL" "AAP" "AAPL" ...
 $ Description                             : chr [1:503] "Agilent Technologies, Inc." "Americ
 $ Sector                                  : chr [1:503] "Health Technology" "Transportation"
 $ Industry                                : chr [1:503] "Medical Specialties" "Airlines" "Sp
 $ Market Capitalization                   : num [1:503] 3.73e+10 1.14e+10 4.20e+09 3.04e+12 2
 $ Price                                   : num [1:503] 126.4 17.5 70.7 193 143.6 ...
```

```
$ 52 Week Low                             : num [1:503] 113.3 11.7 63.6 124.2 131 ...
$ 52 Week High                            : num [1:503] 160 19.1 212 198 168 195 116 81.9 328
$ Return on Equity (TTM)                  : num [1:503] 24.8 NA 14.6 146 51.1 389 NA 14.8 30
$ Return on Assets (TTM)                  : num [1:503] 12.7 3.9 3.35 27.6 5.43 2.79 NA 4.98
$ Return on Invested Capital (TTM)        : num [1:503] 16.51 8.01 6.17 57.18 9.9 ...
$ Gross Margin (TTM)                      : num [1:503] 54.1 23.8 43.8 43.2 72.2 ...
$ Operating Margin (TTM)                  : num [1:503] 23.78 9.39 5.63 29.16 41.07 ...
$ Net Margin (TTM)                        : num [1:503] 19.19 4.98 3.61 24.49 13.3 ...
$ Price to Earnings Ratio (TTM)           : num [1:503] 27.86 4.78 10.5 32.8 33.82 ...
$ Price to Book (FY)                      : num [1:503] 7.04 NA 1.56 60.73 14.73 ...
$ Enterprise Value/EBITDA (TTM)           : num [1:503] 19.5 5.71 8.8 25 9.64 12.9 NA NA 17.
$ EBITDA (TTM)                            : num [1:503] 1.97e+09 7.16e+09 9.21e+08 1.24e+11 
$ EPS Diluted (TTM)                       : num [1:503] 4.54 3.67 6.72 5.89 4.25 ...
$ EBITDA (TTM YoY Growth)                 : num [1:503] 10.52 1074.1 -16 -5.36 10.6 ...
$ EBITDA (Quarterly YoY Growth)           : num [1:503] 8.2 72.2 -39.01 -4.58 11.68 ...
$ EPS Diluted (TTM YoY Growth)            : num [1:503] 9.17 NA -25.21 -4.33 -39.11 ...
$ EPS Diluted (Quarterly YoY Growth)      : num [1:503] 11.69944 154.13308 -68.36829 -0.0065
$ Price to Free Cash Flow (TTM)           : num [1:503] 31.74 7.88 NA 31.38 10.84 ...
$ Free Cash Flow (TTM YoY Growth)         : num [1:503] 11.81 NA -100.23 -7.85 6.68 ...
$ Free Cash Flow (Quarterly YoY Growth)   : num [1:503] 55.7078 -10.2542 -176.1352 -0.0312 -
$ Debt to Equity Ratio (MRQ)              : num [1:503] 0.473 NA 1.582 1.763 4.678 ...
$ Current Ratio (MRQ)                     : num [1:503] 2.37 0.749 1.244 0.94 0.96 ...
$ Quick Ratio (MRQ)                       : num [1:503] 1.708 0.656 0.238 0.878 0.821 ...
$ Dividend Yield Forward                  : num [1:503] 0.723 NA 1.428 0.497 4.163 ...
$ Dividends per share (Annual YoY Growth): num [1:503] 8.25 NA 84.62 5.88 7.53 ...
$ Price to Sales (FY)                     : num [1:503] 5.538 0.235 0.384 7.992 4.399 ...
$ Revenue (TTM YoY Growth)                : num [1:503] 7.8597 29.9089 1.4153 -0.2544 0.0282
$ Revenue (Quarterly YoY Growth)          : num [1:503] 6.85 4.72 1.29 -2.51 -9.7 ...
$ Technical Rating                        : chr [1:503] "Sell" "Buy" "Buy" "Sell" ...
- attr(*, "spec")=
 .. cols(
 ..   Date = col_character(),
 ..   Stock = col_character(),
 ..   Description = col_character(),
 ..   Sector = col_character(),
 ..   Industry = col_character(),
 ..   `Market Capitalization` = col_double(),
 ..   Price = col_double(),
 ..   `52 Week Low` = col_double(),
 ..   `52 Week High` = col_double(),
 ..   `Return on Equity (TTM)` = col_double(),
 ..   `Return on Assets (TTM)` = col_double(),
 ..   `Return on Invested Capital (TTM)` = col_double(),
```

```
..   `Gross Margin (TTM)` = col_double(),
..   `Operating Margin (TTM)` = col_double(),
..   `Net Margin (TTM)` = col_double(),
..   `Price to Earnings Ratio (TTM)` = col_double(),
..   `Price to Book (FY)` = col_double(),
..   `Enterprise Value/EBITDA (TTM)` = col_double(),
..   `EBITDA (TTM)` = col_double(),
..   `EPS Diluted (TTM)` = col_double(),
..   `EBITDA (TTM YoY Growth)` = col_double(),
..   `EBITDA (Quarterly YoY Growth)` = col_double(),
..   `EPS Diluted (TTM YoY Growth)` = col_double(),
..   `EPS Diluted (Quarterly YoY Growth)` = col_double(),
..   `Price to Free Cash Flow (TTM)` = col_double(),
..   `Free Cash Flow (TTM YoY Growth)` = col_double(),
..   `Free Cash Flow (Quarterly YoY Growth)` = col_double(),
..   `Debt to Equity Ratio (MRQ)` = col_double(),
..   `Current Ratio (MRQ)` = col_double(),
..   `Quick Ratio (MRQ)` = col_double(),
..   `Dividend Yield Forward` = col_double(),
..   `Dividends per share (Annual YoY Growth)` = col_double(),
..   `Price to Sales (FY)` = col_double(),
..   `Revenue (TTM YoY Growth)` = col_double(),
..   `Revenue (Quarterly YoY Growth)` = col_double(),
..   `Technical Rating` = col_character()
.. )
- attr(*, "problems")=<externalptr>
```

2. The `str(sp500)` output provides valuable insights into the structure and data types of the columns in the `sp500` tibble. Let's delve into the details.

3. The output reveals that `sp500` is a tibble with dimensions $[503 \times 36]$. This means it consists of 503 rows, each representing a specific S&P500 stock, and 36 columns containing information about each stock.

4. Here is a preliminary breakdown of the information associated with each column:

- The columns labeled `Date`, `Stock`, `Description`, `Sector`, and `Industry` are character columns. They respectively represent the date, stock ticker symbol, description, sector, and industry of each S&P500 stock.

- Columns such as `Market.Capitalization`, `Price`, `X52.Week.Low`, `X52.Week.High`, and other numeric columns contain diverse financial metrics and stock prices related to the S&P500 stocks.

- The column labeled `Technical.Rating` is a character column that assigns a technical rating to each stock.

5. By examining the `str(sp500)` output, we gain a preliminary understanding of the data types and column names present in the `sp500` tibble, enabling us to grasp the structure of the dataset.

6.3.1 Rename Data Columns

1. The names of the data columns are lengthy and confusing.
2. We will rename the data columns to make it easier to work with the data, using the `rename_with()` function.

```
# Define a mapping of new column names
new_names <- c(
  "Date", "Stock", "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"
)
# Rename the columns using the new_names vector
sp500 <- sp500 %>%
  rename_with(~ new_names, everything())
```

This code is designed to rename the columns of the `sp500` tibble using a predefined mapping of new column names. Let's go through the code step by step:

1. A vector named `new_names` is created, which contains the desired new names for each column in the `sp500` tibble. Each element in the `new_names` vector corresponds to a specific column in the `sp500` tibble and represents the desired new name for that column.
2. The `%>%` operator, often referred to as the pipe operator, is used to pass the `sp500` tibble to the subsequent operation in a more readable and concise manner.
3. The `rename_with()` function from the `dplyr` package is applied to the `sp500` tibble. This function allows us to rename columns based on a specified function or formula.

4. In this case, a formula ~ `new_names` is used as the first argument of `rename_with()`. This formula indicates that the new names for the columns should be sourced from the `new_names` vector.
5. The second argument, `everything()`, specifies that the renaming should be applied to all columns in the `sp500` tibble.
6. Finally, the resulting tibble with the renamed columns is assigned back to the `sp500` variable, effectively updating the tibble with the new column names.
7. We could also use the following code to rename the columns.

```
# Rename the columns using the new_names vector
colnames(sp500) <- new_names
```

In essence, the code uses the `new_names` vector as a mapping to assign new column names to the `sp500` tibble, ensuring that each column is given the desired new name specified in `new_names`.

6.3.2 Review the data again after renaming columns

1. We review the column names again after renaming them, using the `colnames()` function can help.

```
colnames(sp500)
```

```
 [1] "Date"                 "Stock"                "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"
```

6.3.3 Understand the Data Columns

1. The complete data has 36 columns. Our goal is to gain a deeper understanding of what the data columns mean.
2. 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
Date	Date (e.g. "7/15/2023")
Stock	Stock Ticker (e.g. AAL)
StockName	Name of the company (e.g "American Airlines Group, Inc.")
Sector	Sector the stock belongs to (e.g. "Transportation")
Industry	Industry the stock belongs to (e.g "Airlines")
MarketCap	Market capitalization of the company
Price	Recent Stock Price

b. The column names described in Table 1b. are related to **Technical Analysis** of each stock, 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

Table 1c: Data Columns related to Profitability	
ColumnName	Description

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

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

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

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

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

6.3.4 Remove Rows containing no data or Null values

1. The following code checks if the "Stock" column in the sp500 dataframe contains any null or blank values. If there are null or blank values present, it removes the corresponding rows from the sp500 dataframe, resulting in a filtered dataframe without null or blank values in the "Stock" column.

```
# Check for blank or null values in the "Stock" column
hasNull <- any(sp500$Stock == "" | is.null(sp500$Stock))
if (hasNull) {
    # Remove rows with null or blank values from the dataframe tibble
    sp500 <- sp500[!(is.null(sp500$Stock) | sp500$Stock == ""), ]
}
```

Here's an alternate code using `dplyr` to achieve the same result:

```
library(dplyr)
# Check for blank or null values in the "Stock" column
hasNull <- any(sp500 %>% pull(Stock) == "" | is.null(sp500 %>% pull(Stock)))
if (hasNull) {
  # Remove rows with null or blank values from the dataframe tibble
  sp500 <- sp500 %>% filter(!(is.null(Stock) | Stock == ""))
}
```

```r
# View the filtered dataframe
nrow(sp500)
```

```
[1] 503
```

Thus, we have 502 stocks of the S&P500 in our dataset.

6.3.5 S&P500 Sector

The S&P500 shares are divided into multiple Sectors. Each stock belongs to a unique sector. Thus, it makes sense to model Sector as a factor() variable.

```r
sp500$Sector <- as.factor(sp500$Sector)
```

It makes sense to convert Sector to a factor variable, since there are 19 distinct Sectors in the S&P500 and each stock belongs to a unique sector. We confirm that Sector is now modelled as a factor variable, by running the str() function.

```r
str(sp500$Sector)
```

```
 Factor w/ 19 levels "Commercial Services",..: 11 18 16 7 11 6 11 9 17 17 ...
```

Now that Sectors is a factor variable, we can use the levels() function to review the different levels it can take.

```r
levels(sp500$Sector)
```

```
 [1] "Commercial Services"    "Communications"         "Consumer Durables"
 [4] "Consumer Non-Durables"  "Consumer Services"      "Distribution Services"
 [7] "Electronic Technology"  "Energy Minerals"        "Finance"
[10] "Health Services"        "Health Technology"      "Industrial Services"
[13] "Non-Energy Minerals"    "Process Industries"     "Producer Manufacturing"
[16] "Retail Trade"           "Technology Services"    "Transportation"
[19] "Utilities"
```

The table() function allows us to count how many stocks are part of each sector.

```r
table(sp500$Sector)
```

```
   Commercial Services         Communications      Consumer Durables 
                    13                      3                     12 
 Consumer Non-Durables      Consumer Services Distribution Services 
                    31                     29                      9 
 Electronic Technology        Energy Minerals                Finance 
                    49                     16                     92 
       Health Services      Health Technology    Industrial Services 
                    12                     47                      9 
   Non-Energy Minerals     Process Industries Producer Manufacturing 
                     7                     24                     31 
          Retail Trade    Technology Services         Transportation 
                    23                     50                     15 
             Utilities 
                    31 
```

Thus, we can see how many stocks are part of each one of the 19 sectors.

We can sum them to confirm that they add up to 502.

```r
sum(table(sp500$Sector))
```

```
[1] 503
```

This completes our review of the Sector variable.

6.3.6 Stock Ratings

In the data, the S&P500 shares have Technical Ratings such as {Buy, Sell, ..}. Since each Stock has a unique Technical Rating, it makes sense to model the data column Rating as a factor() variable.

```r
sp500$Rating <- as.factor(sp500$Rating)
```

We confirm that Rating is now modelled as a factor variable, by running the str() function.

```r
str(sp500$Rating)
```

```
 Factor w/ 5 levels "Buy","Neutral",..: 3 1 1 3 3 3 5 3 1 3 ...
```

We can use the levels() function to review the different levels it can take.

```r
levels(sp500$Rating)
```

```
[1] "Buy"         "Neutral"     "Sell"        "Strong Buy"  "Strong Sell"
```

The table() function allows us to count how many stocks have each Rating.

```r
table(sp500$Rating)
```

```

        Buy     Neutral        Sell  Strong Buy Strong Sell 
        154          60         212          37          40 
```

Thus, we can see how many stocks have ratings ranging from "Strong Sell" to "Strong Buy". This completes our review of Technical Rating.

6.3.7 Summary

We believe this dataset of S&P500 shares is now ready for futher analysis. We end this stage of our analysis in this chapter, by running the str() function to review the data columns.

```r
str(sp500)
```

```
spc_tbl_ [503 x 36] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
 $ Date                   : chr [1:503] "7/25/2023" "7/25/2023" "7/25/2023" "7/25/2023" ...
 $ Stock                  : chr [1:503] "A" "AAL" "AAP" "AAPL" ...
 $ StockName              : chr [1:503] "Agilent Technologies, Inc." "American Airlines Group,
 $ Sector                 : Factor w/ 19 levels "Commercial Services",..: 11 18 16 7 11 6 11 9
 $ Industry               : chr [1:503] "Medical Specialties" "Airlines" "Specialty Stores" "Te
 $ MarketCap              : num [1:503] 3.73e+10 1.14e+10 4.20e+09 3.04e+12 2.53e+11 ...
 $ Price                  : num [1:503] 126.4 17.5 70.7 193 143.6 ...
 $ Low52Wk                : num [1:503] 113.3 11.7 63.6 124.2 131 ...
 $ High52Wk               : num [1:503] 160 19.1 212 198 168 195 116 81.9 328 539 ...
 $ ROE                    : num [1:503] 24.8 NA 14.6 146 51.1 389 NA 14.8 30.7 33.7 ...
 $ ROA                    : num [1:503] 12.7 3.9 3.35 27.6 5.43 2.79 NA 4.98 14.9 17.9 ...
```

```
$ ROIC                 : num [1:503] 16.51 8.01 6.17 57.18 9.9 ...
$ GrossMargin          : num [1:503] 54.1 23.8 43.8 43.2 72.2 ...
$ OperatingMargin      : num [1:503] 23.78 9.39 5.63 29.16 41.07 ...
$ NetMargin            : num [1:503] 19.19 4.98 3.61 24.49 13.3 ...
$ PE                   : num [1:503] 27.86 4.78 10.5 32.8 33.82 ...
$ PB                   : num [1:503] 7.04 NA 1.56 60.73 14.73 ...
$ EVEBITDA             : num [1:503] 19.5 5.71 8.8 25 9.64 12.9 NA NA 17.3 33.4 ...
$ EBITDA               : num [1:503] 1.97e+09 7.16e+09 9.21e+08 1.24e+11 3.18e+10 ...
$ EPS                  : num [1:503] 4.54 3.67 6.72 5.89 4.25 ...
$ EBITDA_YOY           : num [1:503] 10.52 1074.1 -16 -5.36 10.6 ...
$ EBITDA_QYOY          : num [1:503] 8.2 72.2 -39.01 -4.58 11.68 ...
$ EPS_YOY              : num [1:503] 9.17 NA -25.21 -4.33 -39.11 ...
$ EPS_QYOY             : num [1:503] 11.69944 154.13308 -68.36829 -0.00656 -94.89037 ...
$ PFCF                 : num [1:503] 31.74 7.88 NA 31.38 10.84 ...
$ FCF                  : num [1:503] 11.81 NA -100.23 -7.85 6.68 ...
$ FCF_QYOY             : num [1:503] 55.7078 -10.2542 -176.1352 -0.0312 -15.3392 ...
$ DebtToEquity         : num [1:503] 0.473 NA 1.582 1.763 4.678 ...
$ CurrentRatio         : num [1:503] 2.37 0.749 1.244 0.94 0.96 ...
$ QuickRatio           : num [1:503] 1.708 0.656 0.238 0.878 0.821 ...
$ DividendYield        : num [1:503] 0.723 NA 1.428 0.497 4.163 ...
$ DividendsPerShare_YOY: num [1:503] 8.25 NA 84.62 5.88 7.53 ...
$ PS                   : num [1:503] 5.538 0.235 0.384 7.992 4.399 ...
$ Revenue_YOY          : num [1:503] 7.8597 29.9089 1.4153 -0.2544 0.0282 ...
$ Revenue_QYOY         : num [1:503] 6.85 4.72 1.29 -2.51 -9.7 ...
$ Rating               : Factor w/ 5 levels "Buy","Neutral",..: 3 1 1 3 3 3 5 3 1 3 ...
- attr(*, "spec")=
 .. cols(
 ..   Date = col_character(),
 ..   Stock = col_character(),
 ..   Description = col_character(),
 ..   Sector = col_character(),
 ..   Industry = col_character(),
 ..   `Market Capitalization` = col_double(),
 ..   Price = col_double(),
 ..   `52 Week Low` = col_double(),
 ..   `52 Week High` = col_double(),
 ..   `Return on Equity (TTM)` = col_double(),
 ..   `Return on Assets (TTM)` = col_double(),
 ..   `Return on Invested Capital (TTM)` = col_double(),
 ..   `Gross Margin (TTM)` = col_double(),
 ..   `Operating Margin (TTM)` = col_double(),
 ..   `Net Margin (TTM)` = col_double(),
 ..   `Price to Earnings Ratio (TTM)` = col_double(),
```

```
..   `Price to Book (FY)` = col_double(),
..   `Enterprise Value/EBITDA (TTM)` = col_double(),
..   `EBITDA (TTM)` = col_double(),
..   `EPS Diluted (TTM)` = col_double(),
..   `EBITDA (TTM YoY Growth)` = col_double(),
..   `EBITDA (Quarterly YoY Growth)` = col_double(),
..   `EPS Diluted (TTM YoY Growth)` = col_double(),
..   `EPS Diluted (Quarterly YoY Growth)` = col_double(),
..   `Price to Free Cash Flow (TTM)` = col_double(),
..   `Free Cash Flow (TTM YoY Growth)` = col_double(),
..   `Free Cash Flow (Quarterly YoY Growth)` = col_double(),
..   `Debt to Equity Ratio (MRQ)` = col_double(),
..   `Current Ratio (MRQ)` = col_double(),
..   `Quick Ratio (MRQ)` = col_double(),
..   `Dividend Yield Forward` = col_double(),
..   `Dividends per share (Annual YoY Growth)` = col_double(),
..   `Price to Sales (FY)` = col_double(),
..   `Revenue (TTM YoY Growth)` = col_double(),
..   `Revenue (Quarterly YoY Growth)` = col_double(),
..   `Technical Rating` = col_character()
.. )
- attr(*, "problems")=<externalptr>
```

7 Exploring *tibbles* & *dplyr*

July 25, 2023

7.1 tibbles

A `tibble` is essentially an updated version of the conventional data frame, providing more flexible and effective data management features (Müller & Wickham, 2021).

`Tibbles`, also recognized as `tbl_df`, are a component of the tidyverse suite, a collection of R packages geared towards making data science more straightforward. They share many properties with regular data frames 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].

7.2 Basic functions in 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]

7.3 The pipe operator %>%

The `%>%` operator, colloquially known as the "pipe" operator, plays a vital role in enhancing the effectiveness of the `dplyr` package. The purpose of this operator is to facilitate a more readable and understandable chaining of multiple operations. Although this operator was originally introduced by the `magrittr` package, it has since become extensively adopted in `dplyr` and other tidyverse packages.

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. 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. [3]

7.4 Illustration: Using dplyr on mtcars data

We will now illustrate the crucial functions from the `dplyr` package, on the `mtcars` dataset.

7.4.1 Loading required R packages

```
# Load the required libraries, suppressing annoying startup messages
library(dplyr)
```

```

Attaching package: 'dplyr'

The following objects are masked from 'package:stats':

    filter, lag

The following objects are masked from 'package:base':

    intersect, setdiff, setequal, union
```

```
library(tibble)
```

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 the `suppressPackageStartupMessages()`.

```
# Load the required libraries, suppressing annoying startup messages
suppressPackageStartupMessages(library(dplyr))
```

7.4.2 Reading and Viewing the mtcars dataset as a tibble

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

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

```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,~
```

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. Then, 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
# 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)
```

Changing data types: The `as.factor()` function is used to convert the 'cyl', 'vs', 'am', and 'gear' columns from numeric data types to factors. Factors are used in statistical modeling to represent categorical variables. 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.

```
# 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 futher exploration.

7.4.3 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.

```
filtered_data <- tb %>% filter(mpg > 25)
filtered_data
```

```
# 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
```

The tibble 'filtered_data' contains only 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 number of gears is equal to 5.

```
filtered_data2 <- tb %>% filter(mpg > 25 & gear == 5)
filtered_data2
```

```
# 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
```

Thus, we can impose more than one logical conddition in the filter.

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

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

```
# 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 `selected_data` will only contain the `mpg` (miles per gallon), `hp` (horsepower), `cyl` (cylinders) and `am` transmission columns from the mtcars dataset.

4. Now suppose we wanted to both filter and select. Specificially, 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 two primary functions from the dplyr package, 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 > 25 & 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 25 and gear is equal to 5. This application effectively creates a subset of tb that satisfies these conditions (Wickham & Francois, 2016).
- `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 mtcars tibble has the following data columns.

```r
colnames(tb)
```

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

```r
selected_data2 <- tb %>% select(cyl:wt)
selected_data2
```

```
# 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 instruction tells R to select 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.

```r
selected_data3 <- tb %>% select(-cyl:wt)
selected_data3
```

```
# 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 command tells R to select all columns in the tb dataframe, excluding those from cyl to wt inclusive.

As can be seen, the six columns excluding those within the range of cyl and wt, are selected.

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 from the `mtcars` data and sort it in descending order of `mpg`.

```
arranged_data <- tb %>% select(mpg, hp) %>% arrange(desc(mpg))
arranged_data
```

```
# A tibble: 32 x 2
     mpg    hp
   <dbl> <dbl>
 1  33.9    65
 2  32.4    66
 3  30.4    52
 4  30.4   113
 5  27.3    66
 6  26      91
 7  24.4    62
 8  22.8    93
 9  22.8    95
10  21.5    97
# i 22 more rows
```

The steps in the code can be broken down as follows:

`arranged_data <- tb %>% select(mpg, hp)`: The select function is used here to extract only the `mpg` and `hp` columns from the `tb` dataframe. The `%>%` operator is the pipe operator, which is used to chain multiple operations together in a readable manner. This part of the code will create a new dataframe containing only the `mpg` and `hp` columns.

`arrange(desc(mpg))`: The arrange function is then used to order the rows in the newly created dataframe 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 the pipe operator, we would have to nest your operations like this:

```
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
```

Here's how we would do the same operations using the pipe operator:

```
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
```

Here's what each line is doing:

`tb %>%` sends the mtcars data frame into the `filter()` function.

`filter(cyl == 6) %>%` filters the data frame to include only rows where `cyl` is equal to 6, then sends this filtered data frame to the `select()` function.

`select(mpg, hp) %>%` selects only the `mpg` and `hp` columns from the data frame, then sends this subset of the data to the `arrange()` function.

`arrange(desc(mpg))` arranges the rows of the data frame in descending order based on the mpg column.

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 in the mtcars dataset.

```
mutated_data <- tb %>% mutate(efficiency = mpg / hp)
mutated_data
```

```
# A tibble: 32 x 12
     mpg cyl    disp    hp  drat    wt  qsec vs    am    gear   carb efficiency
   <dbl> <fct> <dbl> <dbl> <dbl> <dbl> <dbl> <fct> <fct> <fct> <dbl>      <dbl>
 1  21   6      160    110  3.9   2.62  16.5 0     1     4          4      0.191
 2  21   6      160    110  3.9   2.88  17.0 0     1     4          4      0.191
 3  22.8 4      108     93  3.85  2.32  18.6 1     1     4          1      0.245
 4  21.4 6      258    110  3.08  3.22  19.4 1     0     3          1      0.195
 5  18.7 8      360    175  3.15  3.44  17.0 0     0     3          2      0.107
 6  18.1 6      225    105  2.76  3.46  20.2 1     0     3          1      0.172
 7  14.3 8      360    245  3.21  3.57  15.8 0     0     3          4      0.0584
 8  24.4 4      147.    62  3.69  3.19  20   1     0     4          2      0.394
 9  22.8 4      141.    95  3.92  3.15  22.9 1     0     4          2      0.24
10  19.2 6      168.   123  3.92  3.44  18.3 1     0     4          4      0.156
# i 22 more rows
```

The tibble `mutated_data` will contain a new column `efficiency`, which is the ratio of `mpg` to `hp`. The original data columns in `tb` will be retained.

Remember that these functions do not modify the original dataset, they create new objects with the results. If we want to modify the original dataset, we would need to save the result back to the original variable, or use the mutate_at, mutate_all, mutate_if functions to modify specific columns directly.

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

```
summary_data <- tb %>% summarise(mean_mpg = mean(mpg))
summary_data
```

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

The tibble `summary_data` will contain a single row with the mean value of `mpg` in the mtcars dataset.

11. To include additional statistical measures such as median, quartiles, minimum, and maximum in your 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
```

12. We could convert this back into a standard dataframe and display it.

```
summary_df <- as.data.frame(summary_data)
print(summary_df)
```

```
   N     Mean       SD Median     Q1   Q3  Min  Max
1 32 20.09062 6.026948   19.2 15.425 22.8 10.4 33.9
```

And if we wanted to display only two decimal places, we could code

```
summary_df %>% 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
```

7.5 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.

7.5.1 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.

```
renamed_data <- tb %>% rename(MPG = mpg)
renamed_data
```

```
# 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
```

The dataframe renamed_data now includes the `MPG` column, which was previously named `mpg`.

2. **`group_by()`:** This function is key for performing operations within distinct groups of our data. For example, let us group the `mtcars` dataset by the `cyl` (number of cylinders) column.

```
grouped_data <- tb %>% group_by(cyl)
grouped_data
```

```
# A tibble: 32 x 11
# Groups:   cyl [3]
     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
```

The grouped_data dataframe is now grouped by the `cyl` column, which enables us to carry out operations on each group separately.

3. **`slice()`:** Remember that this function is beneficial when we wish to choose rows based on their positions. For example, let's select the first three rows of the `mtcars` dataset.

```r
sliced_data <- tb %>% slice(1:3)
sliced_data
```

```
# A tibble: 3 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
```

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

4. **`transmute()`:** Recall that if we desire to create new variables and keep only these variables, we apply the `transmute()` function. Suppose we want to create a new variable that is the ratio of horsepower (`hp`) to weight (`wt`), and keep only this new variable.

```r
transmuted_data <- tb %>% transmute(hp_to_wt = hp/wt) %>% head()
transmuted_data
```

```
# A tibble: 6 x 1
  hp_to_wt
     <dbl>
1     42.0
2     38.3
3     40.1
4     34.2
5     50.9
6     30.3
```

The `transmuted_data` tibble now includes the newly created `hp_to_wt` variable, while the other columns have been removed.

5. **`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 `mtcars` dataset.

```
pulled_data <- tb %>% pull(mpg)
pulled_data
```

```
 [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
```

In the `pulled_data` vector, only the values from the `mpg` variable are retained.

6. **`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 thecyl(cylinders) column from themtcars`‘ dataset.

```
distinct_count <- tb %>% summarise(n_distinct_cyl = n_distinct(cyl))
distinct_count
```

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

The `distinct_count` dataframe shows the number of unique values in the `cyl` column of the `mtcars` dataset.

7.6 Summary

This chapter has 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 `%>%`, has also been 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 has given 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.

7.7 References

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

[3] Bache, S. M., & Wickham, H. (2020). magrittr: A Forward-Pipe Operator for R. R package version 2.0.1.

https://CRAN.R-project.org/package=magrittr

Wickham, H. (2014). Tidy data. Journal of Statistical Software, 59(10), 1-23.

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.

8 Categorical Data

July 25, 2023 V3 (Work in progress)

8.1 Overview

1. Categorical data is a type of data that can be divided into categories or groups.
2. Text labels or categorical codes like "male" and "female," "red," "green," and "blue," or "A," "B," and "C" are frequently used to describe category data. 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)
- Eye color (brown, blue, green, hazel)
- Type of car (sedan, SUV, truck)

8.2 Types of Categorical Data – Nominal, Ordinal Data

1. Nominal and ordinal data are two types of categorical data.
2. Nominal data is a type of categorical data that has no inherent order or numerical value.

- For describing categories or groups that are simply named or labelled, such as hair colour, eye colour, or car type, nominal data is frequently employed.
- Nominal data is usually represented by text labels or categorical codes.

3. Ordinal data is a kind of categorical data that naturally has order, while the distinctions between the categories are not always equal.

- 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).

- Ordinal data is usually represented by numerical codes that indicate the order of the categories.

8.3 Categorical Data in R

1. There are several ways to summarize categorical data in R.
2. `table()` function: The frequency table for a categorical vector is returned by the table() function.

- Frequency Table for One Variable

```
data(mtcars)
attach(mtcars)
t0 = table(cyl)
t0
```

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

3. As an alternative, you can use the summary() function to create a summary table for categorical data. This function returns a summary table of the frequency counts for each category and accepts a factor or an object of class "table."

8.3.1 `summary()`

```
summary(cyl)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
  4.000   4.000   6.000   6.188   8.000   8.000 
```

8.4 Frequency Table for More than One Variable

8.4.1 table()

```
t1 = table(am, cyl)
t1
```

```
   cyl
am   4  6  8
  0  3  4 12
  1  8  3  2
```

In this example, 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, while 3 have an automatic transmission and 3.

8.4.2 xtabs()

```
t1 =xtabs(~ cyl + gear
          , data = mtcars)
t1
```

```
   gear
cyl  3  4  5
  4  1  8  2
  6  2  4  1
  8 12  0  2
```

In this example, we generate a two-way contingency table of am and cyl using the xtab() method. 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, while 3 have an automatic transmission and 3. Observe that the table() function used in the preceding example and the xtab() function both yield the same outcome.

8.4.3 ftable()

```
t2 = ftable(gear ~ cyl
            , data = mtcars)
t2
```

```
    gear  3  4  5
cyl
4         1  8  2
6         2  4  1
8        12  0  2
```

In this example, a two-way contingency table of gear and cyl is created using the ftable() function. The frequency of each grouping of categories is displayed in the table that results. As an illustration, there are 12 automobiles with 8 cylinders and 3 speeds as well as 1 car with 4 cylinders.

8.5 Proportions Table for One Variable

- `prop.table`
- Unlike table(), which delivers the count, this function returns the proportions of each category.

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

```
cyl
      4       6       8
0.34375 0.21875 0.43750
```

The prop.table() function is used in this example to determine the percentage of each category in the cyl variable of the mtcars dataset. The fraction of cars with 4, 6, and 8 cylinders, respectively, is represented in the resulting vector p0. For instance, the dataset contains cars with 4 cylinders in 34.375% of the cases.

8.6 Proportions Table for More than One Variable

prop.table

```
t1 = table(am, cyl)
p1 = prop.table(t1)
p1
```

```
   cyl
am        4       6       8
  0 0.09375 0.12500 0.37500
  1 0.25000 0.09375 0.06250
```

In this example, we generate a frequency table (t1) of the variables am and cyl from the mtcars dataset using the table() method. The frequency of each grouping of categories is displayed in the table that results. 12 automobiles, for instance, have a V-shaped engine, a manual transmission, and 8 cylinders. The same data are then used to generate a proportion table (p1) using the prop.table() function. The percentage of each combination of categories is displayed in the following table. For instance, the dataset contains 56.25% of vehicles with a manual transmission, a V-shaped engine, and 8 cylinders.

8.7 Rounding

8.8 This function is used to set the width of decimal numbers

round()

```
r1 = round(p0*100,2)
r1
```

```
cyl
    4     6     8
34.38 21.88 43.75
```

```
r2 = round(p1*100,2)
r2
```

```
    cyl
am      4     6     8
  0  9.38 12.50 37.50
  1 25.00  9.38  6.25
```

In this example, we round the proportion tables p0 and p1 to two decimal places using the round() method. The proportion of each category or group of categories, rounded to two decimal places, is presented in the ensuing tables r1 and r2. For instance, 56.25% of automobiles have an automated transmission and 8 cylinders.

8.9 addmargins()

The row and column sums of a matrix or table are calculated using the addmargins() function in R, and the sums are then added as new rows and columns to the original matrix or table.

```r
r2 = round(p1*100,2)
m1 = addmargins(r2)
m1
```

```
     cyl
am         4      6      8    Sum
  0     9.38  12.50  37.50  59.38
  1    25.00   9.38   6.25  40.63
  Sum  34.38  21.88  43.75 100.01
```

In this illustration, we add row and column margins to the rounded proportion table r2 using the addmargins() function. The proportion of each category or group of categories, rounded to two decimal places, is included in the resulting table m1, along with row and column margins that display the sums for each row and column. For example, the total proportion of cars with 8 cylinders is 60.42%.

8.10 Three Way Relationship

8.10.1 table()

```
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
```

In this example, a three-way contingency table of cyl, gear, and am is created using the table() function. The frequency of each grouping of categories is displayed in the table that results. One vehicle has four cylinders, three gears, and an automatic transmission, whereas eight vehicles have four cylinders, four gears, and manual transmissions. The resulting table, which has a two-dimensional table for each level of the am variable, is three-dimensional.

8.10.2 xtabs()

```
xtabs(~ cyl + gear + am
      , data = mtcars)
```

```
, , 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
```

In this example, a three-way contingency table of cyl, gear, and am is created using the xtabs() function. The frequency of each grouping of categories is displayed in the table that results. One vehicle has four cylinders, three gears, and an automatic transmission, whereas eight vehicles have four cylinders, four gears, and manual transmissions. The resulting table, which has a two-dimensional table for each level of the am variable, is three-dimensional. The output table matches the one created by the table() function used in the preceding example exactly.

8.10.3 ftable()

```
ftable(gear + cyl ~ am
       , data = mtcars)
```

```
   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 example, a three-way contingency table of gear, cyl, and am is created using the ftable() function. The frequency of each grouping of categories is displayed in the table that results. One car has four cylinders, three gears, and an automatic gearbox, whereas there are eight cars with four cylinders, four speeds, and manual transmissions. One table exists for each level of the am variable, resulting in a two-dimensional table. Similar to the table created by the xtabs() function used in the preceding example, the output table is produced.

8.11 Four Way Relationship

```
ftable(am + cyl ~ gear + vs
       , data = mtcars)
```

```
         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
```

In this example, we establish a four-way contingency table containing am, cyl, gear, and vs using the ftable() function. The frequency of each grouping of categories is displayed in the table that results. There are two vehicles with a 6-cylinder, 3-gear, automatic transmission, and inline engine, for instance, and three vehicles with four cylinders. The resulting table, which has two two-dimensional tables for each level of the am variable, is four-dimensional.

8.12 Confidence Interval for a population proportion

In statistics, a confidence interval is a set of values that is thought, with a certain degree of confidence, to contain the real population parameter. The degree of assurance that the genuine population parameter falls inside the interval is indicated by the confidence level, which is typically represented as a percentage.

A range of values that, with a particular degree of confidence, are likely to include the genuine population proportion is known as a confidence interval. The sample size, sample proportion, and desired level of confidence—which is typically stated as a percentage—are used to compute the interval.

The general formula for a 95% confidence interval for a population proportion is:

p̂ ± z*√(p̂(1-p̂)/n)

where: p̂ = sample proportion z = the Z-score corresponding to the desired level of confidence n = sample size

8.12.1 Example of a Confidence Interval for a population proportion

For example, suppose you conduct a survey of 1000 people and find that 120 of them support a particular political candidate. The sample proportion is $\hat{p} = 120/1000 = 0.12$. To find the 95% confidence interval for the population proportion, we can use the formula:

$\hat{p} \pm 1.96 * \sqrt{}(\hat{p}(1-\hat{p})/n)$

$= 0.12 \pm 1.96 * \sqrt{}(0.12(0.88)/1000)$

$= 0.12 \pm 0.024$

Thus, the population proportion's 95% confidence interval is (0.096, 0.144). This indicates that the true population proportion is between 0.096 and 0.144 with a 95% confidence level.

The result is the mtcars data's 95% confidence interval for the percentage of vehicles having automatic transmissions.

This indicates that we have a 95% confidence level that the population's actual proportion of cars with automatic transmissions is between 0.2455 and 0.3694.

8.12.2 Justifying a Claim Based on a Confidence Interval for a Population Proportion

Justifying a claim based on a confidence interval for a population proportion involves two steps:

Interpreting the confidence interval: The confidence interval provides an estimate of the range of values that the true population proportion is likely to fall within. The interval's confidence level expresses how confidently we can say that the true population proportion falls within the interval.

Making the claim: You can use the confidence interval to support a claim if it falls within the range of the interval. If, for instance, the claim is that at least 0.4 of individuals prefer a certain brand of cereal and the 95% confidence interval for the population proportion is between 0.38 and 0.42, you can still support the claim because 0.4 is within the range.

It is crucial to keep in mind that a confidence interval just provides an estimate of where the genuine population proportion is likely to be, not a guarantee. We are less convinced about the position of the genuine population percentage and the estimate's uncertainty increases with the interval's width.

8.12.3 Confidence Intervals for the Difference of Two Proportions

Based on sample data, the difference between two population proportions is estimated using a confidence range for the difference between two proportions. When comparing the proportions of two different groups or treatments, this style of confidence interval is frequently utilised.

To calculate a confidence interval for the difference of two proportions, you need to have a sample of data from each group or treatment. The sample proportion for each group is then used to estimate the population proportion for that group.

Here's the general formula for a confidence interval for the difference of two proportions:

CI = p1 - p2 ± z*sqrt(p1(1-p1)/n1 + p2(1-p2)/n2)

where:

p1 and p2 are the sample proportions for the two groups or treatments n1 and n2 are the sample sizes for the two groups or treatments z is the z-score that corresponds to the desired level of confidence (for example, 1.96 for a 95% confidence interval) sqrt(p1(1-p1)/n1 + p2(1-p2)/n2) is the standard error of the difference of two proportions The confidence interval gives a range of values that is likely to contain the true difference between the two population proportions with a certain level of confidence (for example, 95%). If the confidence interval does not include zero, it provides evidence that the two population proportions are different. The width of the confidence interval depends on the sample sizes, the sample proportions, and the level of confidence desired

8.12.4 Confidence Intervals for the Difference of Two Proportions in R

8.13 Visualization of Categorical Variable

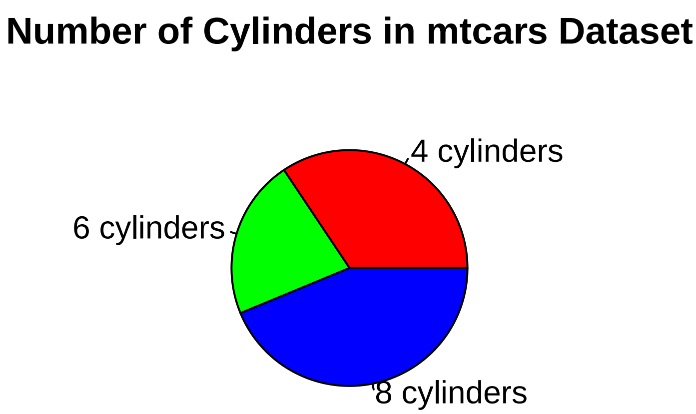
8.14 Pie chart

A pie chart is a circular graph with wedges or slices cut out of it, each of which represents a certain percentage of the entire. Each slice's size reflects the value it represents, while the chart's overall area reflects the sum of all values.

Pie charts are frequently used to display percentages or proportions of a whole or the relative sizes of various categories. They are very helpful for displaying data with few categories or when highlighting a single category or value.

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

```r
# Create a pie chart
pie(cyl_counts, main = "Number of Cylinders in mtcars Dataset",
    labels = c("4 cylinders", "6 cylinders", "8 cylinders"),
    col = c("red", "green", "blue"))
```



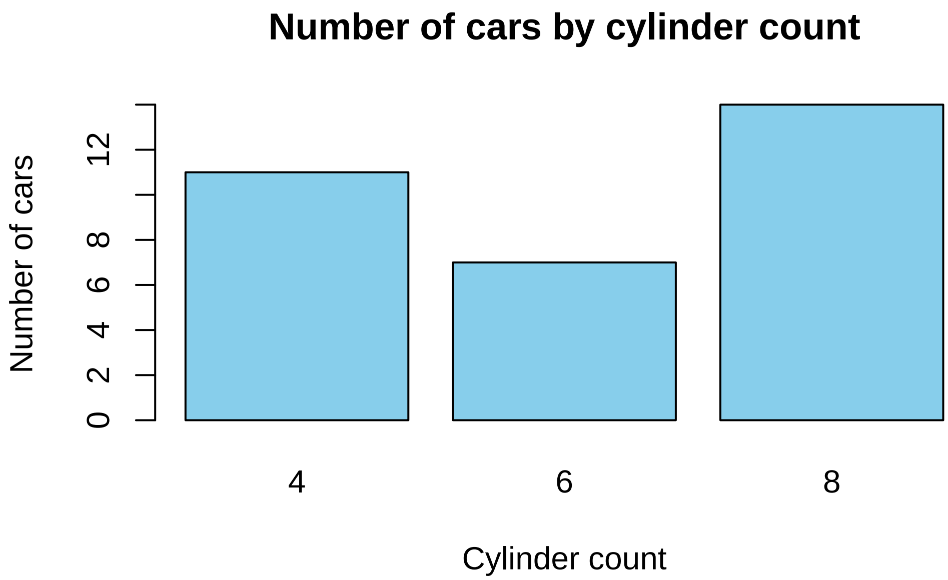
The occurrences of each value of the cyl variable in the mtcars dataset are counted in this code using the table() function, and the resulting table is saved as cyl counts. The cyl counts variable provides the data for the pie chart, which is subsequently created using the pie() function. The chart's title is determined by the main argument, while the labels argument assigns unique labels to the chart's slices. The colours of the slices are specified by the col argument.

8.15 Barplot for categorical data in R

8.15.1 Barplot for Univariate Case

```r
# Load the mtcars dataset
data(mtcars)

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

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

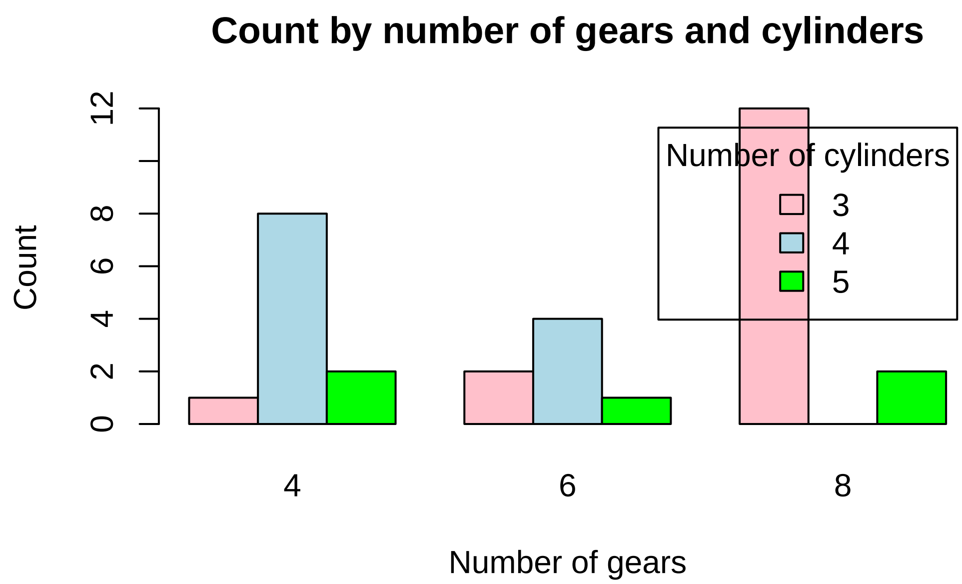
```
        col = "skyblue")
```



By dividing the number of automobiles by the number of cylinders in the mtcars dataset, this code will produce a barplot. The barplot() function is used to generate the actual plot, while the table() function is used to generate a table of counts for the cyl variable in the mtcars dataset. The title and axis labels are added using the main, xlab, and ylab arguments, and the colour of the bars is altered with the col option.

8.15.2 Barplot for Bivariate Case (Grouped Barchart)

```
# Load the mtcars dataset
data(mtcars)

# Create a matrix with count by number of gears and number of cylinders
counts <- table(mtcars$gear, mtcars$cyl)

# Create the bar plot
barplot(counts, beside = TRUE, col = c("pink", "lightblue", "green"),
        xlab = "Number of gears", ylab = "Count",
        main = "Count by number of gears and cylinders",
        legend.text = rownames(counts), args.legend = list(title = "Number of cylinders"))
```



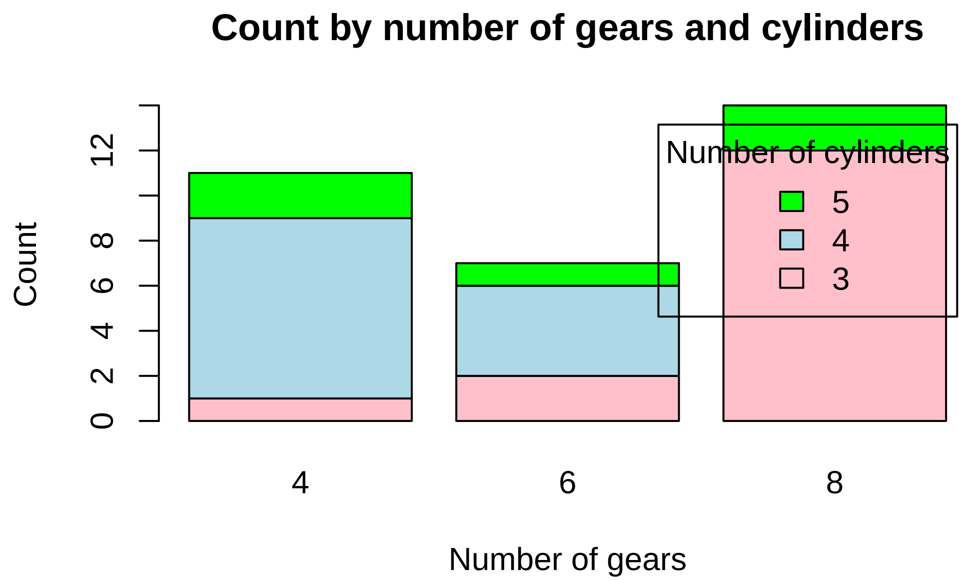
In this code, we first load the mtcars dataset. Then, we use the table() function to compute the counts by number of gears and number of cylinders. We store the result in a matrix called counts.

Finally, we use barplot() to create the plot. We pass the counts matrix as the first argument, and we set beside = TRUE to make sure that the bars are positioned side by side. We also set the colors of the bars using col, and we add labels to the plot using xlab, ylab, and main. We also add a legend to the plot using legend.text and args.legend. Note that rownames(counts) returns the row names of the matrix, which are the number of gears. We set the title of the legend to "Number of cylinders" using args.legend = list(title = "Number of cylinders").

8.15.3 Barplot for Bivariate Case (Stacked Barchart)

```
# Load the mtcars dataset
data(mtcars)

# Create a matrix with count by number of gears and number of cylinders
counts <- table(mtcars$gear, mtcars$cyl)

# Create the stacked bar plot
barplot(counts, col = c("pink", "lightblue", "green"),
        xlab = "Number of gears", ylab = "Count",
        main = "Count by number of gears and cylinders",
        legend.text = rownames(counts), args.legend = list(title = "Number of cylinders"),
```

```
        beside = FALSE)
```



In this code, we first load the mtcars dataset. Then, we use the table() function to compute the counts by number of gears and number of cylinders. We store the result in a matrix called counts.

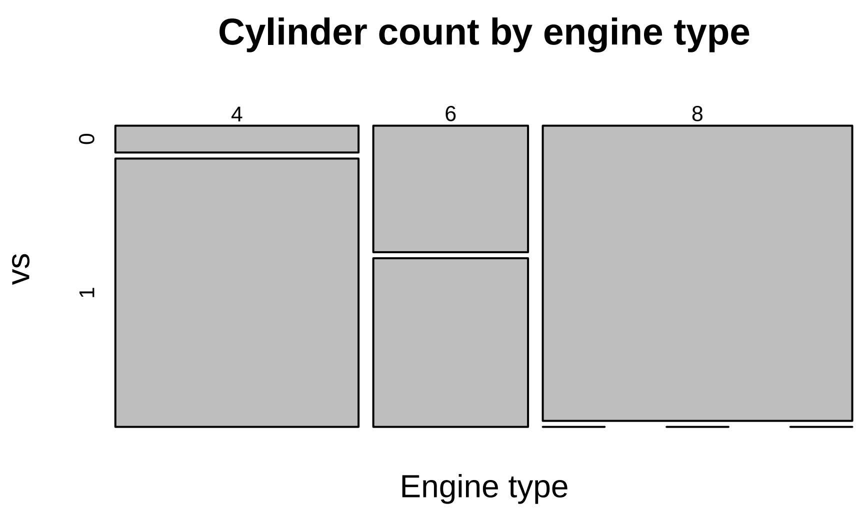
Finally, we use barplot() to create the plot. We pass the counts matrix as the first argument, and we set beside = FALSE to make sure that the bars are stacked on top of each other. We also set the colors of the bars using col, and we add labels to the plot using xlab, ylab, and main. We also add a legend to the plot using legend.text and args.legend. Note that rownames(counts) returns the row names of the matrix, which are the number of gears. We set the title of the legend to "Number of cylinders" using args.legend = list(title = "Number of cylinders").

8.16 Mosaic plot

The distribution of two categorical variables in a dataset is displayed graphically in a mosaic plot. Rectangular blocks with sizes proportionate to the number of observations for each combination of the two variables make up the plot. The relative frequency of each category of the second variable within each category of the first variable is represented by segments inside each block.

The interactions between categorical variables can be visualised using mosaic plots, which can also be used to find patterns and associations in large, complicated datasets. 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.

```
# Load the mtcars dataset
data(mtcars)

# Create a mosaic plot of the data
mosaicplot(table(mtcars$cyl, mtcars$vs),
           main = "Cylinder count by engine type",
           xlab = "Engine type",
           ylab = "vs")
```



With the help of this code, a mosaic plot of the number of vehicles in the mtcars dataset broken down by cylinder count and engine type will be produced (V-shaped or straight). The mosaicplot() method is used to generate the actual plot, and the table() function is used to generate a table of counts for the cyl and vs variables in the mtcars dataset. A title and axis labels are added using the main, xlab, and ylab variables.

To build a mosaic plot of the categorical data in mtcars that interests you, you can change this code. To plot the variables, simply swap out mtcars$cyl and mtcars$vs for the desired values. Remember that mosaic plots can be used to compare the distribution of categories within several groups.

```
# Load the mtcars dataset
data(mtcars)
```

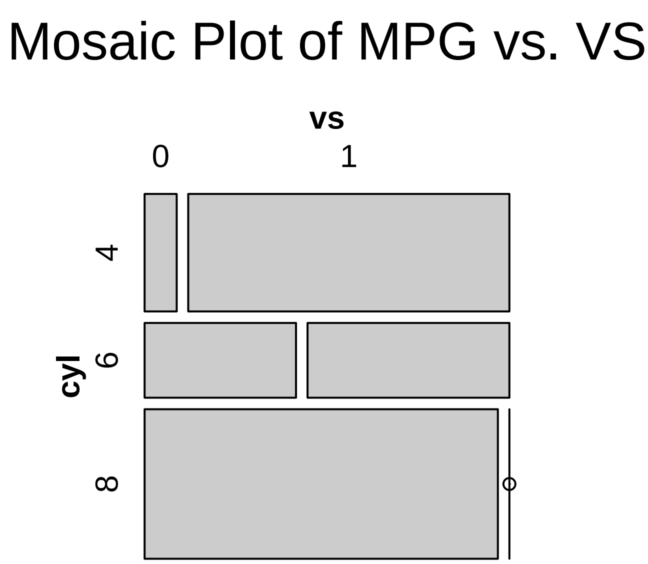
```r
# Install and load the vcd package (if it's not already installed)
install.packages("vcd")
```

```
Installing package into '/cloud/lib/x86_64-pc-linux-gnu-library/4.3'
(as 'lib' is unspecified)
```

```r
library(vcd)
```

```
Loading required package: grid
```

```r
# Create a mosaic plot of mpg (miles per gallon) vs. vs (engine shape)
mosaic(~ cyl + vs, data = mtcars, main = "Mosaic Plot of MPG vs. VS")
```



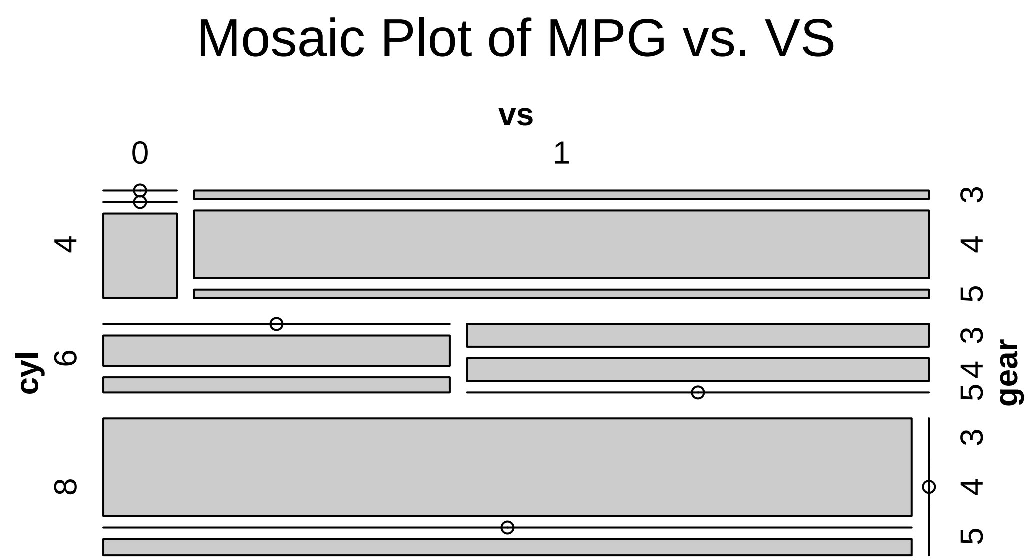
The mtcars dataset, a built-in dataset in R that contains data on 32 cars, is initially loaded by this code. The vcd package, which has utilities for making mosaic plots and other kinds of visualisations, is then installed and loaded by the code.

Finally, using the mosaic() function from the vcd package, the code generates a mosaic plot of the mpg (miles per gallon) and vs (engine shape) variables in the mtcars dataset. The resulting plot illustrates how vehicles with V-shaped vs. straight engines have different mpg distributions (vs values of 0 vs. 1, respectively).

```
# Load the mtcars dataset
data(mtcars)

# Install and load the vcd package (if it's not already installed)
install.packages("vcd")
```

```
Installing package into '/cloud/lib/x86_64-pc-linux-gnu-library/4.3'
(as 'lib' is unspecified)
```

```
library(vcd)

# Create a mosaic plot of mpg (miles per gallon) vs. vs (engine shape)
mosaic(~ cyl + vs + gear, data = mtcars, main = "Mosaic Plot of MPG vs. VS")
```



8.17 References

Healy, K., & Lenard, M. T. (2014). A practical guide to creating better looking plots in R. University of Oregon. https://escholarship.org/uc/item/07m6r

Few, S. (2004). Show me the numbers: Designing tables and graphs to enlighten. Analytics Press.

Friendly, M. (1994). Mosaic displays for multi-way contingency tables. Journal of the American Statistical Association, 89(425), 190-200.

9 Live Case: S&P500 (2 of 3)

July 23, 2023 V2

9.1 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. https://www.investopedia.com/terms/s/sp500.asp

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.

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.

Among these 503 stocks, Apple, the technology giant, holds the top position with a market capitalization of $2.35 billion. Following Apple, Microsoft and Amazon.com rank as the second and third largest stocks in the S&P 500, respectively. The next positions are held by Nvidia Corp, Tesla, Berkshire Hathaway, and two classes of shares from Google's parent company, Alphabet..

9.2 S&P 500 Data - Preliminary Analysis

We will analyze a real-world, recent dataset containing information about the S&P500 stocks. The dataset is located in a Google Sheet

The data is disorganized and challenging to understand. We will review the data and proceed in a step-by-step manner.

9.2.1 Read the S&P500 data from a Google Sheet into a tibble dataframe.

1. The complete URL is https://docs.google.com/spreadsheets/d/11ahk9uWxBkDqrhNm7qYmiTwrlSC53N1zvXYfv7ttOCM/
2. The Google Sheet ID is: `11ahk9uWxBkDqrhNm7qYmiTwrlSC53N1zvXYfv7ttOCM`. We can use the function `gsheet2tbl` in package `gsheet` to read the Google Sheet into a tibble or 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"
url500 <- paste(prefix,sheetID) # Form the URL to connect to
sp500 <- gsheet2tbl(url500) # Read it into a tibble called sp500
```

9.3 Review the data

1. We want to understand the different data columns and their data structure. For this purpose, we run the `str()` function.

```
str(sp500)
```

```
spc_tbl_ [503 x 36] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
 $ Date                                  : chr [1:503] "7/25/2023" "7/25/2023" "7/25/2023" "
 $ Stock                                 : chr [1:503] "A" "AAL" "AAP" "AAPL" ...
 $ Description                           : chr [1:503] "Agilent Technologies, Inc." "America
 $ Sector                                : chr [1:503] "Health Technology" "Transportation"
 $ Industry                              : chr [1:503] "Medical Specialties" "Airlines" "Spe
 $ Market Capitalization                 : num [1:503] 3.73e+10 1.14e+10 4.20e+09 3.04e+12 2
 $ Price                                 : num [1:503] 126.4 17.5 70.7 193 143.6 ...
```

```
$ 52 Week Low                           : num [1:503] 113.3 11.7 63.6 124.2 131 ...
$ 52 Week High                          : num [1:503] 160 19.1 212 198 168 195 116 81.9 328
$ Return on Equity (TTM)                : num [1:503] 24.8 NA 14.6 146 51.1 389 NA 14.8 30
$ Return on Assets (TTM)                : num [1:503] 12.7 3.9 3.35 27.6 5.43 2.79 NA 4.98
$ Return on Invested Capital (TTM)      : num [1:503] 16.51 8.01 6.17 57.18 9.9 ...
$ Gross Margin (TTM)                    : num [1:503] 54.1 23.8 43.8 43.2 72.2 ...
$ Operating Margin (TTM)                : num [1:503] 23.78 9.39 5.63 29.16 41.07 ...
$ Net Margin (TTM)                      : num [1:503] 19.19 4.98 3.61 24.49 13.3 ...
$ Price to Earnings Ratio (TTM)         : num [1:503] 27.86 4.78 10.5 32.8 33.82 ...
$ Price to Book (FY)                    : num [1:503] 7.04 NA 1.56 60.73 14.73 ...
$ Enterprise Value/EBITDA (TTM)         : num [1:503] 19.5 5.71 8.8 25 9.64 12.9 NA NA 17.3
$ EBITDA (TTM)                          : num [1:503] 1.97e+09 7.16e+09 9.21e+08 1.24e+11 3
$ EPS Diluted (TTM)                     : num [1:503] 4.54 3.67 6.72 5.89 4.25 ...
$ EBITDA (TTM YoY Growth)               : num [1:503] 10.52 1074.1 -16 -5.36 10.6 ...
$ EBITDA (Quarterly YoY Growth)         : num [1:503] 8.2 72.2 -39.01 -4.58 11.68 ...
$ EPS Diluted (TTM YoY Growth)          : num [1:503] 9.17 NA -25.21 -4.33 -39.11 ...
$ EPS Diluted (Quarterly YoY Growth)    : num [1:503] 11.69944 154.13308 -68.36829 -0.00656
$ Price to Free Cash Flow (TTM)         : num [1:503] 31.74 7.88 NA 31.38 10.84 ...
$ Free Cash Flow (TTM YoY Growth)       : num [1:503] 11.81 NA -100.23 -7.85 6.68 ...
$ Free Cash Flow (Quarterly YoY Growth) : num [1:503] 55.7078 -10.2542 -176.1352 -0.0312 -1
$ Debt to Equity Ratio (MRQ)            : num [1:503] 0.473 NA 1.582 1.763 4.678 ...
$ Current Ratio (MRQ)                   : num [1:503] 2.37 0.749 1.244 0.94 0.96 ...
$ Quick Ratio (MRQ)                     : num [1:503] 1.708 0.656 0.238 0.878 0.821 ...
$ Dividend Yield Forward                : num [1:503] 0.723 NA 1.428 0.497 4.163 ...
$ Dividends per share (Annual YoY Growth): num [1:503] 8.25 NA 84.62 5.88 7.53 ...
$ Price to Sales (FY)                   : num [1:503] 5.538 0.235 0.384 7.992 4.399 ...
$ Revenue (TTM YoY Growth)              : num [1:503] 7.8597 29.9089 1.4153 -0.2544 0.0282
$ Revenue (Quarterly YoY Growth)        : num [1:503] 6.85 4.72 1.29 -2.51 -9.7 ...
$ Technical Rating                      : chr [1:503] "Sell" "Buy" "Buy" "Sell" ...
- attr(*, "spec")=
 .. cols(
 ..   Date = col_character(),
 ..   Stock = col_character(),
 ..   Description = col_character(),
 ..   Sector = col_character(),
 ..   Industry = col_character(),
 ..   `Market Capitalization` = col_double(),
 ..   Price = col_double(),
 ..   `52 Week Low` = col_double(),
 ..   `52 Week High` = col_double(),
 ..   `Return on Equity (TTM)` = col_double(),
 ..   `Return on Assets (TTM)` = col_double(),
 ..   `Return on Invested Capital (TTM)` = col_double(),
```

```
..   `Gross Margin (TTM)` = col_double(),
..   `Operating Margin (TTM)` = col_double(),
..   `Net Margin (TTM)` = col_double(),
..   `Price to Earnings Ratio (TTM)` = col_double(),
..   `Price to Book (FY)` = col_double(),
..   `Enterprise Value/EBITDA (TTM)` = col_double(),
..   `EBITDA (TTM)` = col_double(),
..   `EPS Diluted (TTM)` = col_double(),
..   `EBITDA (TTM YoY Growth)` = col_double(),
..   `EBITDA (Quarterly YoY Growth)` = col_double(),
..   `EPS Diluted (TTM YoY Growth)` = col_double(),
..   `EPS Diluted (Quarterly YoY Growth)` = col_double(),
..   `Price to Free Cash Flow (TTM)` = col_double(),
..   `Free Cash Flow (TTM YoY Growth)` = col_double(),
..   `Free Cash Flow (Quarterly YoY Growth)` = col_double(),
..   `Debt to Equity Ratio (MRQ)` = col_double(),
..   `Current Ratio (MRQ)` = col_double(),
..   `Quick Ratio (MRQ)` = col_double(),
..   `Dividend Yield Forward` = col_double(),
..   `Dividends per share (Annual YoY Growth)` = col_double(),
..   `Price to Sales (FY)` = col_double(),
..   `Revenue (TTM YoY Growth)` = col_double(),
..   `Revenue (Quarterly YoY Growth)` = col_double(),
..   `Technical Rating` = col_character()
.. )
- attr(*, "problems")=<externalptr>
```

2. The `str(sp500)` output provides valuable insights into the structure and data types of the columns in the `sp500` tibble. Let's delve into the details.
3. The output reveals that `sp500` is a tibble with dimensions [503 × 36]. This means it consists of 503 rows, each representing a specific S&P500 stock, and 36 columns containing information about each stock.
4. Here is a preliminary breakdown of the information associated with each column:

- The columns labeled `Date`, `Stock`, `Description`, `Sector`, and `Industry` are character columns. They respectively represent the date, stock ticker symbol, description, sector, and industry of each S&P500 stock.
- Columns such as `Market.Capitalization`, `Price`, `X52.Week.Low`, `X52.Week.High`, and other numeric columns contain diverse financial metrics and stock prices related to the S&P500 stocks.

- The column labeled `Technical.Rating` is a character column that assigns a technical rating to each stock.

5. By examining the `str(sp500)` output, we gain a preliminary understanding of the data types and column names present in the `sp500` tibble, enabling us to grasp the structure of the dataset.

9.3.1 Rename Data Columns

1. The names of the data columns are lengthy and confusing.
2. We will rename the data columns to make it easier to work with the data, using the `rename_with()` function.

```
# Define a mapping of new column names
new_names <- c(
  "Date", "Stock", "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"
)
# Rename the columns using the new_names vector
sp500 <- sp500 %>%
  rename_with(~ new_names, everything())
```

This code is designed to rename the columns of the `sp500` tibble using a predefined mapping of new column names. Let's go through the code step by step:

1. A vector named `new_names` is created, which contains the desired new names for each column in the `sp500` tibble. Each element in the `new_names` vector corresponds to a specific column in the `sp500` tibble and represents the desired new name for that column.
2. The `%>%` operator, often referred to as the pipe operator, is used to pass the `sp500` tibble to the subsequent operation in a more readable and concise manner.
3. The `rename_with()` function from the `dplyr` package is applied to the `sp500` tibble. This function allows us to rename columns based on a specified function or formula.

4. In this case, a formula ~ `new_names` is used as the first argument of `rename_with()`. This formula indicates that the new names for the columns should be sourced from the `new_names` vector.
5. The second argument, `everything()`, specifies that the renaming should be applied to all columns in the `sp500` tibble.
6. Finally, the resulting tibble with the renamed columns is assigned back to the `sp500` variable, effectively updating the tibble with the new column names.
7. We could also use the following code to rename the columns.

```
# Rename the columns using the new_names vector
colnames(sp500) <- new_names
```

In essence, the code uses the `new_names` vector as a mapping to assign new column names to the `sp500` tibble, ensuring that each column is given the desired new name specified in `new_names`.

9.3.2 Review the data again after renaming columns

1. We review the column names again after renaming them, using the `colnames()` function can help.

```
colnames(sp500)
```

```
 [1] "Date"                 "Stock"                "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"
```

9.3.3 Understand the Data Columns

1. The complete data has 36 columns. Our goal is to gain a deeper understanding of what the data columns mean.

2. 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
Date	Date (e.g. ”7/15/2023”)
Stock	Stock Ticker (e.g. AAL)
StockName	Name of the company (e.g ”American Airlines Group, Inc.”)
Sector	Sector the stock belongs to (e.g. ”Transportation”)
Industry	Industry the stock belongs to (e.g ”Airlines”)
MarketCap	Market capitalization of the company
Price	Recent Stock Price

b. The column names described in Table 1b. are related to **Technical Analysis** of each stock, 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

Table 1c: Data Columns related to Profitability

ColumnName	Description

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

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

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

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

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

9.3.4 Remove Rows containing no data or Null values

1. The following code checks if the "Stock" column in the sp500 dataframe contains any null or blank values. If there are null or blank values present, it removes the corresponding rows from the sp500 dataframe, resulting in a filtered dataframe without null or blank values in the "Stock" column.

```r
# Check for blank or null values in the "Stock" column
hasNull <- any(sp500$Stock == "" | is.null(sp500$Stock))
if (hasNull) {
    # Remove rows with null or blank values from the dataframe tibble
    sp500 <- sp500[!(is.null(sp500$Stock) | sp500$Stock == ""), ]
}
```

Here's an alternate code using `dplyr` to achieve the same result:

```r
library(dplyr)
# Check for blank or null values in the "Stock" column
hasNull <- any(sp500 %>% pull(Stock) == "" | is.null(sp500 %>% pull(Stock)))
if (hasNull) {
  # Remove rows with null or blank values from the dataframe tibble
  sp500 <- sp500 %>% filter(!(is.null(Stock) | Stock == ""))
}
```

```
# View the filtered dataframe
nrow(sp500)
```

```
[1] 503
```

Thus, we have 502 stocks of the S&P500 in our dataset.

9.3.5 S&P500 Sector

The S&P500 shares are divided into multiple Sectors. Each stock belongs to a unique sector. Thus, it makes sense to model Sector as a factor() variable.

```
sp500$Sector <- as.factor(sp500$Sector)
```

It makes sense to convert Sector to a factor variable, since there are 19 distinct Sectors in the S&P500 and each stock belongs to a unique sector. We confirm that Sector is now modelled as a factor variable, by running the str() function.

```
str(sp500$Sector)
```

```
 Factor w/ 19 levels "Commercial Services",..: 11 18 16 7 11 6 11 9 17 17 ...
```

Now that Sectors is a factor variable, we can use the levels() function to review the different levels it can take.

```
levels(sp500$Sector)
```

```
 [1] "Commercial Services"    "Communications"         "Consumer Durables"
 [4] "Consumer Non-Durables"  "Consumer Services"      "Distribution Services"
 [7] "Electronic Technology"  "Energy Minerals"        "Finance"
[10] "Health Services"        "Health Technology"      "Industrial Services"
[13] "Non-Energy Minerals"    "Process Industries"     "Producer Manufacturing"
[16] "Retail Trade"           "Technology Services"    "Transportation"
[19] "Utilities"
```

The table() function allows us to count how many stocks are part of each sector.

```r
table(sp500$Sector)
```

```
  Commercial Services        Communications      Consumer Durables 
                   13                     3                     12 
Consumer Non-Durables     Consumer Services Distribution Services 
                   31                    29                      9 
Electronic Technology       Energy Minerals               Finance 
                   49                    16                     92 
      Health Services     Health Technology    Industrial Services 
                   12                    47                      9 
  Non-Energy Minerals    Process Industries Producer Manufacturing 
                    7                    24                     31 
         Retail Trade   Technology Services         Transportation 
                   23                    50                     15 
            Utilities 
                   31 
```

Thus, we can see how many stocks are part of each one of the 19 sectors.

We can sum them to confirm that they add up to 502.

```r
sum(table(sp500$Sector))
```

```
[1] 503
```

This completes our review of the Sector variable.

9.3.6 Stock Ratings

In the data, the S&P500 shares have Technical Ratings such as {Buy, Sell, ..}. Since each Stock has a unique Technical Rating, it makes sense to model the data column Rating as a factor() variable.

```r
sp500$Rating <- as.factor(sp500$Rating)
```

We confirm that Rating is now modelled as a factor variable, by running the str() function.

```r
str(sp500$Rating)
```

```
 Factor w/ 5 levels "Buy","Neutral",..: 3 1 1 3 3 3 5 3 1 3 ...
```

We can use the levels() function to review the different levels it can take.

```
levels(sp500$Rating)
```

```
[1] "Buy"         "Neutral"     "Sell"        "Strong Buy"  "Strong Sell"
```

The table() function allows us to count how many stocks have each Rating.

```
table(sp500$Rating)
```

```

        Buy     Neutral        Sell  Strong Buy Strong Sell 
        154          60         212          37          40 
```

Thus, we can see how many stocks have ratings ranging from "Strong Sell" to "Strong Buy". This completes our review of Technical Rating.

9.3.7 Summary

We believe this dataset of S&P500 shares is now ready for futher analysis. We end this stage of our analysis in this chapter, by running the str() function to review the data columns.

```
str(sp500)
```

```
spc_tbl_ [503 x 36] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
 $ Date                    : chr [1:503] "7/25/2023" "7/25/2023" "7/25/2023" "7/25/2023" ...
 $ Stock                   : chr [1:503] "A" "AAL" "AAP" "AAPL" ...
 $ StockName               : chr [1:503] "Agilent Technologies, Inc." "American Airlines Group,
 $ Sector                  : Factor w/ 19 levels "Commercial Services",..: 11 18 16 7 11 6 11 9
 $ Industry                : chr [1:503] "Medical Specialties" "Airlines" "Specialty Stores" "Te
 $ MarketCap               : num [1:503] 3.73e+10 1.14e+10 4.20e+09 3.04e+12 2.53e+11 ...
 $ Price                   : num [1:503] 126.4 17.5 70.7 193 143.6 ...
 $ Low52Wk                 : num [1:503] 113.3 11.7 63.6 124.2 131 ...
 $ High52Wk                : num [1:503] 160 19.1 212 198 168 195 116 81.9 328 539 ...
 $ ROE                     : num [1:503] 24.8 NA 14.6 146 51.1 389 NA 14.8 30.7 33.7 ...
 $ ROA                     : num [1:503] 12.7 3.9 3.35 27.6 5.43 2.79 NA 4.98 14.9 17.9 ...
```

```
$ ROIC                : num [1:503] 16.51 8.01 6.17 57.18 9.9 ...
$ GrossMargin         : num [1:503] 54.1 23.8 43.8 43.2 72.2 ...
$ OperatingMargin     : num [1:503] 23.78 9.39 5.63 29.16 41.07 ...
$ NetMargin           : num [1:503] 19.19 4.98 3.61 24.49 13.3 ...
$ PE                  : num [1:503] 27.86 4.78 10.5 32.8 33.82 ...
$ PB                  : num [1:503] 7.04 NA 1.56 60.73 14.73 ...
$ EVEBITDA            : num [1:503] 19.5 5.71 8.8 25 9.64 12.9 NA NA 17.3 33.4 ...
$ EBITDA              : num [1:503] 1.97e+09 7.16e+09 9.21e+08 1.24e+11 3.18e+10 ...
$ EPS                 : num [1:503] 4.54 3.67 6.72 5.89 4.25 ...
$ EBITDA_YOY          : num [1:503] 10.52 1074.1 -16 -5.36 10.6 ...
$ EBITDA_QYOY         : num [1:503] 8.2 72.2 -39.01 -4.58 11.68 ...
$ EPS_YOY             : num [1:503] 9.17 NA -25.21 -4.33 -39.11 ...
$ EPS_QYOY            : num [1:503] 11.69944 154.13308 -68.36829 -0.00656 -94.89037 ...
$ PFCF                : num [1:503] 31.74 7.88 NA 31.38 10.84 ...
$ FCF                 : num [1:503] 11.81 NA -100.23 -7.85 6.68 ...
$ FCF_QYOY            : num [1:503] 55.7078 -10.2542 -176.1352 -0.0312 -15.3392 ...
$ DebtToEquity        : num [1:503] 0.473 NA 1.582 1.763 4.678 ...
$ CurrentRatio        : num [1:503] 2.37 0.749 1.244 0.94 0.96 ...
$ QuickRatio          : num [1:503] 1.708 0.656 0.238 0.878 0.821 ...
$ DividendYield       : num [1:503] 0.723 NA 1.428 0.497 4.163 ...
$ DividendsPerShare_YOY: num [1:503] 8.25 NA 84.62 5.88 7.53 ...
$ PS                  : num [1:503] 5.538 0.235 0.384 7.992 4.399 ...
$ Revenue_YOY         : num [1:503] 7.8597 29.9089 1.4153 -0.2544 0.0282 ...
$ Revenue_QYOY        : num [1:503] 6.85 4.72 1.29 -2.51 -9.7 ...
$ Rating              : Factor w/ 5 levels "Buy","Neutral",..: 3 1 1 3 3 3 5 3 1 3 ...
- attr(*, "spec")=
 .. cols(
 ..   Date = col_character(),
 ..   Stock = col_character(),
 ..   Description = col_character(),
 ..   Sector = col_character(),
 ..   Industry = col_character(),
 ..   `Market Capitalization` = col_double(),
 ..   Price = col_double(),
 ..   `52 Week Low` = col_double(),
 ..   `52 Week High` = col_double(),
 ..   `Return on Equity (TTM)` = col_double(),
 ..   `Return on Assets (TTM)` = col_double(),
 ..   `Return on Invested Capital (TTM)` = col_double(),
 ..   `Gross Margin (TTM)` = col_double(),
 ..   `Operating Margin (TTM)` = col_double(),
 ..   `Net Margin (TTM)` = col_double(),
 ..   `Price to Earnings Ratio (TTM)` = col_double(),
```

```
 ..   `Price to Book (FY)` = col_double(),
 ..   `Enterprise Value/EBITDA (TTM)` = col_double(),
 ..   `EBITDA (TTM)` = col_double(),
 ..   `EPS Diluted (TTM)` = col_double(),
 ..   `EBITDA (TTM YoY Growth)` = col_double(),
 ..   `EBITDA (Quarterly YoY Growth)` = col_double(),
 ..   `EPS Diluted (TTM YoY Growth)` = col_double(),
 ..   `EPS Diluted (Quarterly YoY Growth)` = col_double(),
 ..   `Price to Free Cash Flow (TTM)` = col_double(),
 ..   `Free Cash Flow (TTM YoY Growth)` = col_double(),
 ..   `Free Cash Flow (Quarterly YoY Growth)` = col_double(),
 ..   `Debt to Equity Ratio (MRQ)` = col_double(),
 ..   `Current Ratio (MRQ)` = col_double(),
 ..   `Quick Ratio (MRQ)` = col_double(),
 ..   `Dividend Yield Forward` = col_double(),
 ..   `Dividends per share (Annual YoY Growth)` = col_double(),
 ..   `Price to Sales (FY)` = col_double(),
 ..   `Revenue (TTM YoY Growth)` = col_double(),
 ..   `Revenue (Quarterly YoY Growth)` = col_double(),
 ..   `Technical Rating` = col_character()
 .. )
- attr(*, "problems")=<externalptr>
```

10 Continuous Data (1 of 3)

July 23, 2023

10.1 Univariate Continuous Data

1. Reading Data and Attaching Data to Memory

```
data(mtcars)
attach(mtcars)
```

10.2 Measures of Central Tendency

2. In R, we can summarize continuous data using descriptive statistics such as measures of central tendency (mean, median, and mode).
3. Measure the mean and median of the `wt` of all the cars in the dataframe `mtcars`

```
# Mean of wt in the mtcars dataframe
mean(mtcars$wt)
```

```
[1] 3.21725
```

```
# Median of wt in the mtcars dataframe
median(mtcars$wt)
```

```
[1] 3.325
```

4. In the above code, we calculate the mean and median of the mpg column using the `mean()` and `median()` functions, respectively.
5. To calculate the mode of the mpg column, we first load the `modeest` package using the `library()` function, and then use the `mfv()` function to compute the mode.

```r
# Mode of wt in the mtcars dataframe
library(modeest)
mfv(mtcars$mpg) # Mode
```

```
[1] 10.4 15.2 19.2 21.0 21.4 22.8 30.4
```

6. Note that the mtcars dataset contains continuous data, and so it does not have a well-defined mode in the traditional sense. The `mfv()` function computes the mode using a kernel density estimator, which may not always correspond to a single value in the dataset.

10.3 Measures of Variability

1. In R, we can calculate measures of variability (range, interquartile range, variance, and standard deviation).
2. To calculate these statistics, we can use built-in functions in R such as `range()`, `IQR()`, `var()`, and `sd()`.

```r
# Standard Deviation of wt in the mtcars dataframe
sd(mtcars$wt)
```

```
[1] 0.9784574
```

```r
# Variance of wt in the mtcars dataframe
var(mtcars$wt)
```

```
[1] 0.957379
```

```r
# Range of wt in the mtcars dataframe
range(mtcars$wt)
```

```
[1] 1.513 5.424
```

```
# Inter-Quartile Range of wt in the mtcars dataframe
IQR(mtcars$wt)
```

```
[1] 1.02875
```

3. Note that the range() function returns the minimum and maximum values in the dataset, while the IQR() function returns the difference between the 75th and 25th percentiles.

10.4 Other functions

```
# Minimum wt in the mtcars dataframe
min(mtcars$mpg)
```

```
[1] 10.4
```

```
# Maximum wt in the mtcars dataframe
max(mtcars$mpg)
```

```
[1] 33.9
```

10.5 Summarizing a data column

10.5.1 `summary()`

1. Display a summary of `mpg` in the dataframe mtcars using `summary()`

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

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

10.5.2 `describe()`

2. Display a summary of the `mpg` in the dataframe mtcars using `describe()`

```r
library(psych)
```

```
Registered S3 method overwritten by 'psych':
  method         from
  plot.residuals rmutil
```

```r
describe(mtcars$mpg)
```

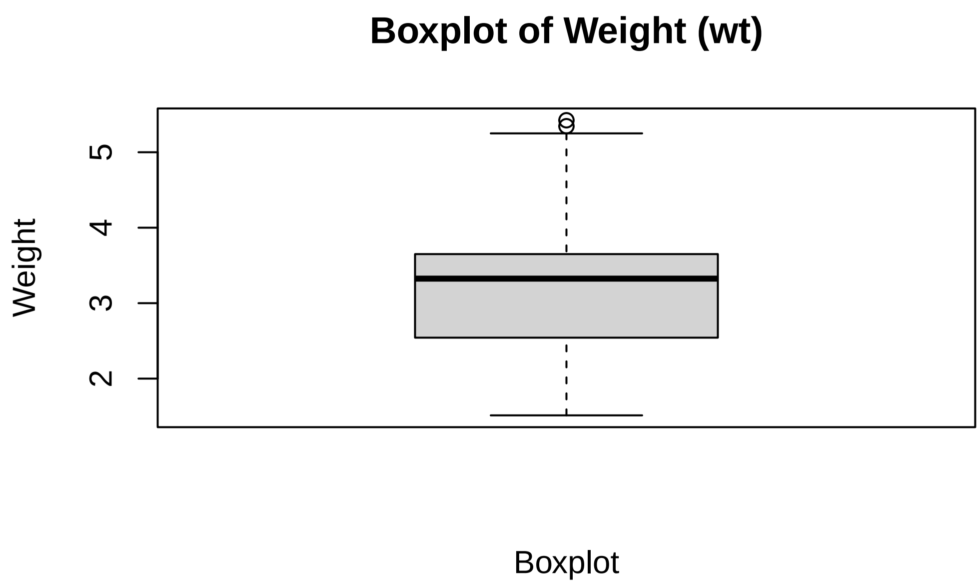
```
   vars  n  mean   sd median trimmed  mad  min  max range skew kurtosis   se
X1    1 32 20.09 6.03   19.2    19.7 5.41 10.4 33.9  23.5 0.61    -0.37 1.07
```

10.6 Visualizing Univariate Continuous Data

10.7 Boxplot

1. A boxplot is a graphical representation of the distribution of continuous data.
2. Display the Boxplot of the `wt` of the cars in the `mtcars` dataset

```r
boxplot(mtcars$wt,
        xlab = "Boxplot",
        ylab = "Weight",
        main = "Boxplot of Weight (wt)"
       )
```



3. The resulting boxplot will display the median, quartiles, and any outliers in the data.
4. The box represents the interquartile range, which contains the middle 50% of the data.
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.

10.8 Violin plot

1. A violin plot is similar to a boxplot, but instead of just showing the quartiles, it displays the full distribution of the data using a kernel density estimate.
2. We can create a violin plot in R using the violinplot() function from the vioplot package.

```r
# Load the vioplot package
library(vioplot)
```

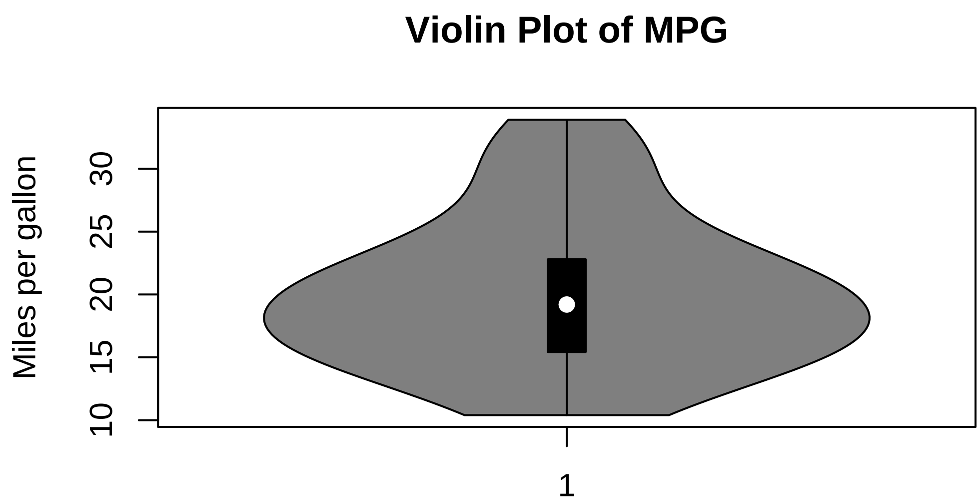
```
Loading required package: sm
```

```
Package 'sm', version 2.2-5.7: type help(sm) for summary information
```

```
Loading required package: zoo
```

```
Attaching package: 'zoo'


The following objects are masked from 'package:base':

    as.Date, as.Date.numeric
```

```r
# Create a violin plot of mpg column
vioplot(mtcars$mpg,
        main="Violin Plot of MPG",
        ylab="Miles per gallon")
```



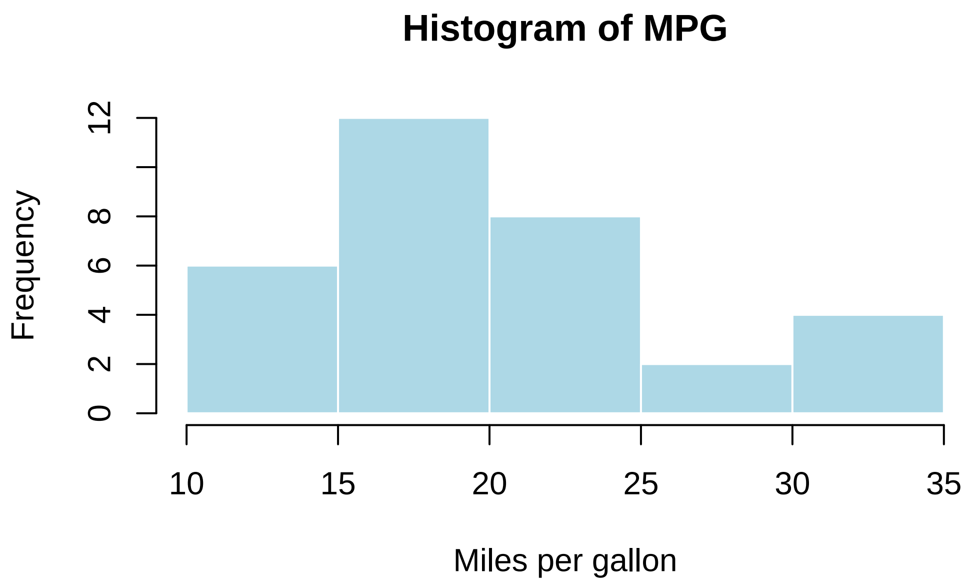
3. In the above code, we create a violin plot of the `mpg` column using the `vioplot()` function. The `main` argument is used to specify the title of the plot, and the `ylab` argument is used to specify the label for the y-axis.
4. The resulting plot will display the full distribution of the `mpg` data using a kernel density estimate, with thicker sections indicating a higher density of data points.
5. The plot also shows the median, quartiles, and any outliers in the data.

10.9 Histogram

1. A histogram is a plot that shows the frequency of each value or range of values in a dataset.

2. It can be useful for showing the shape of the distribution of the data. We can create a histogram in R using the `hist()` function.

```
# Create a histogram of mpg column
hist(mtcars$mpg,
     main="Histogram of MPG",
     xlab="Miles per gallon",
     col="lightblue",
     border="white")
```

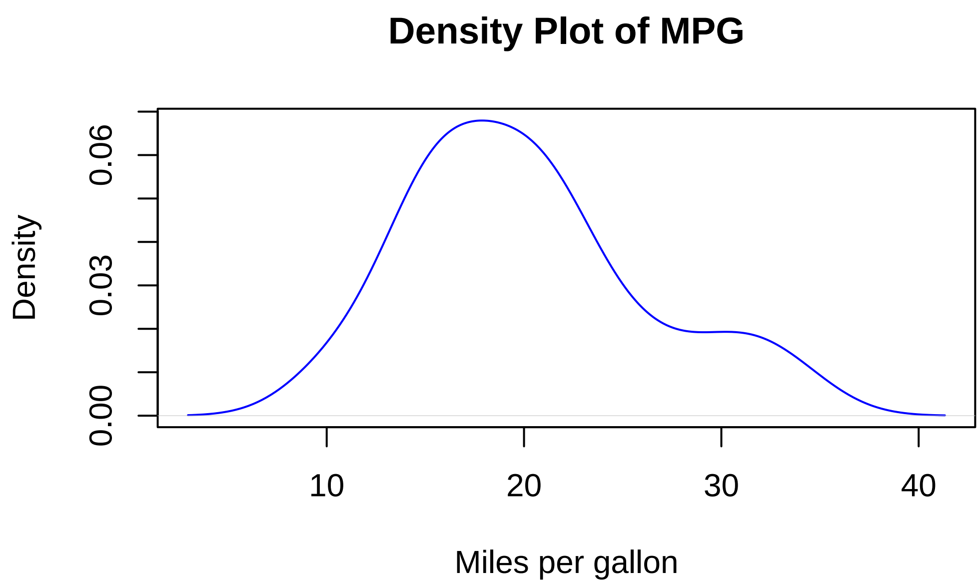


3. We create a histogram of the `mpg` column using the `hist()` function. The main argument is used to specify the title of the plot, and the xlab argument is used to specify the label for the x-axis.
4. The `col` argument is used to set the color of the bars in the histogram, and the border argument is used to set the color of the border around the bars.
5. The resulting histogram will display the frequency of `mpg` values in the dataset, with the bars representing the number of observations falling within a specific range of values.

10.10 Density plot

1. A density plot is similar to a histogram, but instead of displaying the frequency of each value, it shows the probability density of the data.

```
# Create a density plot of mpg column
plot(density(mtcars$mpg),
     main="Density Plot of MPG",
     xlab="Miles per gallon",
     col="blue")
```

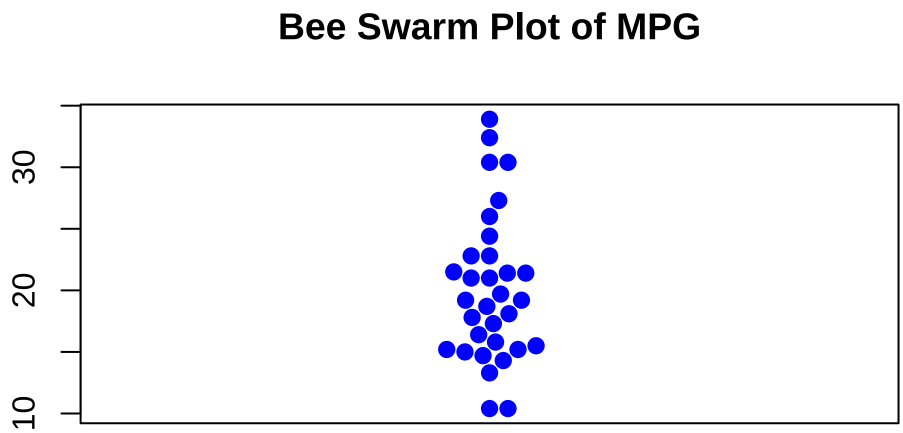


2. In the above code, we create a density plot of the mpg column using the `density()` function.
3. The `plot()` function is used to plot the resulting density object.
4. The `main` argument is used to specify the title of the plot, and the `xlab` argument is used to specify the label for the x-axis.
5. The `col` argument is used to set the color of the plot line.
6. The resulting plot will display the probability density of `mpg` values in the dataset, with the curve representing the distribution of the data.

10.11 Bee Swarm plot

1. A bee swarm plot is a plot that displays all of the individual data points along with a visual representation of their distribution.
2. It can be useful for displaying the distribution of small datasets.

```
# Load the beeswarm package
library(beeswarm)

# Create a bee swarm plot of mpg column
beeswarm(mtcars$mpg,
         main="Bee Swarm Plot of MPG",
         pch=16,
         cex=1.2,
         col="blue")
```



3. In the above code, we load the `beeswarm` package using the `library()` function.
4. We then create a bee swarm plot of the `mpg` column using the `beeswarm()` function.
5. The `main` argument is used to specify the title of the plot.
6. 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.
7. The `col` argument is used to set the color of the points.
8. The resulting plot will display the individual `mpg` 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.

11 Continuous Data (2 of 3)

July 23, 2023

11.1 Overview of Bivariate Continuous Data

1. Reading Data and Attaching Data to Memory

```
data(mtcars)
attach(mtcars)
```

11.2 Bivariate Continuous and Categorical data

1. Bivariate Relationship between Weight (wt) and Transmission (am)
2. Display a summary table showing the descriptive statistics of weight of the cars broken down by transmission (am=1 or am=0)

11.2.1 aggregate()

```
aggregate(mtcars$wt,
          by = list("am" = mtcars$am),
          mean)
```

```
  am        x
1  0 3.768895
2  1 2.411000
```

```
aggregate(mtcars$wt,
          by = list("am" = mtcars$am),
          sd)
```

```
  am         x
1  0 0.7774001
2  1 0.6169816
```

11.2.2 tapply()

```
tapply(mtcars$wt, mtcars$am, mean)
```

```
       0        1
3.768895 2.411000
```

```
tapply(mtcars$wt, mtcars$am, sd)
```

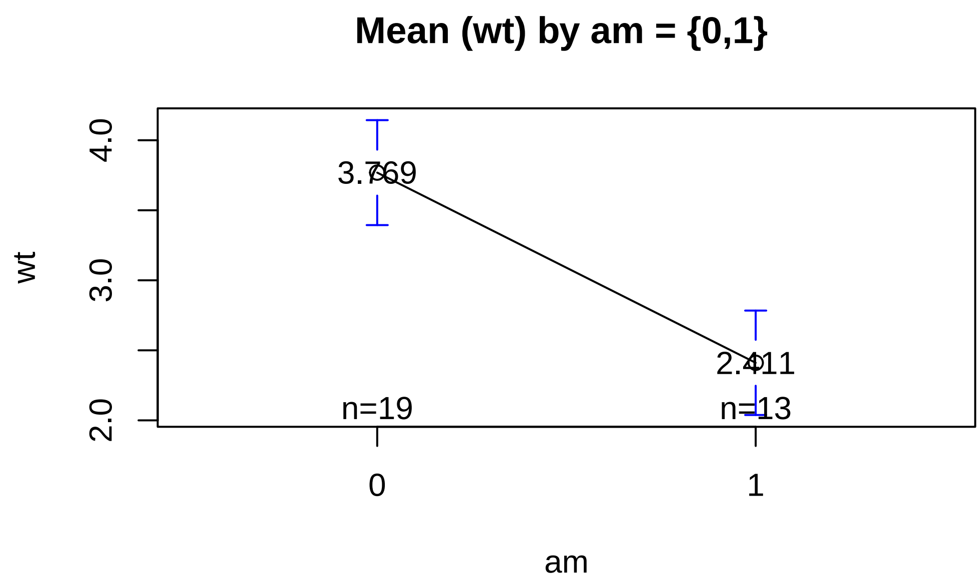
```
        0         1
0.7774001 0.6169816
```

11.3 Visualizing Means – mean plot showing the average weight of the cars, broken down by transmission (am=1 & am=0)

```
library(gplots)
```

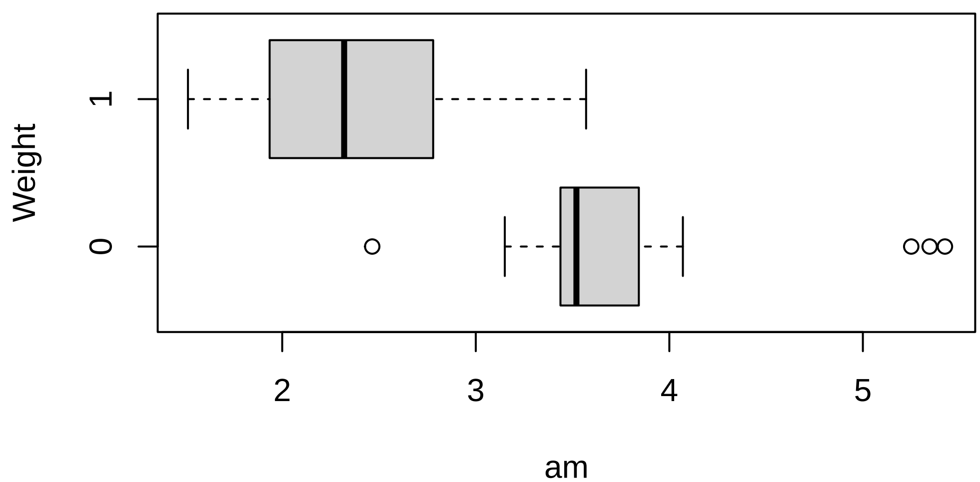
```

Attaching package: 'gplots'

The following object is masked from 'package:stats':

    lowess
```

```
plotmeans(wt ~ am
          ,data = mtcars
          ,mean.labels = TRUE
          ,digits=3
          ,main = "Mean (wt) by am = {0,1} "
          )
```



11.4 Visualizing Median using Box Plot – median weight of the cars broken down by transmission (am=1 & am=0)

```
boxplot(wt~am
        , xlab = "am"
        , ylab = "Weight"
        , horizontal = TRUE
        )
```



11.5 Bivariate Relationship between Weight (wt) and Cylinders (cyl)

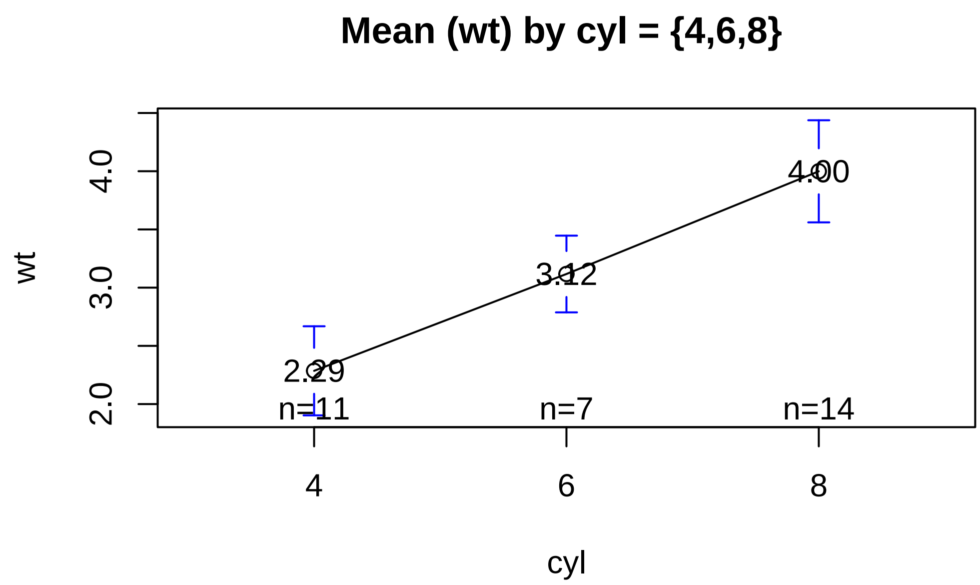
Display a summary table showing the mean weight of the cars broken down by cylinders (cyl=4,6,8)

```
psych::describeBy(wt, cyl)
```

```
 Descriptive statistics by group
group: 4
   vars  n mean   sd median trimmed  mad  min  max range skew kurtosis   se
X1    1 11 2.29 0.57    2.2    2.27 0.54 1.51 3.19  1.68  0.3    -1.36 0.17
------------------------------------------------------------
group: 6
   vars n mean   sd median trimmed  mad  min  max range  skew kurtosis   se
X1    1 7 3.12 0.36   3.21    3.12 0.36 2.62 3.46  0.84 -0.22    -1.98 0.13
------------------------------------------------------------
group: 8
   vars  n mean   sd median trimmed  mad  min  max range skew kurtosis  se
X1    1 14    4 0.76   3.76    3.95 0.41 3.17 5.42  2.25 0.99    -0.71 0.2
```

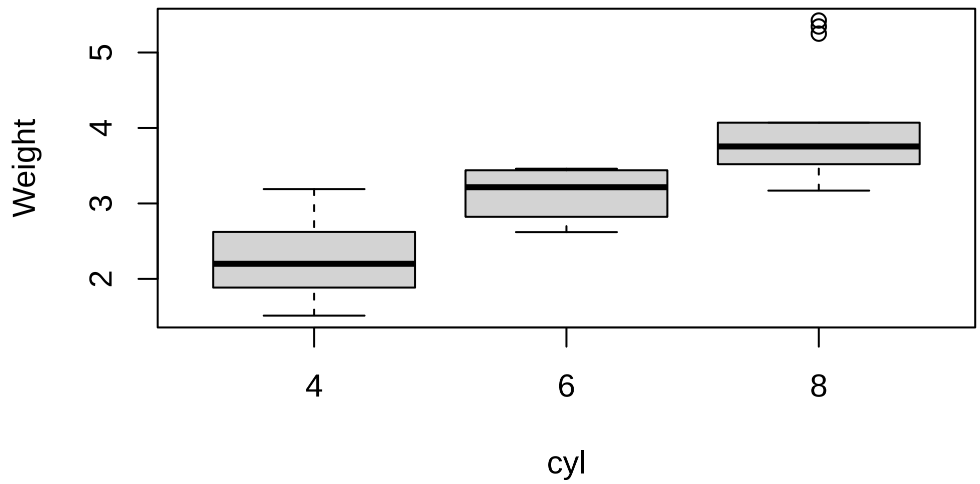
11.6 Show a mean plot showing the mean weight of the cars broken down by cylinders (cyl=4,6,8)

```
library(gplots)
plotmeans(wt ~ cyl,
          data = mtcars
          , mean.labels = TRUE
          , digits=2
          , main = "Mean (wt) by cyl = {4,6,8} ")
```



11.7 Show a box plot showing the median weight of the cars broken down by cylinders (cyl=4,6,8)

```r
boxplot(wt ~ cyl,
        xlab = "cyl", ylab = "Weight"
        )
```



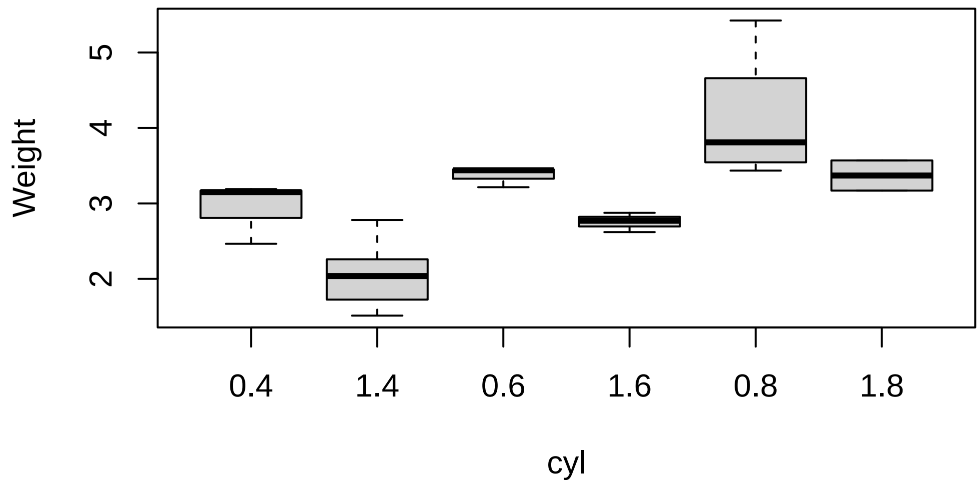
11.8 Distribution of Weight (wt) by Cylinders (cyl = {4,6,8}) and Transmisson Type (am = {0,1})

```
aggregate(wt,
          by = list("am" =am, "cyl" = cyl),
          mean)
```

```
  am cyl        x
1  0   4 2.935000
2  1   4 2.042250
3  0   6 3.388750
4  1   6 2.755000
5  0   8 4.104083
6  1   8 3.370000
```

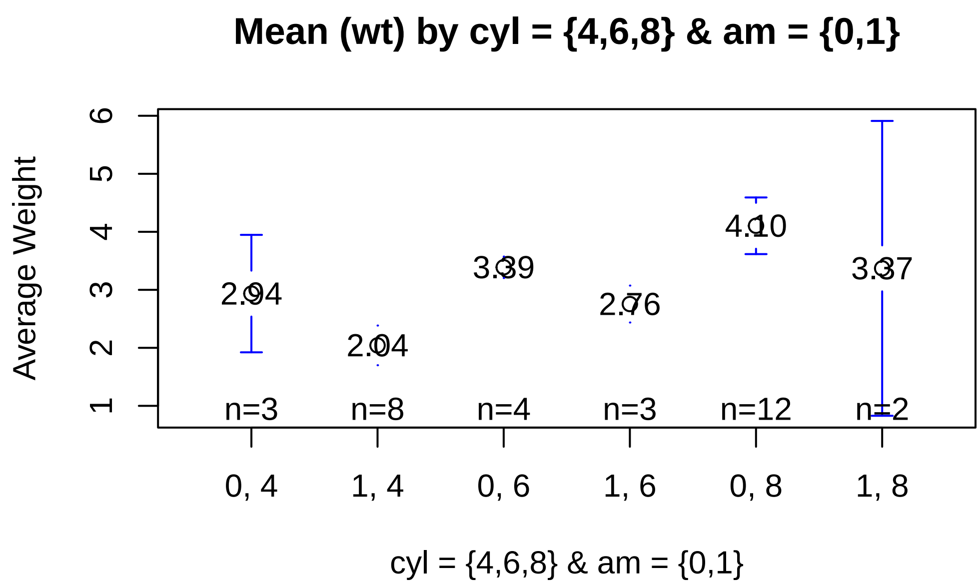
11.9 Visualization - Show a box plot showing the mean weight of the cars broken down by Transmission Type (am=1 & am=0) & cylinders (cyl=4,6,8)

```
boxplot(wt ~ am:cyl
        , xlab = "cyl"
        , ylab = "Weight"
        )
```



11.10 Visualization - Show a mean plot showing the mean weight of the cars broken down by Transmission Type (am=1 & am=0) & cylinders (cyl=4,6,8)

```
library(gplots)
plotmeans(wt ~ interaction(am, cyl, sep = ", ")
          , data = mtcars
          , mean.labels = TRUE
          , digits=2
          , connect = FALSE
          , main = "Mean (wt) by cyl = {4,6,8} & am = {0,1}"
          , xlab= "cyl = {4,6,8} & am = {0,1}"
          , ylab="Average Weight"
          )
```



12 Continuous Data (3 of 3)

July 23, 2023

12.1 Overview of Bivariate Continuous Data

1. Reading Data and Attaching Data to Memory

```
data(mtcars)
attach(mtcars)
```

12.2 Bivariate relationships between Continuous data

12.3 Scatterplot

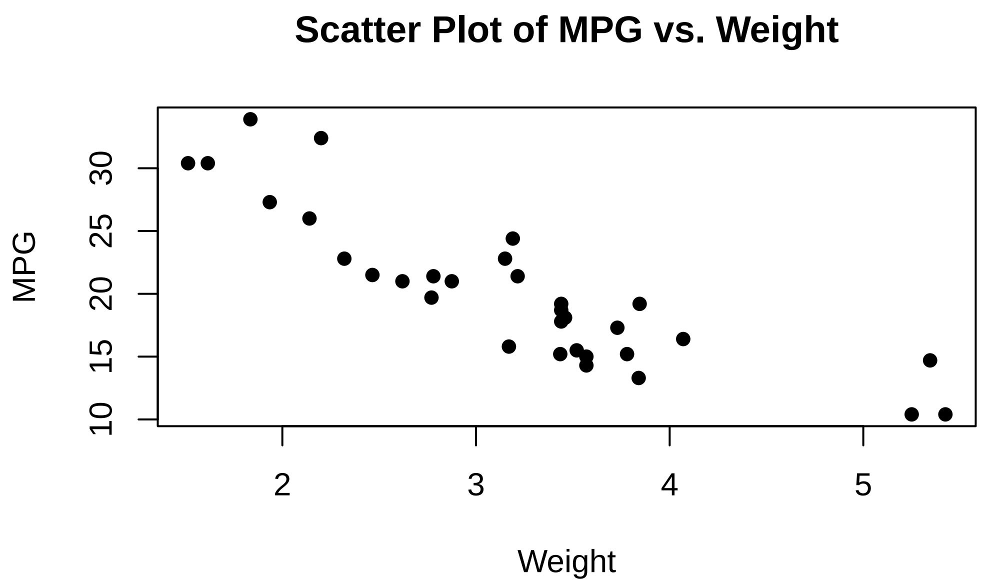
A scatter plot is a type of graph used to display the relationship between two continuous variables. It is a graphical representation of a bivariate distribution, where the values of two variables are plotted as points on a two-dimensional coordinate system.

A scatter plot can be used to identify trends, clusters, outliers, 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.

The mtcars data set in R is a built-in data set that contains data on various car models. To create a scatter plot of mpg (miles per gallon) against wt (weight) in the mtcars data set, you can use the following code:

12.3.1 Scatterplot using plot()

```
data(mtcars)
plot(mtcars$wt, mtcars$mpg, main = "Scatter Plot of MPG vs. Weight",
     xlab = "Weight", ylab = "MPG", pch = 16)
```



This code will first load the mtcars data set, then 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, and the pch argument changes the shape of the points to a solid circle. The resulting scatter plot will show the relationship between `mpg` and `wt` in the `mtcars` data set.

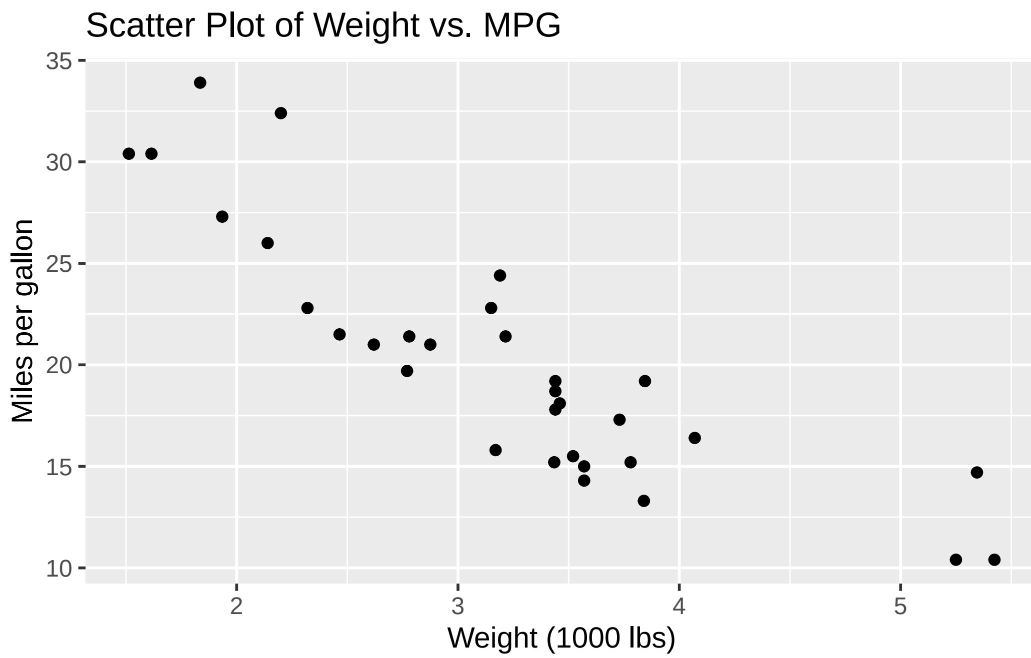
12.3.2 Scatterplot using ggplot2

```
# Load the ggplot2 package
library(ggplot2)
```

```
Attaching package: 'ggplot2'


The following object is masked from 'mtcars':
```

mpg

```
# Create the scatter plot
ggplot(mtcars, aes(x = wt, y = mpg)) +
  geom_point() +
  xlab("Weight (1000 lbs)") +
  ylab("Miles per gallon") +
  ggtitle("Scatter Plot of Weight vs. MPG")
```

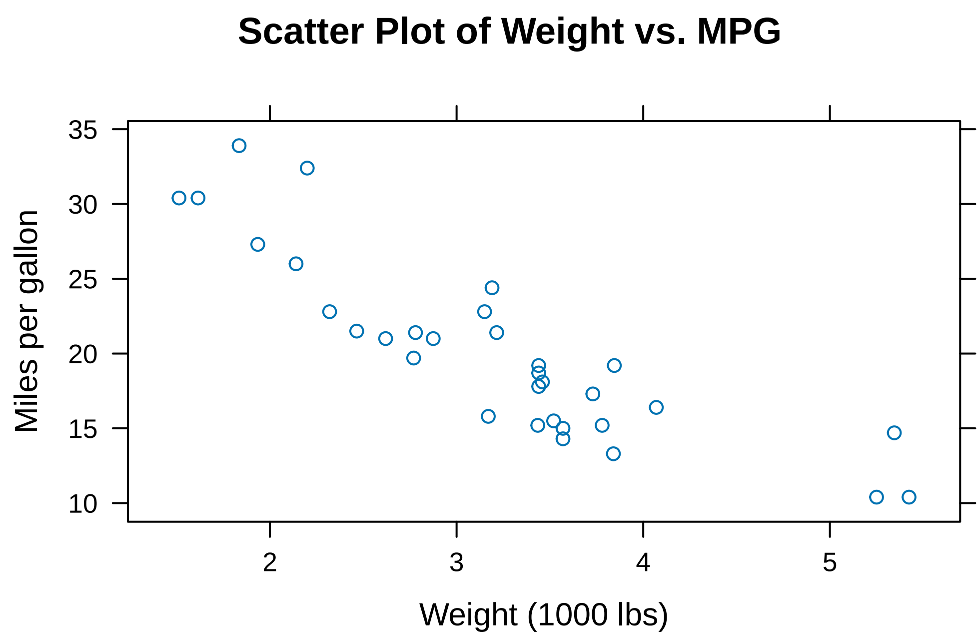


This code creates a scatter plot of the wt variable (weight in 1000 lbs) on the x-axis and the mpg variable (miles per gallon) on the y-axis. The geom_point() function is used to add the points to the plot, and xlab(), ylab(), and ggtitle() are used to add axis labels and a plot title, respectively. You can adjust the aesthetics of the plot, such as the color and size of the points, by adding additional arguments to the geom_point() function.

12.3.3 Scatterplot using Lattice

```
# Load the Lattice package
library(lattice)

# Create the scatter plot
```

```r
xyplot(mpg ~ wt, data = mtcars,
       xlab = "Weight (1000 lbs)",
       ylab = "Miles per gallon",
       main = "Scatter Plot of Weight vs. MPG")
```



This code creates a scatter plot of the wt variable (weight in 1000 lbs) on the x-axis and the mpg variable (miles per gallon) on the y-axis using the xyplot() function. The data argument specifies the data frame to use, and xlab, ylab, and main are used to add axis labels and a plot title, respectively. You can also add additional arguments to adjust the aesthetics of the plot, such as the size and color of the points or the type of line connecting the points, depending on your data and preferences.

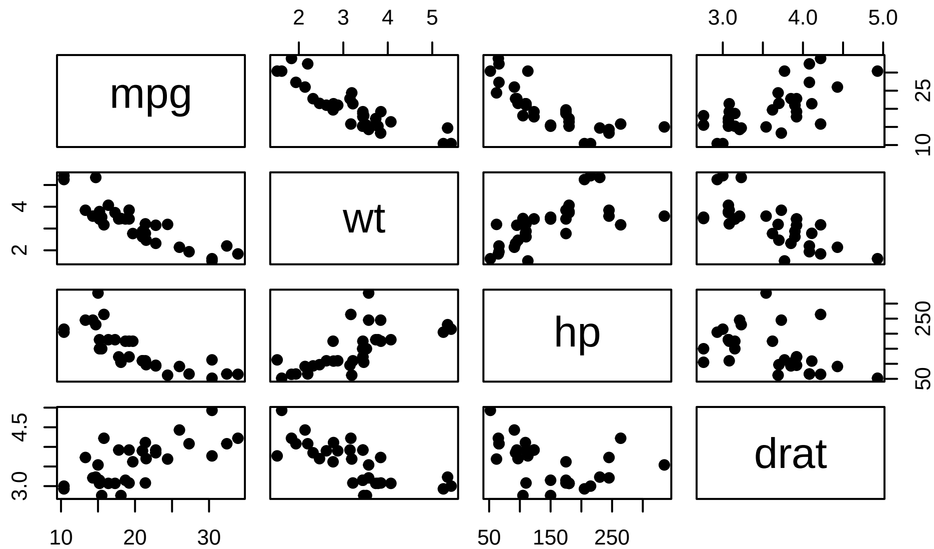
12.4 Scatterplot Matrix

A scatter plot matrix (also called a pairs plot or a SPLOM) is a graphical display of pairwise scatter plots of a set of variables. In a scatter plot matrix, each variable in the dataset is plotted against every other variable in a matrix format. This allows us to visualize the relationships between pairs of variables and explore potential patterns or trends in the data.

A scatter plot matrix is particularly useful for exploring multivariate datasets, as it allows us to quickly identify which pairs of variables may be strongly correlated, which may have weak or no correlation, and which may exhibit nonlinear relationships. It can also be used to identify outliers or unusual observations, and to visualize clusters or groups of observations based on patterns in the scatter plots.

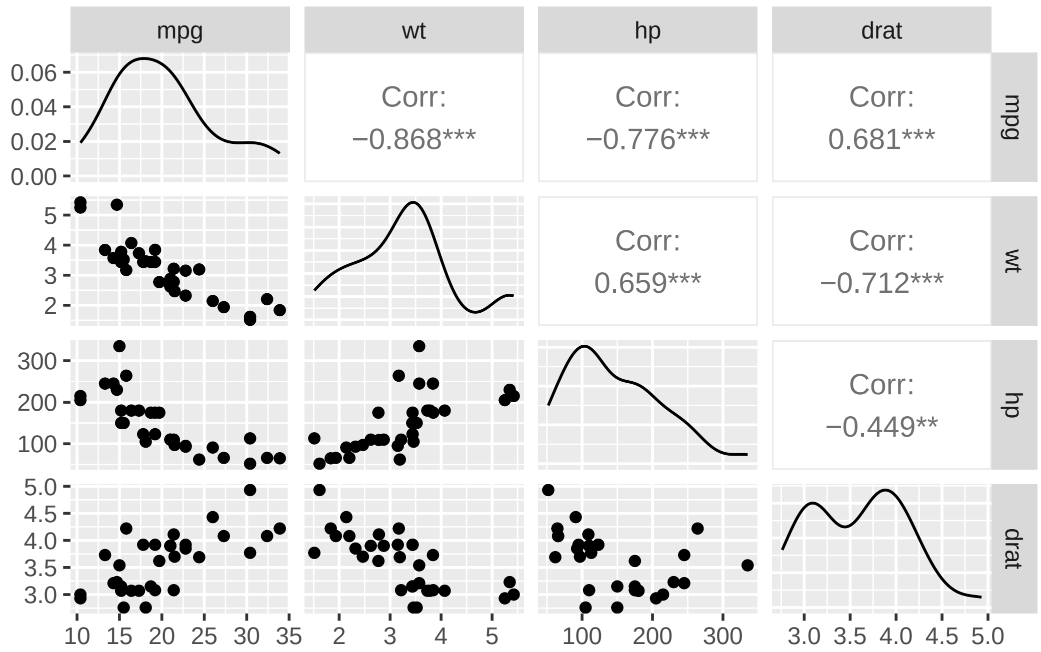
12.4.1 Scatterplot Matrix Using pairs()

```
# scatter plot matrix for mpg, wt, hp, drat
pairs(mtcars[,c("mpg","wt","hp","drat")], pch = 19)
```



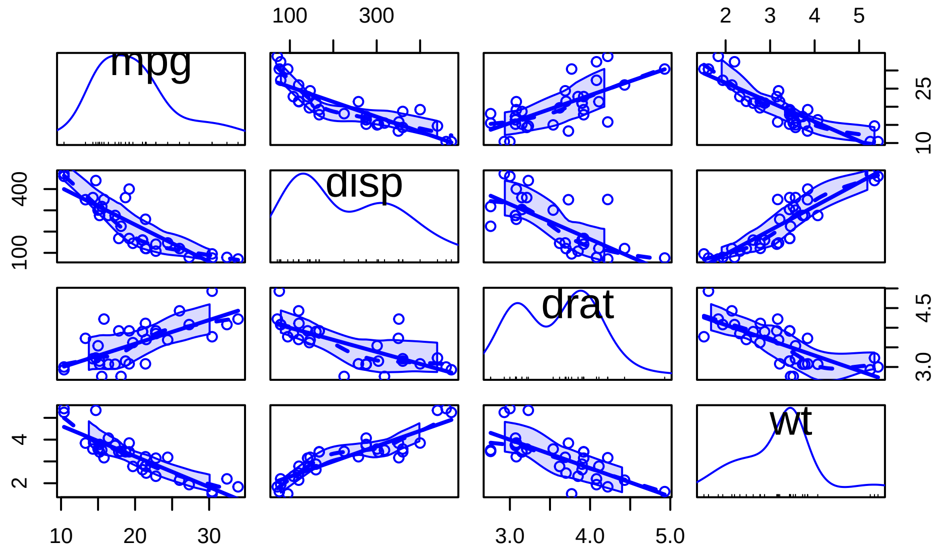
12.4.2 Scatterplot Matrix Using ggpairs()

```
# Load the GGally package
library(GGally)

# Create a scatterplot matrix using ggpairs()
ggpairs(mtcars[,c("mpg","wt","hp","drat")])
```



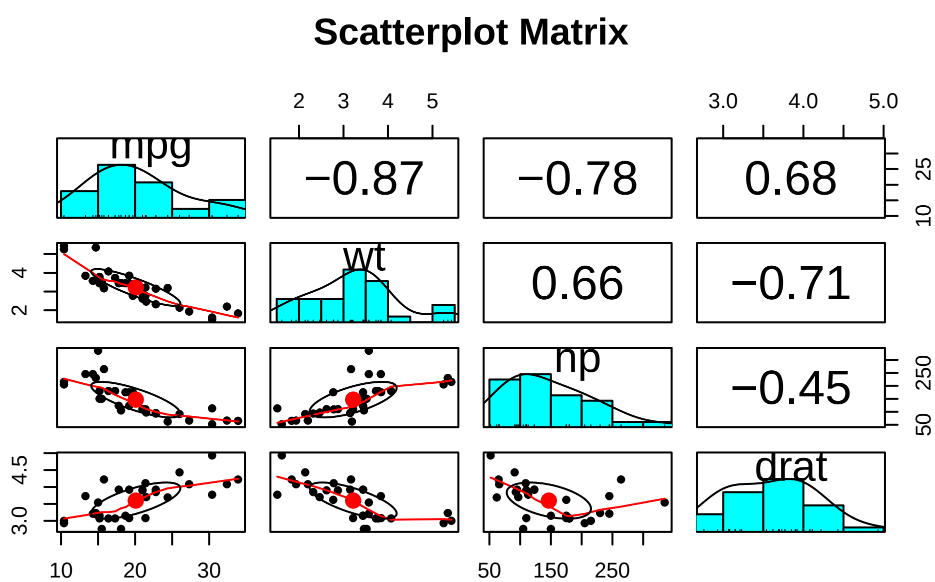
12.4.3 Scatterplot Matrix Using scatterplotMatrix()

```
# Load the car package
library(car)

# Create a scatterplot matrix using scatterplotMatrix()
scatterplotMatrix(~ mpg + disp +drat +wt,
                  data = mtcars, col = c("blue", "red"))
```



12.4.4 Scatterplot Matrix Using pairs.panels()

```
# Load the psych package
library(psych)

# Create a scatterplot matrix using pairs.panels()
pairs.panels(mtcars[,c("mpg","wt","hp","drat")],
             main = "Scatterplot Matrix")
```

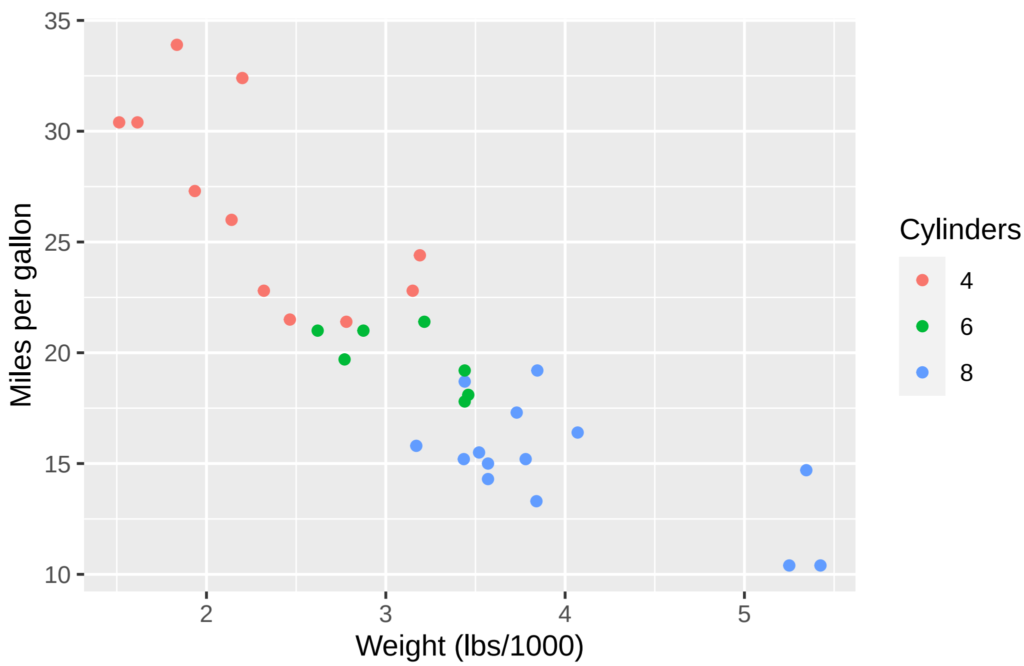


12.5 Scatterplots broken down by Categorical Variables

12.5.1 Scatterplot with colored by Categorical Variable Using ggplot()

This will create a scatterplot of miles per gallon (mpg) against weight, with each point colored according to the number of cylinders in the engine (cyl).

```
# Load the ggplot2 package
library(ggplot2)

# Create a scatterplot of mpg vs. wt, colored by cyl
ggplot(mtcars, aes(x = wt, y = mpg, color = factor(cyl))) +
  geom_point() +
  labs(x = "Weight (lbs/1000)", y = "Miles per gallon") +
  scale_color_discrete(name = "Cylinders")
```



12.5.2 Scatterplot with broken down by 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).

```r
# Load the ggplot2 package
library(ggplot2)

# Create a scatterplot matrix using ggplot()
ggplot(mtcars, aes(x = mpg, y = disp)) +
  geom_point() +
  facet_grid(. ~ cyl)
```