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

2023-07-04

Table of contents

Preface

Version 49

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 for the researcher 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 Univariate and Bivariate Analysis of data and the corresponding Data Visualization.
- We cover some basic Multivariate Analysis.
- We emphasize Insight Generation.

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

We further illustrate how to use R programming 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 the Return on Equity , Return on Assets, Return on Invested Capital of S&P500 shares.

1 Getting Started

Version 49

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 install 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 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

Version 49

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()
```

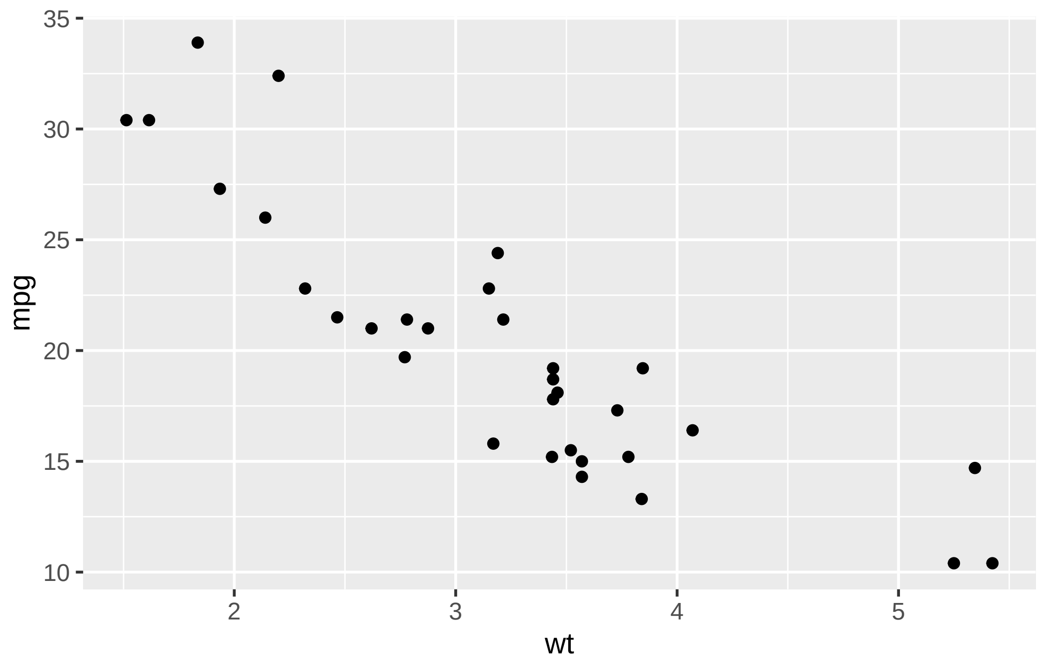


Figure 2.1: Scatterplot of Car Mileage with Car Weight

2.4.1 Getting help

To seek assistance with an R package, you can explore the following avenues:

1. Documentation: Most R packages come with comprehensive documentation that explains the package's functions, datasets, and provides usage examples. You can access the documentation by using the **`help()`** function or typing **`?package_name`** in the R console, where "package_name" is the name of the specific package you want to learn about.

2. Integrated help system: R has an integrated help system that offers documentation and demonstrations for functions and packages. In the R console, you can access the help system by typing **`help(topic)`** or **`?topic`**, where "topic" represents the name of the function or package you need 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 Inbuilt R functions

3.1 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: R can perform basic arithmetic operations such as addition (+), subtraction (-), multiplication (*), and division (/).

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

```
[1] 11
```

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

```
[1] 28
```

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

```
# exponentiation
2^6
```

```
[1] 64
```

```
# 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=-5
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
```

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

```
[1] 4
```

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

```
[1] 3
```

```
# 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.

```
x <- c(0, 1, 1, 2, 3, 5, 8)   # create a vector of 7 Fibonacci numbers
length(x) # count how many numbers do we have
```

```
[1] 7
```

```
mean(x)   # calculate the mean
```

```
[1] 2.857143
```

```
median(x) # calculate the median
```

```
[1] 2
```

```r
sd(x)     # calculate the standard deviation
```

```
[1] 2.794553
```

```r
y <- c(1, 2, 3, 4, 5, 6, 7) # create a new vector of positive integers
cor(x,y)  # calculate the correlation between x and y
```

```
[1] 0.938668
```

3.2 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
# use the assignment operator <-
sales <- 9
# alternately, use =
sales = 9
```

2. It is possible to use <- or = for variable assignments.
3. R is case-sensitive. This means that `Sales` is different from `sales`
4. It is possible to perform some operations with it.

```r
# multiply sales by 2
2 * sales
```

```
[1] 18
```

5. We can change the value stored in a variable

```r
# change the value
sales <- 15
# display the revised sales
sales
```

```
[1] 15
```

6. The following R code creates two variables holding the sales and the price of a product and we can use them to compute the revenue.

```
# sales
sales <- 5

# price
price <- 7

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

```
[1] 35
```

4 Data Structures

4.1 Popular Data Structures

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**: Similar to a spreadsheet, a data frame is a two-dimensional table-like structure 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]

4.2 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]

4.2.1 Vectors in R

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

```r
# Read data into vectors
names <- c("Ashok", "Bullu", "Charu", "Divya")
ages <- c(72, 49, 46, 42)
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."

4.2.2 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:

```r
names[2]
```

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

This returns “Bullu”, the second element of the “names” 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 females’ 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. To calculate the sum of the “ages” vector, for instance, we can use:

```
sum(ages)
```

```
[1] 209
```

This returns 209, the sum of the four ages.

5. **Logical Operations:** We can perform logical operations on vectors, which are also executed element-by-element. To create a new vector titled “middle_age” that indicates whether each individual is 45 to 55 years old, for instance, we can use:

```r
middle_age <- (ages >= 45) & (ages <= 55)
middle_age
```

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

This generates a new vector with four elements containing logical values indicating whether each person is between 45 and 55 years of age.

To test whether any of the elements in the "ages" vector are greater than 50, we can use:

```r
any(ages > 50)
```

```
[1] TRUE
```

6. **Unique Values:** We can find the unique values in a vector using the "unique()" function. For example, to find the unique values in the "ages" vector, we can use:

```r
unique(ages)
```

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

7. **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(ages, decreasing = TRUE)
```

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

4.2.3 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.
6. **Variance**: The variance is another measure of the spread. It is square of the Standard Deviation.

```r
mean(ages)
```

```
[1] 52.25
```

```r
sd(ages)
```

```
[1] 13.47529
```

```r
var(ages)
```

```
[1] 181.5833
```

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

```r
median(ages)
```

```
[1] 47.5
```

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

```r
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.

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.

4.2.4 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
substr(names, 1, 3)
```

```
[1] "Ash" "Bul" "Cha" "Div"
```

This returns a new character vector containing the initial three letters of each name ("Ash", "Bul", "Cha", and "Div").

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

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

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

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

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

```r
toupper(names)
```

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

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

4. **Pattern Matching:** Using the `grep()` 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:

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

```
[1] 3 4
```

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

5. **Regular Expressions:** We can use regular expressions with the `grep()` and `grepl()` functions to search for patterns in the elements of a character vector. To find the names in the "names" vector that begin with the letter "C", for instance, we can use:

```r
grep("^C", names)
```

```
[1] 3
```

This returns a vector containing the indexes of the elements in “names” that begin with the letter “C.” [1]

4.3 References

[1]

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

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

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

5 Reading Data

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

5.1 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]

5.1.1 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
print(people)
```

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

5.2 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.

5.2.1 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  
```

5.2.2 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 ...
```

5.3 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.

5.3.1 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:

```
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:

```r
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]

5.3.2 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.

```r
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.

5.3.3 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.

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

5.3.4 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: `1nm688a3GsPM5cadJIwu6zj336WBaduglY9TSTUaM9jk`
- We can use the function `gsheet2tbl` in package `gsheet` to read the Google Sheet into a dataframe, as demonstrated in the following code.

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

prefix <- "https://docs.google.com/spreadsheets/d/"
sheetID <- "1nm688a3GsPM5cadJIwu6zj336WBaduglY9TSTUaM9jk"
suffix <- "/edit#gid=0"

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

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

```
No encoding supplied: defaulting to UTF-8.
```

- 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, and the `suffix` variable contains the URL's suffix.
- The `paste()` function is used to combine the `prefix`, `sheetID`, and `suffix` 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 `gf`.
- Once the preceding code is executed, the `gf` dataframe will contain the Google Sheet data, which can then be analyzed further in R.

5.3.5 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 gf.
- 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 presend in another Google Sheet.
library(gsheet)

prefix <- "https://docs.google.com/spreadsheets/d/"
sheetID <- "1F5KvFATcehrdJuGjYVqppNYC9hEKSww9rXYHCk2g6OA"
suffix <- "/edit#gid=0"

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

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

```
No encoding supplied: defaulting to UTF-8.
```

- We now have two dataframes named `tv` 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.
- The following code illusrates how to merge two dataframes:

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

- We now have a new dataframe named `M.df`, which contains the data got from merging the two dataframes `tv` and `gf`.

5.4 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:

```r
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
```

```r
# 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.

5.4.1 Converting a dataframe into a tibble

```
# 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.

5.4.2 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.

5.5 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

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/setwd.html

6 Exploring a Dataframe

Here's how you can explore the mtcars data frame:

View the data: Use the head() or tail() function to view the first or last few rows of the data frame, respectively. For example:

1. Find the dimensions (rows and columns) of the dataframe

```r
data(mtcars)
dim(mtcars)
```

```
[1] 32 11
```

2. Retrieve the names of the columns in the dataframe

```r
names(mtcars)
```

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

3. str()

to display the structure of the dataframe, including the data type and the first few rows.

```r
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. head()

to view the first few 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
```

5. tail()

to view the last few 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
```

6. summary()

to generate summary statistics for each column in the dataframe.

```
summary(mtcars)
```

```
      mpg             cyl             disp             hp       
 Min.   :10.40   Min.   :4.000   Min.   : 71.1   Min.   : 52.0  
 1st Qu.:15.43   1st Qu.:4.000   1st Qu.:120.8   1st Qu.: 96.5  
 Median :19.20   Median :6.000   Median :196.3   Median :123.0  
 Mean   :20.09   Mean   :6.188   Mean   :230.7   Mean   :146.7  
 3rd Qu.:22.80   3rd Qu.:8.000   3rd Qu.:326.0   3rd Qu.:180.0  
 Max.   :33.90   Max.   :8.000   Max.   :472.0   Max.   :335.0  
      drat             wt             qsec             vs        
 Min.   :2.760   Min.   :1.513   Min.   :14.50   Min.   :0.0000  
 1st Qu.:3.080   1st Qu.:2.581   1st Qu.:16.89   1st Qu.:0.0000  
 Median :3.695   Median :3.325   Median :17.71   Median :0.0000  
 Mean   :3.597   Mean   :3.217   Mean   :17.85   Mean   :0.4375  
 3rd Qu.:3.920   3rd Qu.:3.610   3rd Qu.:18.90   3rd Qu.:1.0000  
 Max.   :4.930   Max.   :5.424   Max.   :22.90   Max.   :1.0000  
       am              gear            carb      
 Min.   :0.0000   Min.   :3.000   Min.   :1.000  
 1st Qu.:0.0000   1st Qu.:3.000   1st Qu.:2.000  
 Median :0.0000   Median :4.000   Median :2.000  
 Mean   :0.4062   Mean   :3.688   Mean   :2.812  
 3rd Qu.:1.0000   3rd Qu.:4.000   3rd Qu.:4.000  
 Max.   :1.0000   Max.   :5.000   Max.   :8.000  
```

7. table()

to generate a frequency table for a categorical variable.

```
table(mtcars$cyl)
```

```

 4  6  8 
11  7 14 
```

8. unique()

to find unique values in a column of the dataframe.

```
unique(mtcars$cyl)
```

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

6.1 Logical operations

Here are some examples of logical operations functions in R using the mtcars dataset:

1. Subsetting based on a condition:

The logical expression [] and square bracket notation can be used to subset the mtcars dataset according to a criterion. For instance, to only choose the rows where the mpg is higher than 20:

```
# 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
```

2. The which() function:

The which() function returns the indexes of the vector's members that adhere to a predicate. To determine the indices of the rows where mpg is larger than 20 for instance:

```
# 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
```

3. The ifelse() function:

The ifelse() function applies a logical condition to a vector and returns a new vector with values depending on whether the condition is TRUE or FALSE. It is a vectorized version of the if-else statement. For instance, to add a new column called high mpg that shows whether or not the mpg value is more than 20:

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

4. The all() and any() functions:

If every element in a vector satisfies a logical criterion, the all() function returns TRUE; otherwise, it returns FALSE. If at least one element in a vector satisfies a logical criterion, the any() method returns TRUE; otherwise, it returns FALSE. To determine whether every value in the mpg column is larger than 20, for instance:

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

```
[1] FALSE
```

And to check if at least one value in the mpg column is greater than 20:

Check if any value in mpg column is greater than 20

```
any(mtcars$mpg > 20)
```

```
[1] TRUE
```

6.2 Creating new functions in R

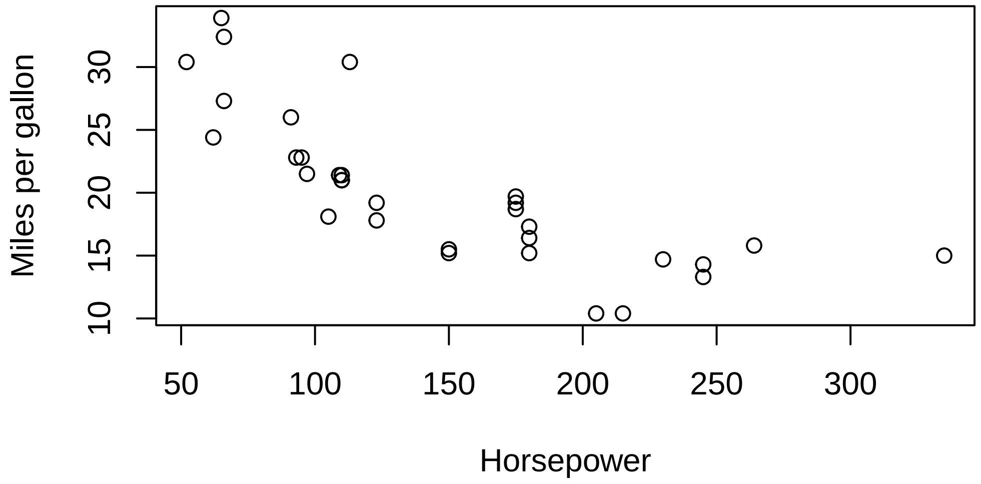
1. Function to calculate average mileage:

```
avg_mileage <- function(data) {
  mean(data$mpg)
}

# Usage
avg_mileage(mtcars) # Returns the average mileage of all cars in the dataset
```

```
[1] 20.09062
```

2. Function to plot a scatter plot of horsepower vs. miles per gallon:

```
plot_horsepower_vs_mpg <- function(data) {
  plot(data$hp, data$mpg, xlab = "Horsepower", ylab = "Miles per gallon")
}

# Usage
plot_horsepower_vs_mpg(mtcars) # Plots a scatter plot of horsepower vs. miles per gallon
```



3. 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])
}

# Usage

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

```
[1] 26.66364
```

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

```
[1] 19.74286
```

4. Function to calculate average horsepower for cars with a specific number of gears:

```
avg_hp_by_gear <- function(data, gear) {
  mean(data$hp[data$gear == gear])
}

# Returns the average horsepower of cars with 3 gears
avg_hp_by_gear(mtcars, 3)
```

```
[1] 176.1333
```

```
# Returns the average horsepower of cars with 4 gears
avg_hp_by_gear(mtcars, 4)
```

```
[1] 89.5
```

7 Summarizing and Visualizing Categorical Data

7.1 Categorical Data

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)

7.2 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.

7.3 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."

7.3.1 `summary()`

```
summary(cyl)
```

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

7.4 4. Frequency Table for More than One Variable

7.4.1 4a) 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.

7.4.2 4b) 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.

7.4.3 4c) 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.

7.5 5. `prop.table` Proportions Table for One Variable

- 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.

7.6 6. `prop.table` Proportions Table for More than One Variable

```
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.

7.7 7. `round()`: This function is used to set the width of decimal numbers

```
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.

7.8 8. `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%.

7.9 9. Three Way Relationship

7.9.1 9a) table()

```r
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.

7.9.2 9b) 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.

7.9.3 9c) ftable()

```r
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.

7.10 10. Four Way Relationship

```r
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.

7.11 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

7.11.1 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 p̂ = 120/1000 = 0.12. To find the 95% confidence interval for the population proportion, we can use the formula:

p̂ ± 1.96 * √(p̂(1-p̂)/n)

= 0.12 ± 1.96 * √(0.12(0.88)/1000)

= 0.12 ± 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.

7.11.2 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.

7.11.3 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

7.11.4 Confidence Intervals for the Difference of Two Proportions in R

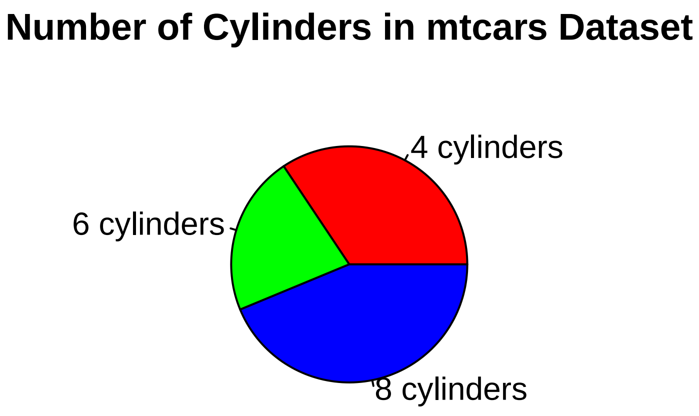
7.12 Visualization of Categorical Variable

7.13 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.

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

# 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"))
```

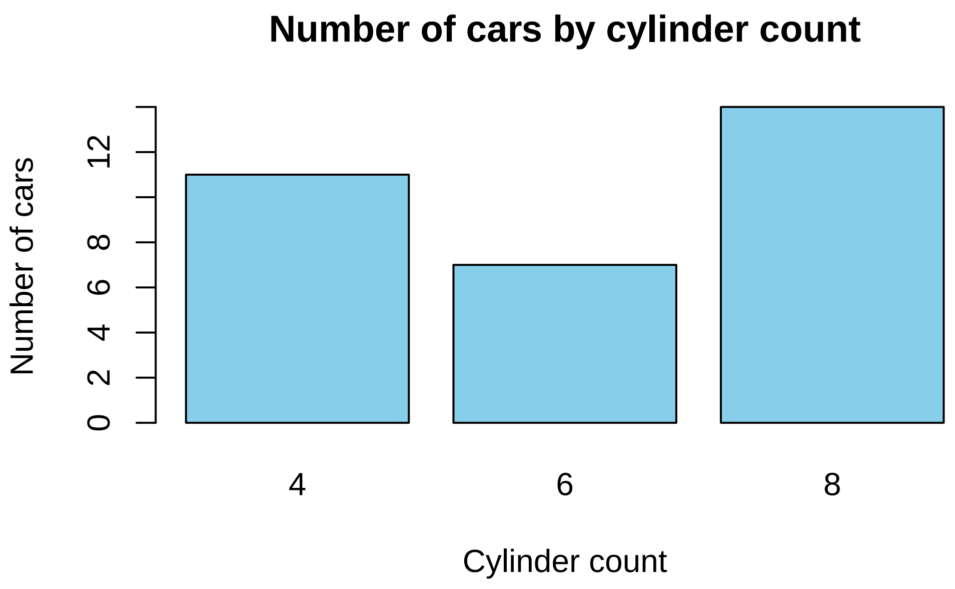


The occurrences of each value of the cyl variable in the mtcars dataset are counted in this code using the table() function, and the resulting table is saved as cyl counts. The cyl counts variable provides the data for the pie chart, which is subsequently created using the pie() function. The chart's 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.

7.14 Barplot for categorical data in R

7.14.1 Barplot for Univariate Case

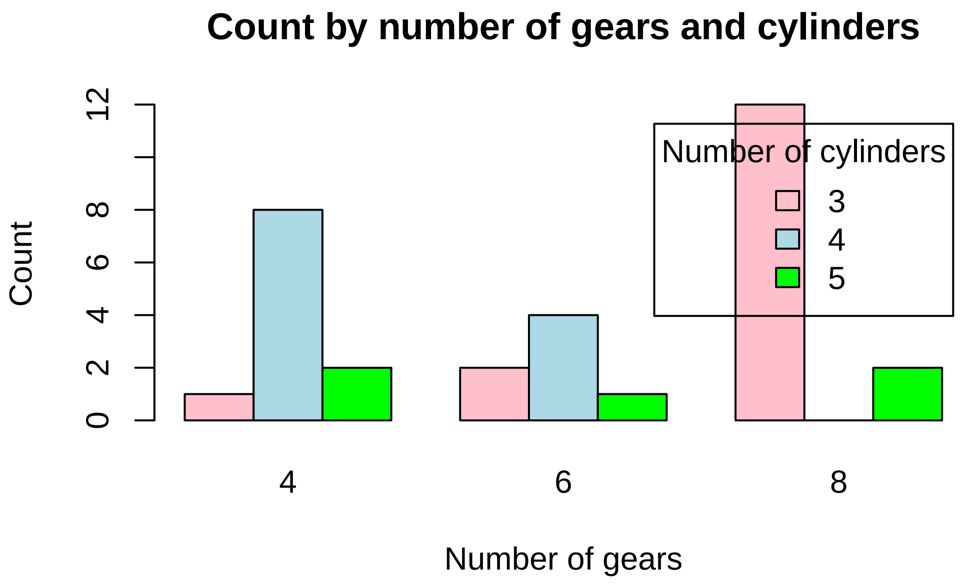
```r
# Load the mtcars dataset
data(mtcars)

# 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.

7.14.2 Barplot for Bivariate Case (Grouped Barchart)

```r
# 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"))
```

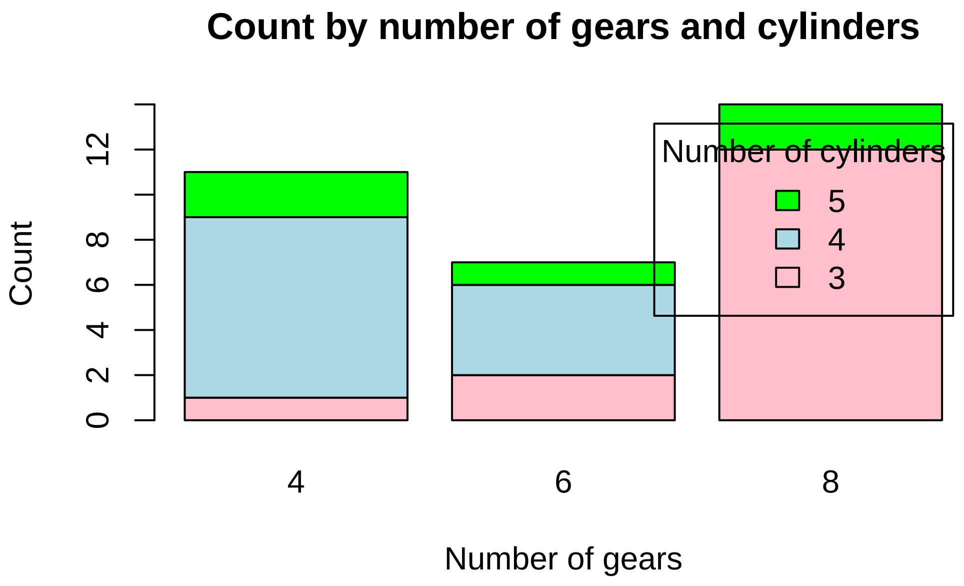


In this code, we first load the mtcars dataset. Then, we use the table() function to compute the counts by number of gears and number of cylinders. We store the result in a matrix called counts.

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”).

7.14.3 Barplot for Bivariate Case (Stacked Barchart)

```r
# 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.

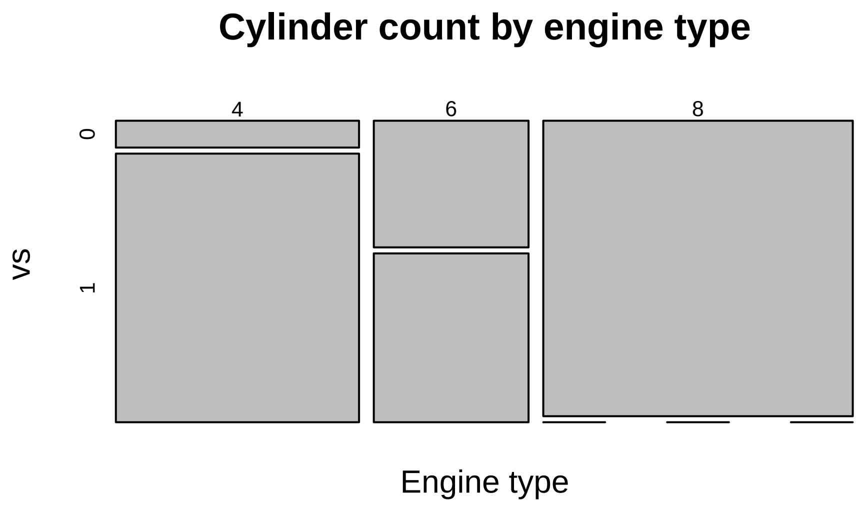
Finally, we use barplot() to create the plot. We pass the counts matrix as the first argument, and we set beside = FALSE to make sure that the bars are stacked on top of each other. We also set the colors of the bars using col, and we add labels to the plot using xlab, ylab, and main. We also add a legend 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”).

7.15 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.

```r
# 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.

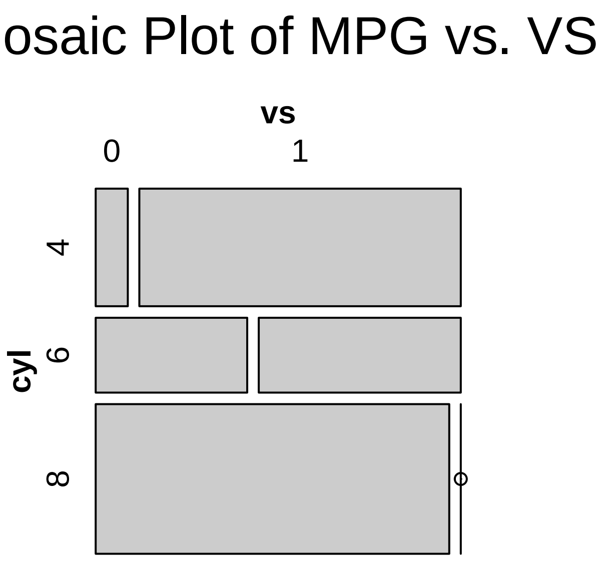
```r
# Load the mtcars dataset
data(mtcars)

# 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)
```

```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")
```



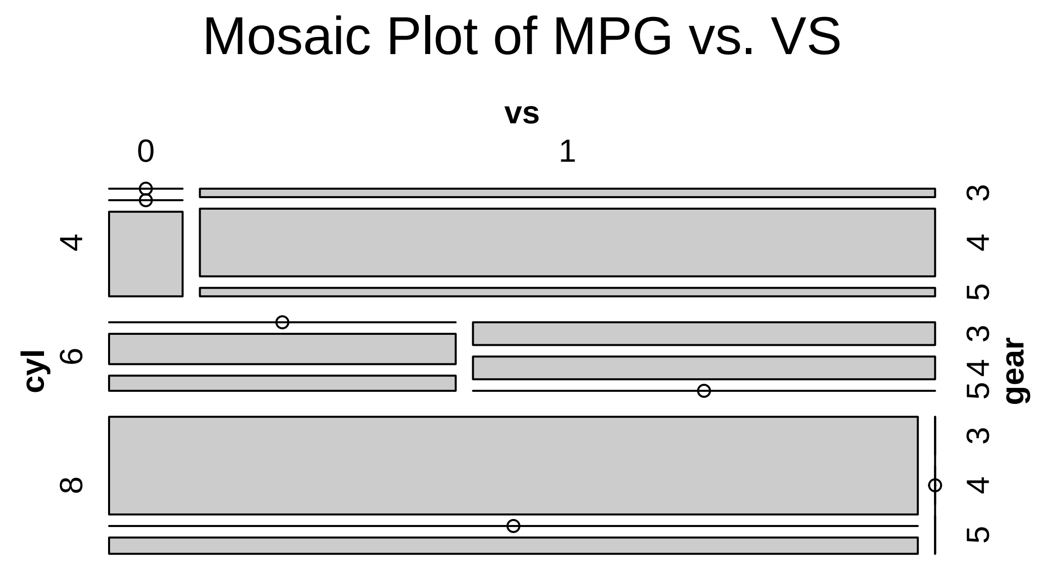
The mtcars dataset, a built-in dataset in R that contains data on 32 cars, is initially loaded by this code. The vcd package, which has utilities for making mosaic plots and other kinds of visualisations, is then installed and loaded by the code.

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).

```r
# 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)
```

```r
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")
```



7.16 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.

8 Data Manipulation using dplyr

dplyr is a package in R that provides a grammar of data manipulation, enabling you to easily manipulate data in a data frame or tibble. Here are some commonly used functions in dplyr:

select(): Selects specific columns from a data frame or tibble.

filter(): Filters rows based on a specified condition.

mutate(): Creates new columns based on calculations or transformations of existing columns.

arrange(): Sorts rows based on one or more columns.

group_by(): Groups the data by one or more columns.

summarize(): Calculates summary statistics for each group.

distinct(): Removes duplicate rows based on a specific column or columns.

rename(): Renames specific columns in a data frame or tibble.

left_join(), right_join(), inner_join(), full_join(): Joins two data frames or tibbles based on a common column or columns.

case_when(): Creates conditional statements to generate new columns.

if_else(): Creates a conditional statement based on a logical expression.

8.1 Illustration on mtcars data

Here's an illustration of how to use dplyr to manipulate the mtcars data:

Load the dplyr package by running library(dplyr).

Create a tibble from the mtcars data frame:

R Copy code # Create a tibble from the mtcars data frame mtcars_tbl <- as_tibble(mtcars) Select specific columns from the mtcars tibble using select() function. For example, select the mpg, cyl, and hp columns: R Copy code # Select specific columns from the mtcars tibble selected_cols <- select(mtcars_tbl, mpg, cyl, hp) Filter rows based on a condition using the filter() function. For example, filter the mtcars tibble to only include cars with a mpg greater than or equal to 20: R Copy code # Filter rows based on a condition filtered_tbl

<- filter(mtcars_tbl, mpg >= 20) Create new columns based on existing columns using the mutate() function. For example, add a new column called kmpl that contains the mpg column converted to kilometers per liter: R Copy code # Create new columns based on existing columns mutated_tbl <- mutate(mtcars_tbl, kmpl = mpg * 0.425144) Sort rows based on one or more columns using the arrange() function. For example, sort the mtcars tibble by descending mpg: R Copy code # Sort rows based on one or more columns arranged_tbl <- arrange(mtcars_tbl, desc(mpg)) Group the data by one or more columns using the group_by() function and calculate summary statistics for each group using the summarise() function. For example, group the mtcars tibble by the cyl column and calculate the mean mpg for each group: R Copy code # Group the data by one or more columns and calculate summary statistics for each group grouped_tbl <- group_by(mtcars_tbl, cyl) %>% summarise(mean_mpg = mean(mpg))

9 Summarizing and Visualizing Continuous Data (Part 1 of 3)

9.1 Univariate Continuous Data

1. Reading Data and Attaching Data to Memory

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

9.2 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.

9.3 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
```

```r
# 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.

9.4 Other functions

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

```
[1] 10.4
```

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

```
[1] 33.9
```

9.5 Summarizing a data column

9.5.1 `summary()`

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

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

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

9.5.2 `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)
```

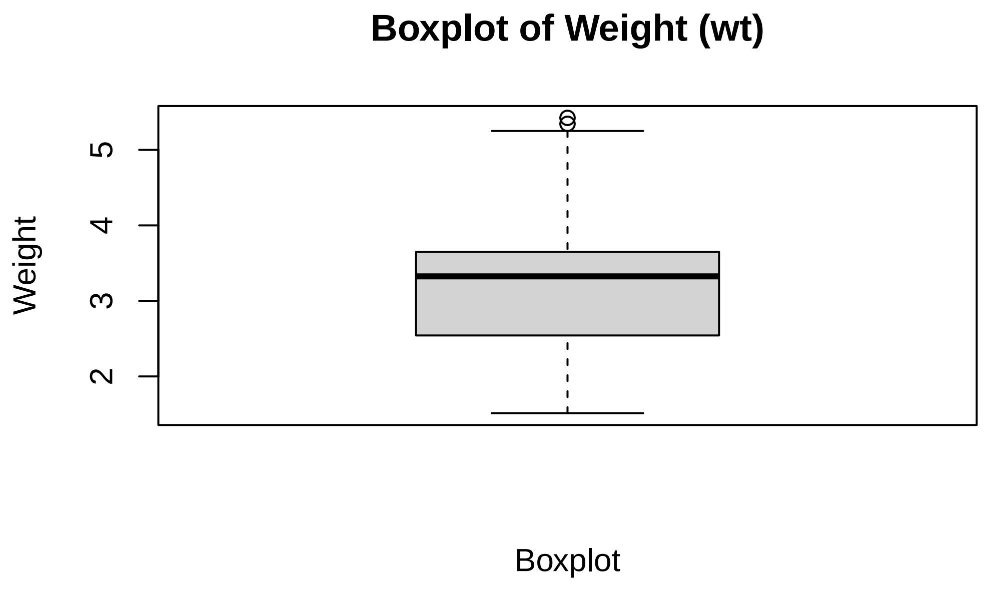
```
   vars  n  mean   sd median trimmed  mad  min  max range skew kurtosis   se
X1    1 32 20.09 6.03   19.2    19.7 5.41 10.4 33.9  23.5 0.61    -0.37 1.07
```

9.6 Visualizing Univariate Continuous Data

9.7 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.

9.8 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.

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

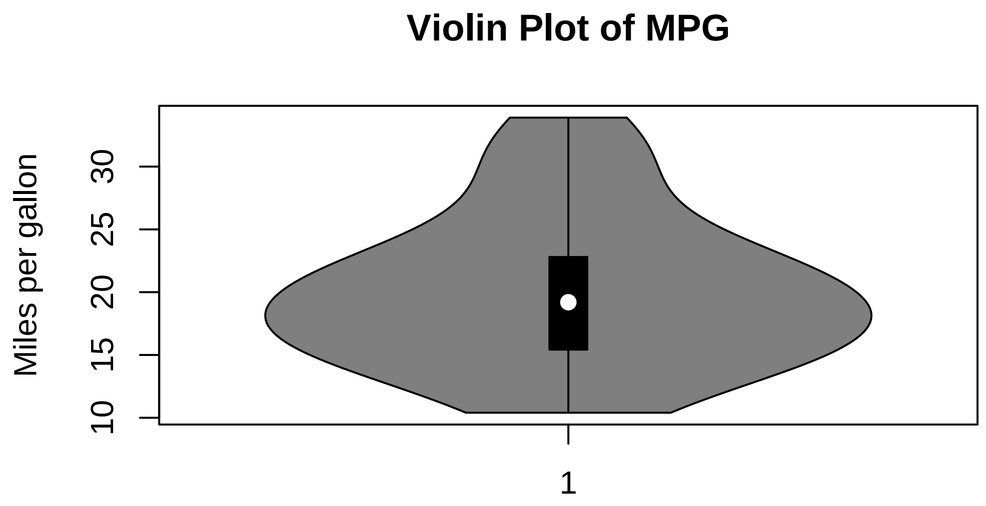
```
Loading required package: sm
```

```
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
```

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



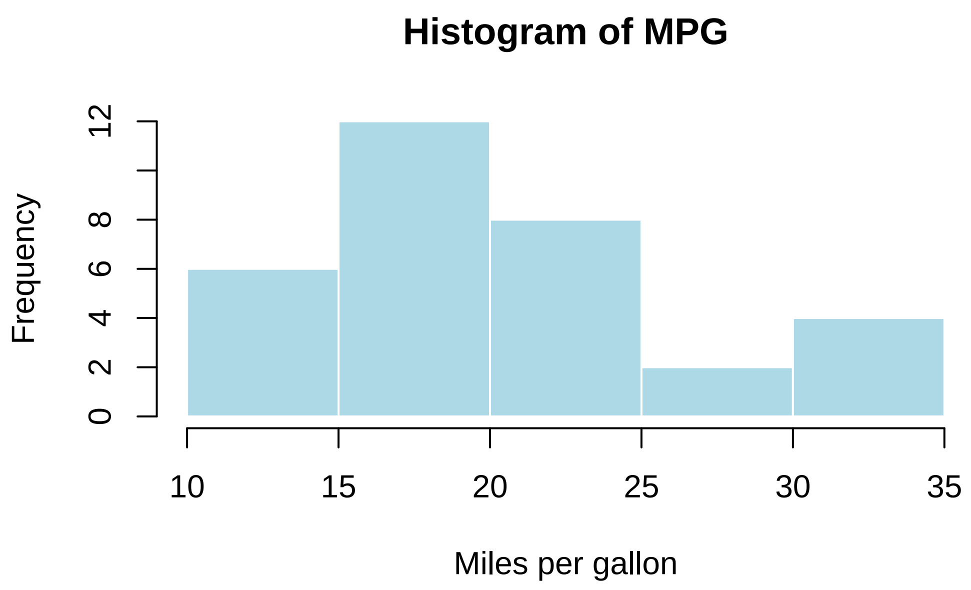
3. In the above code, we create a violin plot of the `mpg` column using the `vioplot()` function. The `main` argument is used to specify the title of the plot, and the `ylab` argument is used to specify the label for the y-axis.
4. The resulting plot will display the full distribution of the `mpg` data 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.

9.9 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.

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

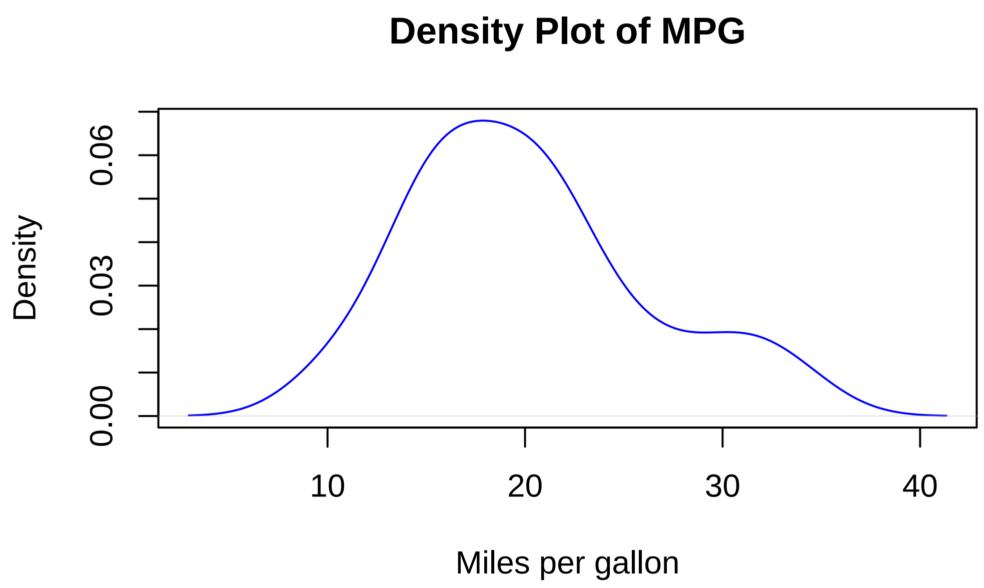


3. We create a histogram of the `mpg` column using the `hist()` function. The main argument is used to specify the title of the plot, and the xlab argument is used to specify the label for the x-axis.
4. The `col` argument is used to set the color of the bars in the histogram, and the border argument 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.

9.10 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")
```

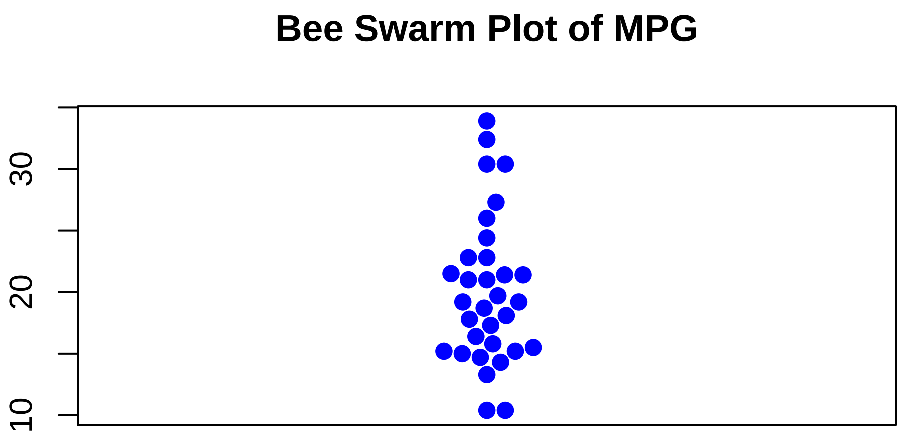


2. In the above code, we create a density plot of the mpg column using the `density()` function.
3. The `plot()` function is used to plot the resulting density object.
4. The `main` argument is used to specify the title of the plot, and the `xlab` argument is used to specify the label for the x-axis.
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.

9.11 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.

```r
# 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.

10 Summarizing and Visualizing Continuous Data (Part 2 of 3)

10.1 Overview of Bivariate Continuous Data

1. Reading Data and Attaching Data to Memory

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

10.2 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)

10.2.1 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
```

10.2.2 tapply()

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

```
       0        1 
3.768895 2.411000 
```

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

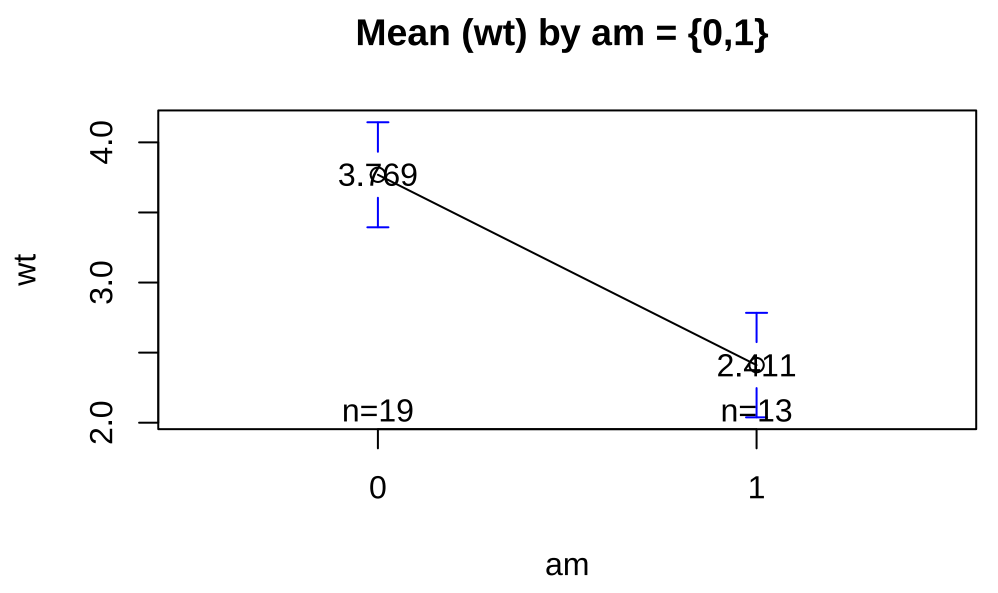
```
        0         1 
0.7774001 0.6169816 
```

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

```
library(gplots)
```

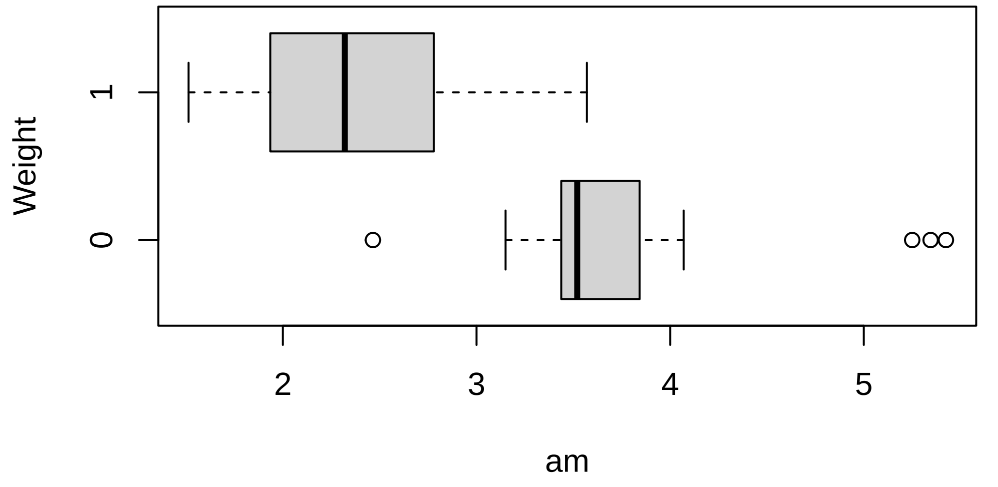
```

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} "
          )
```



10.4 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
        )
```



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

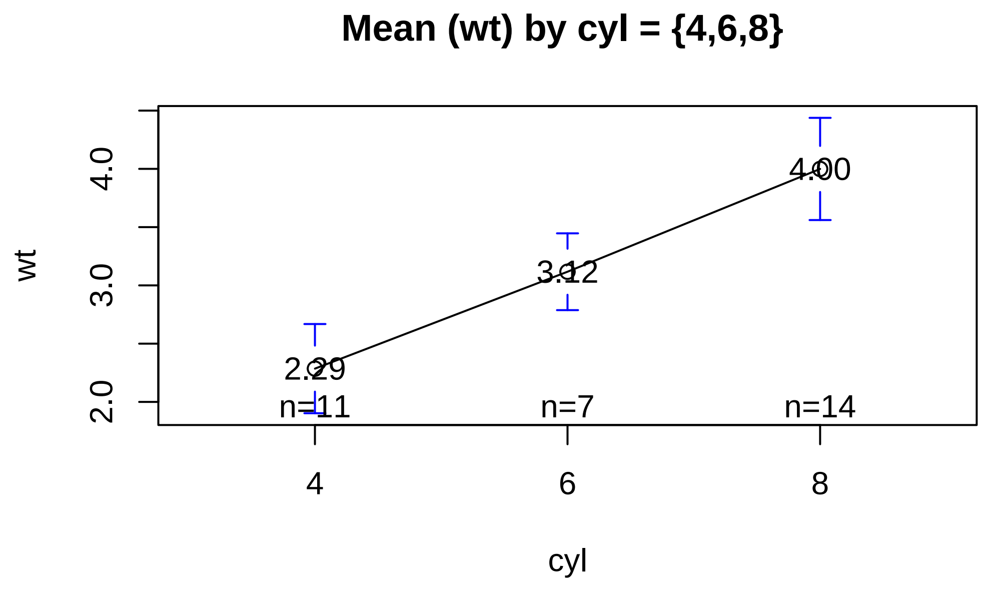
Display a summary table showing the mean weight of the cars broken down by cylinders (cyl=4,6,8)

```r
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
```

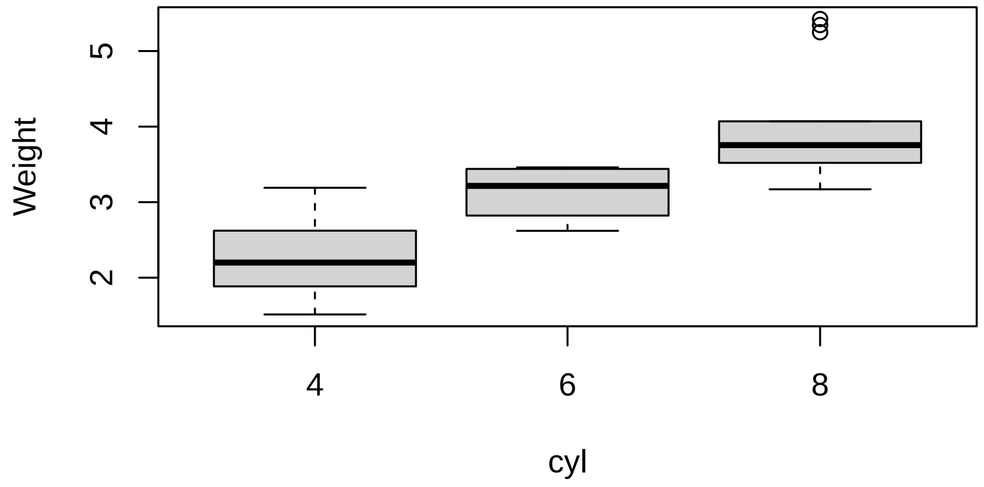
10.6 Show a mean plot showing the mean weight of the cars broken down by cylinders (cyl=4,6,8)

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



10.7 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"
        )
```



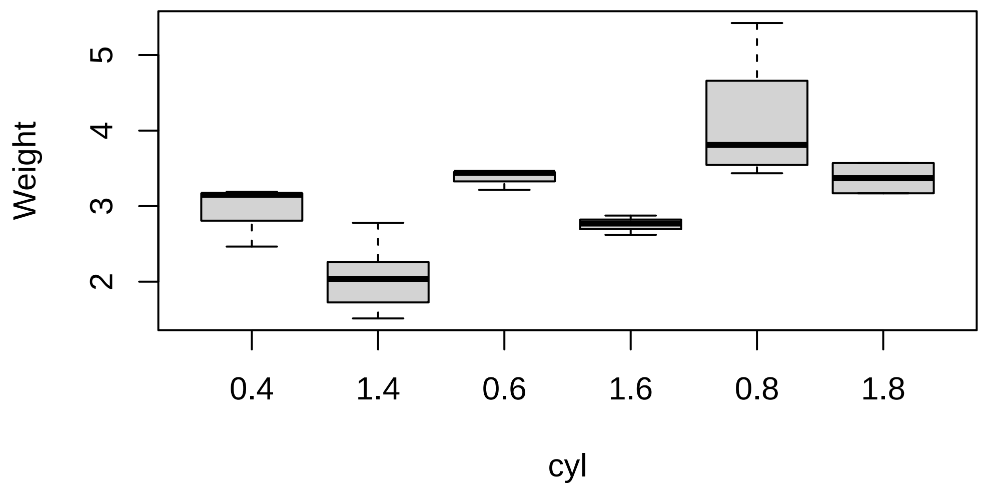
10.8 Distribution of Weight (wt) by Cylinders (cyl = {4,6,8}) and Transmisson Type (am = {0,1})

```
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
```

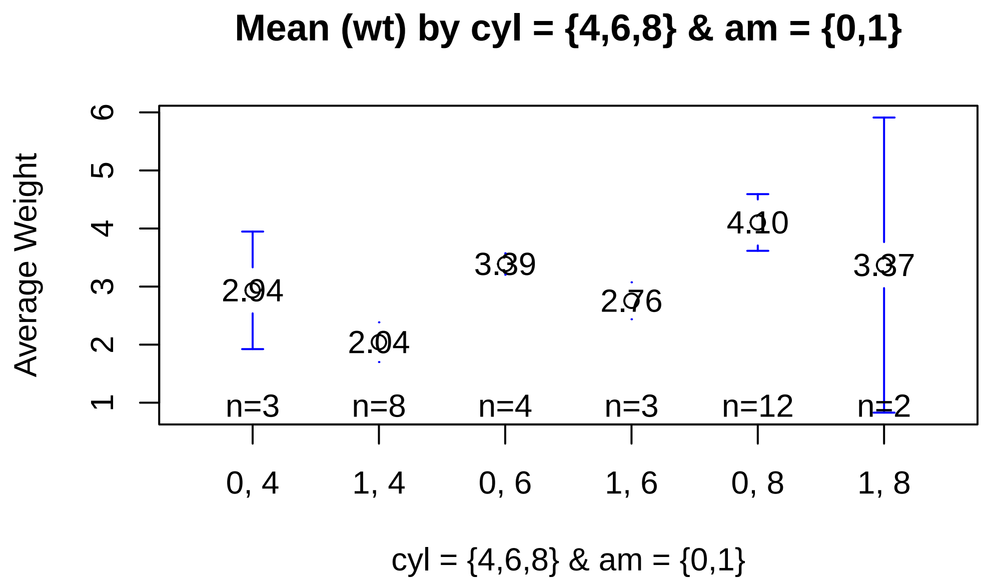
10.9 Visualization - Show a box plot showing the mean weight of the cars broken down by Transmission Type (am=1 & am=0) & cylinders (cyl=4,6,8)

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



10.10 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)

```r
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"
            )
```



11 Summarizing and Visualizing Continuous Data (Part 3 of 3)

11.1 Overview of Bivariate Continuous Data

1. Reading Data and Attaching Data to Memory

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

11.2 Bivariate relationships between Continuous data

11.3 Scatterplot

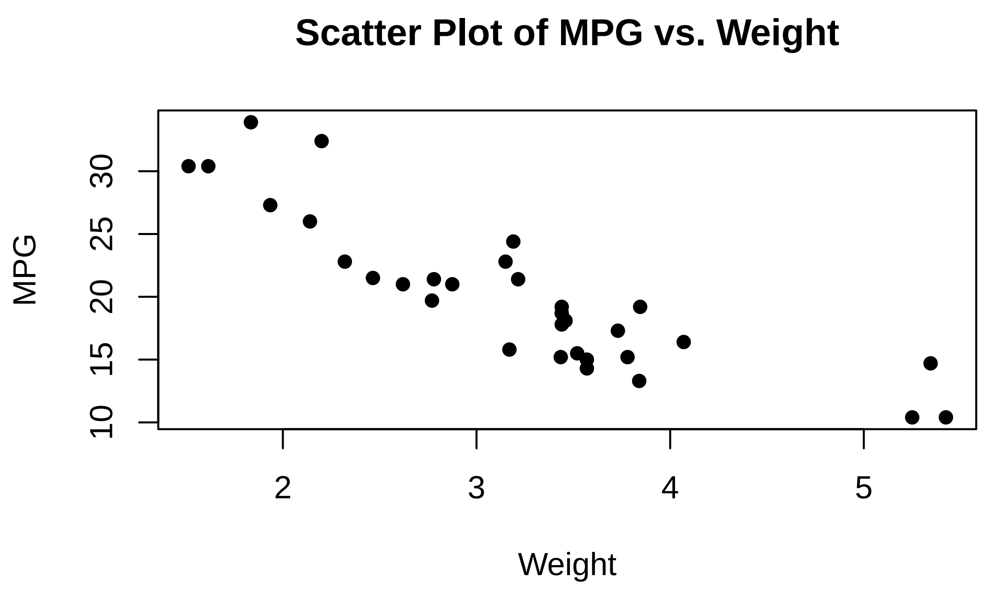
A scatter plot is a type of graph used to display the relationship between two continuous variables. It is a graphical representation of a bivariate distribution, where the values of two variables are plotted as points on a two-dimensional coordinate system.

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:

11.3.1 Scatterplot using plot()

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



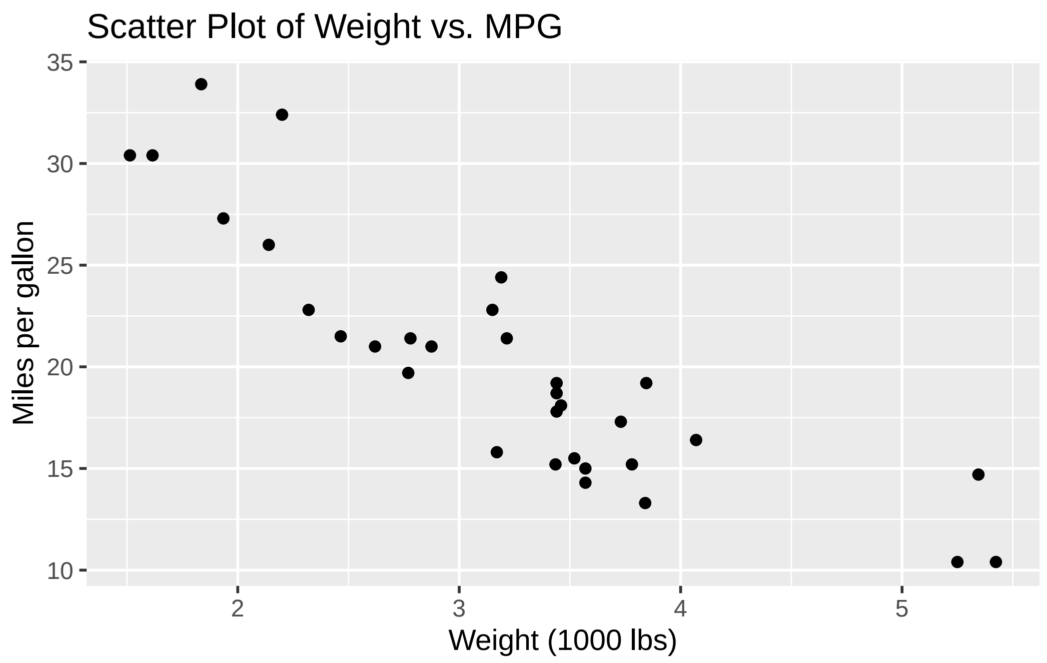
This code will first load the mtcars data set, then create a scatter plot of `mpg` against `wt` using the `plot()` function. The main argument adds a title to the plot, the `xlab` and `ylab` arguments add axis labels, and the pch argument changes the shape of the points to a solid circle. The resulting scatter plot will show the relationship between `mpg` and `wt` in the `mtcars` data set.

11.3.2 Scatterplot using ggplot2

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

```

Attaching package: 'ggplot2'

The following object is masked from 'mtcars':

    mpg
```

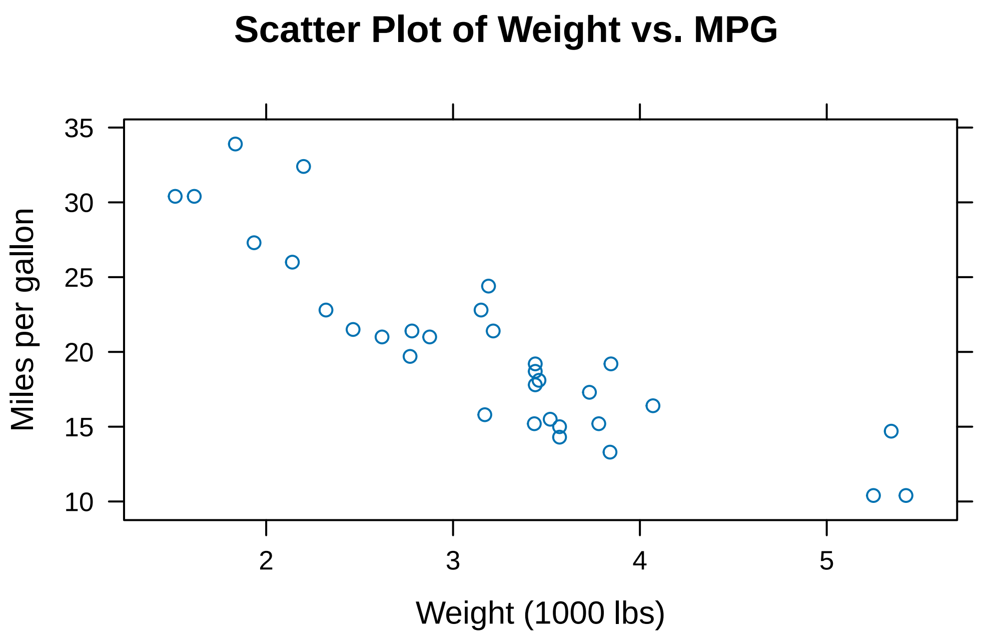
```r
# Create the scatter plot
ggplot(mtcars, aes(x = wt, y = mpg)) +
  geom_point() +
  xlab("Weight (1000 lbs)") +
  ylab("Miles per gallon") +
  ggtitle("Scatter Plot of Weight vs. MPG")
```



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.

11.3.3 Scatterplot using Lattice

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

# Create the scatter plot
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.

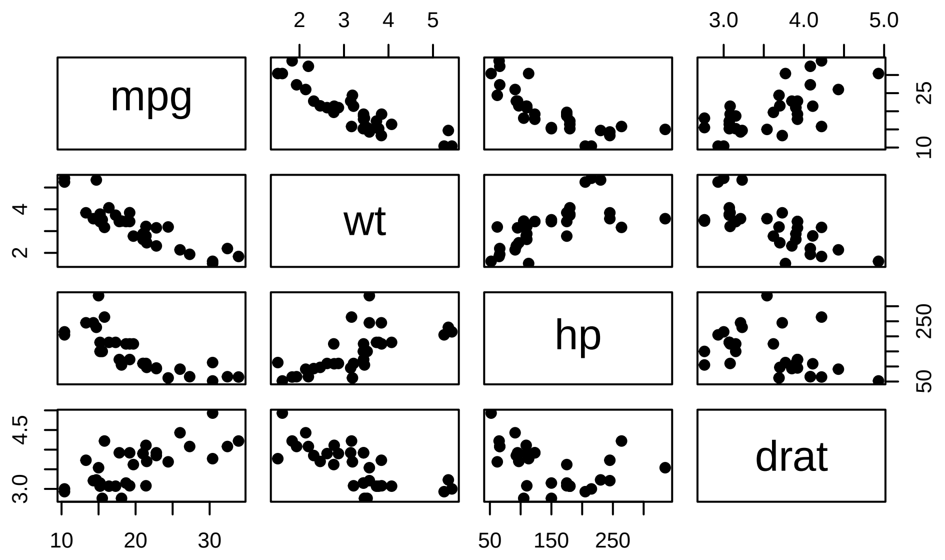
11.4 Scatterplot Matrix

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.

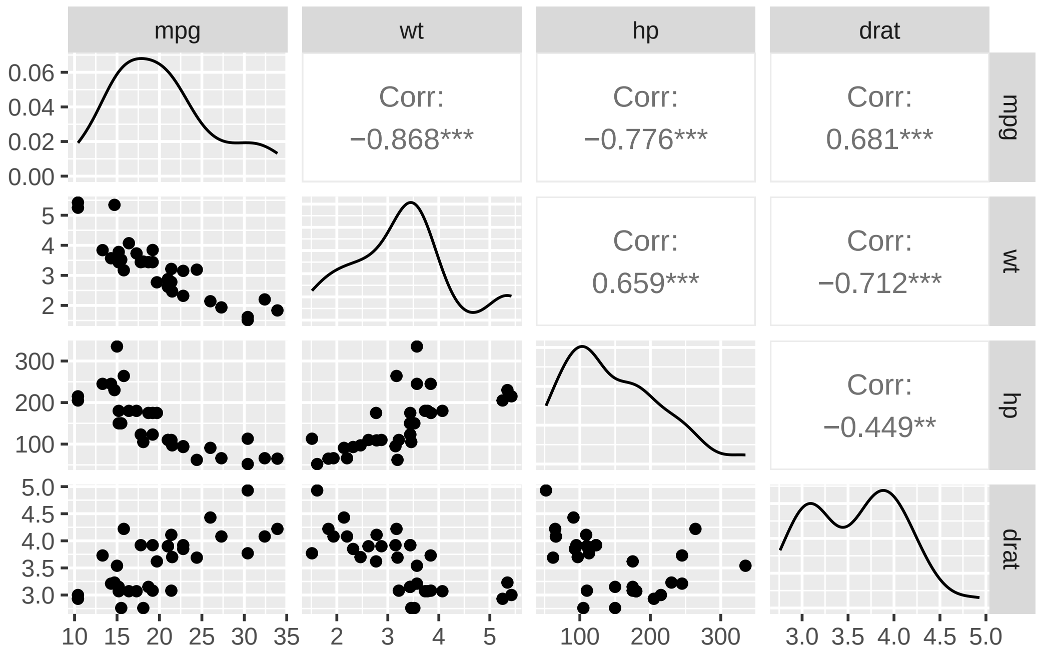
11.4.1 Scatterplot Matrix Using pairs()

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



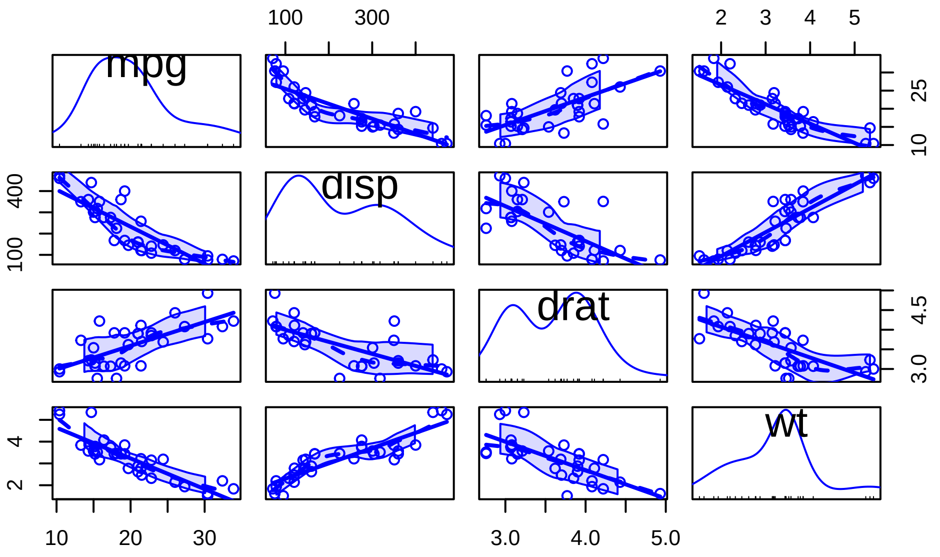
11.4.2 Scatterplot Matrix Using ggpairs()

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

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



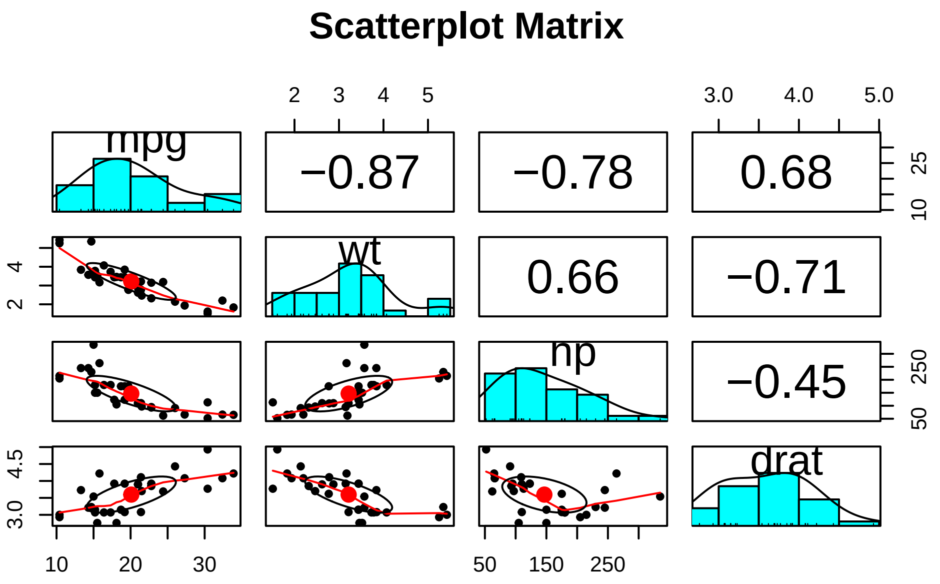
11.4.3 Scatterplot Matrix Using scatterplotMatrix()

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

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



11.4.4 Scatterplot Matrix Using pairs.panels()

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

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

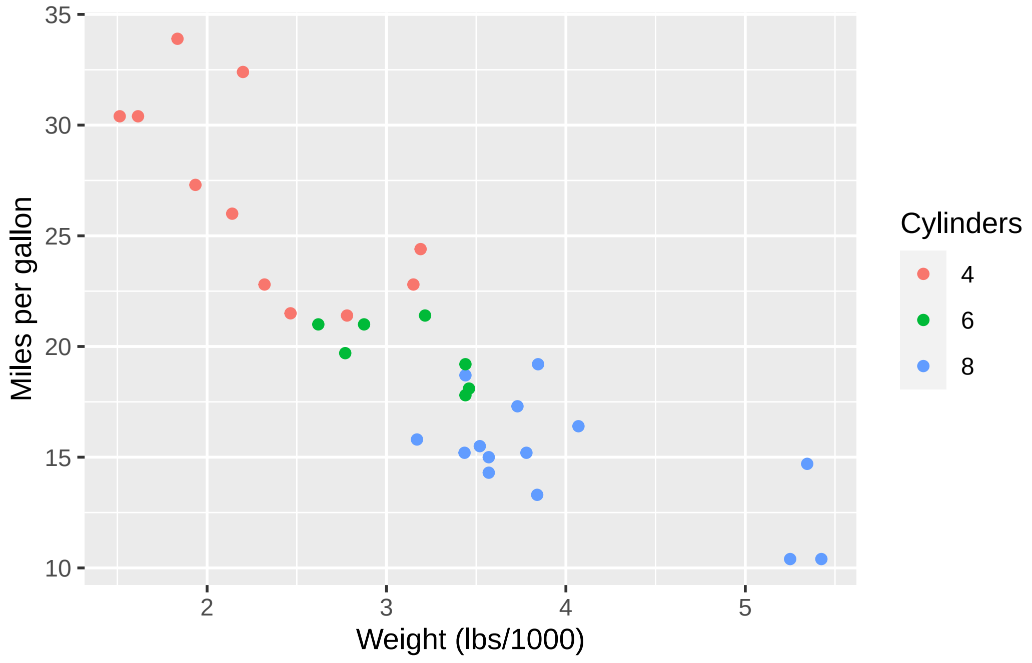


11.5 Scatterplots broken down by Categorical Variables

11.5.1 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).

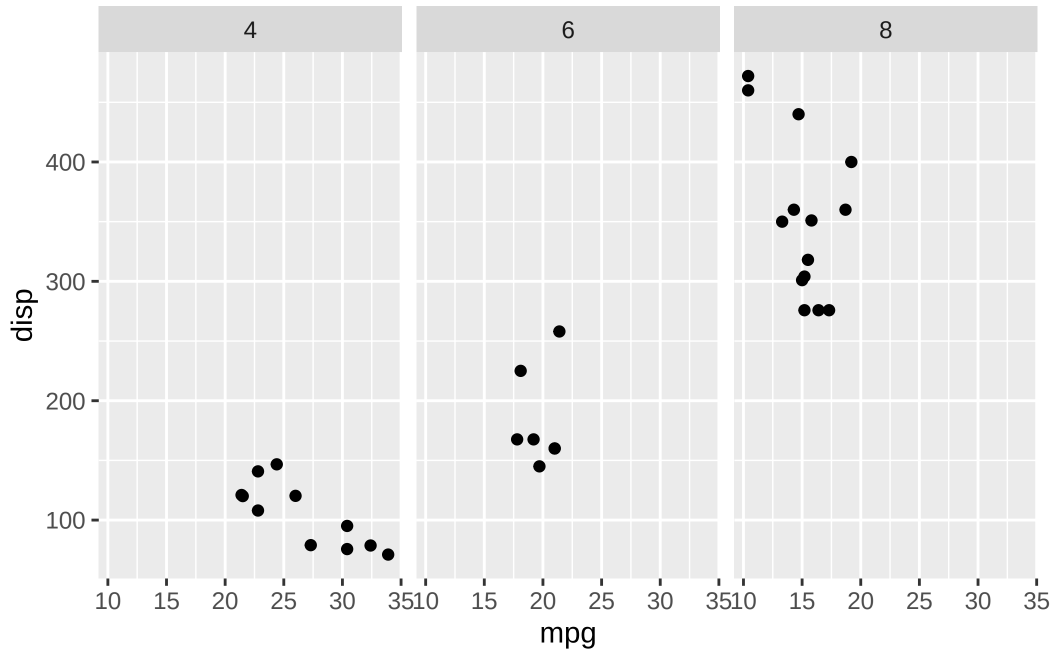
```
# Load the ggplot2 package
library(ggplot2)

# 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")
```



11.5.2 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)
```


disp
mpg