# R Programming Cheat Sheet for Machine Learning 📊

Santiago Jácome

## 1. R Basics and Fundamentals 📝

### Assigning Variables

To store a value in a variable, R provides the assignment operator `<-`, which is the most commonly used and is considered best practice among R users. You can think of `<-` as an arrow pointing to the variable name, helping you remember that the value is being assigned to that variable. Although you can also use `=` for assignment, `<-` is generally preferred in the R community for readability and consistency.

- Syntax: `<-` or `=`
- Example:

```r
x <- 5         # Assigns the value 5 to the variable x
y <- "Hello"   # Assigns the string "Hello" to the variable y
z = 10         # This also assigns the value 10 to z, but using =
```

In this example:
- `x` is assigned the numeric value `5`.
- `y` is assigned a character string `"Hello"`.
- `z` is assigned the numeric value `10` using `=` instead of `<-`.

### Comments

Comments are essential for making your code more understandable. In R, any text following the `#` symbol is considered a comment and is ignored during execution. Comments are helpful for documenting code, explaining complex logic, and making it easier for others (and yourself!) to read and understand the code.

- Syntax: Use `#` to start a comment.
- Example:
```r
# This is a comment explaining the code below
x <- 5        # Assigns 5 to x
y <- "Hello"  # Assigns the string "Hello" to y
```

In this example:
- The line `# This is a comment explaining the code below` is a comment and won’t be executed.
- The comments after `x <- 5` and `y <- "Hello"` provide additional explanations for those assignments.

## Functions

Functions in R are reusable blocks of code designed to perform specific tasks. They can take inputs, called arguments, and return outputs. R has many built-in functions for calculations, data manipulation, and more. Additionally, you can create custom functions for tasks specific to your analysis or project.

You can create your own functions using the `function` keyword. Custom functions allow you to automate repetitive tasks and define operations specific to your needs.

- Syntax for Custom Functions:
```r
function_name <- function(arg1, arg2) {
  # Function body: code that operates on arguments
  result <- arg1 + arg2
  return(result)
}
```

- Example:
```r
# Define a custom function that adds two numbers
add_numbers <- function(a, b) {
  return(a + b)   # Returns the sum of a and b
}
```

```
 # Use the function
 add_numbers(3, 7)   # Outputs 10
```

In this example:
- The function `add_numbers` takes two arguments, `a` and `b`.
- It adds `a` and `b` together and returns the result.
- `add_numbers(3, 7)` will output `10`.

## Summary of Basics and Fundamentals

Assignment: Use <- (preferred) or = to assign values.
Comments: Use # to add comments and explain code.
Functions: R provides built-in functions; you can also create custom functions for repetitive tasks.

# 2. Data Types and Atomic Vectors 🧬

## Data Types in R

R includes several fundamental data types. Knowing how and when to use each is essential for efficient coding and analysis:

- **Numeric**: Represents numbers and includes both integers and doubles (decimals).
  - Example: `5`, `3.14`, `-1.23`
  - When to use: For any quantitative data or values that require mathematical operations.
  - Creating a Numeric Variable:
  ```r
  age <- 30       # An integer
  height <- 5.8   # A decimal (double)
  ```

- **Character**: Used to store text or string data.
  - Example: `"Hello"`, `"Data Science"`
  - When to use: For any textual data, such as names, labels, or categorical variables.
  - Creating a Character Variable:

```r
name <- "Alice"          # Single string
course <- "Data Science"   # Another example of a string
```

- **Logical**: Represents `TRUE` or `FALSE` values, used in comparisons and conditions.
  - Example: `TRUE`, `FALSE`
  - When to use: Useful for conditional checks, filtering data, and boolean logic operations.
  - Creating a Logical Variable:

```r
is_student <- TRUE       # Indicates a true/false condition
has_passed <- FALSE
```

- **Factor**: A categorical data type used to store a fixed set of values (levels), often used for grouping and statistical analysis.
  - Example: `"Male"`, `"Female"`
  - When to use: Best for categorical variables with a limited number of levels.
  - Creating a Factor Variable:

```r
gender <- factor(c("Male", "Female", "Female", "Male"))
```

- **Complex**: Used for complex numbers, which have real and imaginary parts.
  - Example: `4 + 3i`
  - When to use: Rarely used, but can be helpful in advanced mathematical computations.
  - Creating a Complex Variable:

```r
complex_num <- 4 + 3i    # A complex number with real and imaginary parts
```

## Vectors in R

Vectors are the most fundamental data structure in R. A vector is a sequence of data elements of the same type. R uses vectors extensively, and all elements in a vector must be of the same type.

- **Atomic Vectors**: These vectors contain only one data type (numeric, character, logical, or complex).
  - Example: `c(1, 2, 3)` (numeric vector), `c("A", "B", "C")` (character vector)

To create a vector, you use the `c()` function (short for "combine" or "concatenate"). You can combine multiple elements of the same type into a vector.

- Syntax:
```r
my_vector <- c(element1, element2, element3, ...)
```

- Examples:
```r
# Numeric vector
ages <- c(21, 25, 30, 35)      # Creates a numeric vector

# Character vector
names <- c("Alice", "Bob", "Charlie")  # Creates a character vector

# Logical vector
passed <- c(TRUE, FALSE, TRUE)         # Creates a logical vector
```

In this example:
- `ages` is a numeric vector containing age values.
- `names` is a character vector containing names.
- `passed` is a logical vector with boolean values indicating pass/fail status.

Vectors are essential for data operations, statistical calculations, and data analysis in R. Here's how they are commonly used:

1. Mathematical Operations on Vectors:
   - Mathematical operations can be performed element-wise on numeric vectors.
   - Example:
   ```r
   scores <- c(10, 20, 30)
   scores + 5          # Adds 5 to each element, resulting in c(15, 25, 35)
   scores * 2          # Multiplies each element by 2, resulting in c(20, 40, 60)
   ```

2. Logical Operations:
   - You can use logical operations to create logical vectors based on conditions.

   - Example:
   ```r
   ages <- c(18, 22, 25, 17)
   is_adult <- ages >= 18      # Results in c(TRUE, TRUE, TRUE, FALSE)
   ```

3. Subsetting Vectors:
   - Vectors can be subsetted using indices or conditions.
   - Example:
   ```r
   ages <- c(18, 22, 25, 17)
   ages[ages >= 18]            # Returns only values greater than or equal to 18
   ```

4. Named Vectors:
   - You can assign names to vector elements to make data more descriptive.
   - Example:
   ```r
   scores <- c(85, 90, 78)
   names(scores) <- c("Math", "Science", "History")
   scores["Math"]             # Accesses the score for "Math"
   ```

Vectors in R are strictly typed, meaning that all elements must be of the same type. If you attempt to combine different types in a vector, R will coerce them to a common type, following a hierarchy: logical → integer → numeric → character.

- Example of Coercion:
 ```r
 mixed_vector <- c(1, TRUE, "Hello")  # Results in a character vector: c("1", "TRUE", "Hello")
 ```

In this example:
- The logical value `TRUE` and numeric `1` are converted to characters to match the type of `"Hello"`.

## Summary of Vectors and Data Types

- Data Types:

  - Numeric: For calculations.
  - Character: For text data.
  - Logical: For true/false values.
  - Factor: For categorical data.
  - Complex: For complex numbers.

- Creating Vectors:
  - Use `c()` to combine elements into a vector.
  - Use `is.numeric()`, `is.character()`, and `is.logical()` to check types.

## 3. Data Structures in R 📂

Data structures in R allow us to store, organize, and manipulate complex data efficiently. Each structure has its unique advantages, and understanding when and how to use each can significantly enhance your data analysis workflows. Here, we'll explore data frames, tibbles, and lists in R, and provide insights into when to use each.

### Data Frames

Data Frames are one of the most commonly used data structures in R. They allow you to store data in a tabular format, similar to a spreadsheet or SQL table, where each column is a vector and each row represents an observation. Data frames are ideal for storing mixed data types because each column can hold a different type (e.g., numeric, character, or logical).

- Creating a Data Frame:
  - Use the `data.frame()` function to create a data frame.
  - Syntax: `data.frame(column1 = c(...), column2 = c(...))`
  - Example:

```r
# Create a data frame with columns of different data types
df <- data.frame(
  Name = c("Alice", "Bob", "Charlie"),
  Age = c(25, 30, 35),
  Passed = c(TRUE, FALSE, TRUE)
)
```

- Key Characteristics:
  - Each column is a vector of the same type, but different columns can have different types.
  - Rows represent individual observations, and columns represent variables.

- When to Use Data Frames:
  - Ideal for storing datasets with multiple variables.
  - Commonly used for statistical analysis, data manipulation, and machine learning.

## Inspecting Data Frames

R provides several functions to quickly inspect data frames, allowing you to view structure, dimensions, and summaries.

1. `head()` – Displays the first few rows of the data frame (default: 6 rows).
```r
head(df)
```

2. `colnames()` – Returns a character vector of column names.
```r
colnames(df)
```

3. `str()` – Displays the structure of the data frame, showing each column's type and the first few entries.
```r
str(df)
```

4. `summary()` – Provides summary statistics for each column, including mean, median, and range for numeric columns, and counts for categorical columns.
```r
summary(df)
```

5. `View()` – Opens the data frame in a spreadsheet-like view in RStudio, which is particularly helpful for exploring large datasets visually.
```r
View(df)
```

6. `glimpse()` (from `dplyr` package) – Similar to `str()`, but displays a concise view with additional formatting, showing the data types and the first few values of each column. Remember that dplyr can also be loaded through tidyverse.

```r
  library(dplyr)
  glimpse(df)
  ```

## Tibbles

Tibbles are an enhanced version of data frames provided by the `tibble` package (part of `tidyverse`). They offer several advantages over base R data frames, particularly for data manipulation and analysis workflows.

- Creating a Tibble:
  - Use `tibble()` to create a tibble.
  - Syntax: `tibble(column1 = c(...), column2 = c(...))`
  - Example:
    ```r
    library(tibble)

    # Create a tibble
    tbl <- tibble(
      Name = c("Alice", "Bob", "Charlie"),
      Age = c(25, 30, 35),
      Passed = c(TRUE, FALSE, TRUE)
    )
    ```

- Key Characteristics:
  - Tibbles never change the types of inputs, preserving the types as they were entered.
  - They don’t automatically convert character strings to factors (a behavior that data frames sometimes exhibit).
  - Tibbles offer better printing: they only show as many rows and columns as fit in the console, making them more readable.

- Differences Between Tibbles and Data Frames:
  - Type Stability: Tibbles keep the data types stable, while data frames may convert strings to factors by default.
  - Readable Output: Tibbles print in a compact, user-friendly format.
  - Enhanced Functionality: Tibbles work more smoothly with the `tidyverse` functions, particularly when chaining operations with the pipe `%>%`.

- When to Use Tibbles:

  - Use tibbles when working with `tidyverse` packages or when you need consistent data types.
  - They are ideal for data manipulation workflows where readability and data integrity are priorities.

## Inspecting Tibbles

You can inspect tibbles using the same functions as data frames (`head()`, `colnames()`, `str()`, `summary()`, `View()`), with a few notable tibble-specific options:

- `print(tbl, n = 10, width = Inf)` – Prints the first `n` rows and displays all columns in the console.
```r
print(tbl, n = 10, width = Inf)
```

## Lists

Lists in R are flexible containers that can store multiple types of elements, including vectors, data frames, matrices, and even other lists. Unlike data frames and tibbles, lists can hold elements of different lengths and types, making them very powerful for complex data structures.

- Creating a List:
  - Use the `list()` function to create a list.
  - Syntax: `list(name1 = value1, name2 = value2, ...)`
  - Example:
  ```r
  # Create a list with different types of elements
  my_list <- list(
    Name = "Alice",
    Age = 25,
    Scores = c(90, 85, 88),
    Passed = TRUE
  )
  ```

- Accessing Elements in a List:
  - By Name: Use `$` followed by the element’s name.
  ```r
  my_list$Name    # Returns "Alice"
  ```

~~~
  - By Index: Use double square brackets `[[ ]]`.
  ```r
  my_list[[2]]    # Returns 25
  ~~~
  - By Nested Index: For nested lists, use multiple `[[ ]]` calls.
  ```r
  nested_list <- list(list(a = 1, b = 2), list(c = 3, d = 4))
  nested_list[[2]][["c"]]  # Returns 3
  ```

- When to Use Lists:
  - Lists are ideal for storing and manipulating complex data structures, especially when data elements have different lengths or types.
  - Commonly used in functions that return multiple outputs or in situations where you need to store related objects in a single structure.

## Comparison of Data Frames, Tibbles, and Lists

| **Feature** | **Data Frame** | **Tibble** | **List** |
|---|---|---|---|
| **Structure** | Tabular | Tabular | Flexible |
| **Column Types** | Only columns can have different types | Only columns can have different types | Elements can have different types |
| **Coercion** | May convert strings to factors | No automatic coercion | No automatic coercion |
| **Printing** | Displays entire data (large output) | Compact display with limited rows | Prints entire content |
| **Usage** | General-purpose tabular data | Ideal for tidyverse workflows | Storing mixed-type or complex data |

## When to Use Each Data Structure

- Data Frames:
  - Use data frames for general-purpose data storage in R, especially when working outside of the `tidyverse`.

  - Suitable for datasets that may contain different data types across columns and can be used for statistical analysis or modeling.

- Tibbles:
  - Use tibbles when working within the `tidyverse` ecosystem (e.g., `dplyr`, `ggplot2`).
  - They are ideal for data manipulation and visualization workflows where data integrity and readability are essential.

- Lists:
  - Use lists for complex or hierarchical data where elements may differ in length or type.
  - Lists are commonly used to store results from multiple calculations, group related objects, or pass multiple outputs from functions.

# 4. Working with Dates and Times 🕒

## The lubridate Package

The lubridate package, part of the `tidyverse`, makes it easier to work with dates and times in R by simplifying date conversion, manipulation, and formatting.

## Basic Functions

- Installation and Loading:
  ```r
  install.packages("lubridate")
  library(lubridate)
  ```

1. `today()`: Returns the current date.
   - Example:
     ```r
     today_date <- today()
     print(today_date)   # Outputs today's date in YYYY-MM-DD format
     ```

2. `now()`: Returns the current date and time.
   - Example:
     ```r
     current_time <- now()
     print(current_time)   # Outputs the current date and time
     ```

## Converting Strings to Dates

lubridate provides a series of functions for converting strings to dates, making it easy to handle different date formats without specifying a format explicitly.

- Common Conversion Functions:
  - `ymd()`: Converts strings in "Year-Month-Day" format.
  - `mdy()`: Converts strings in "Month-Day-Year" format.
  - `dmy()`: Converts strings in "Day-Month-Year" format.

- Examples:
  ```r
  date1 <- ymd("2024-10-30")     # Converts to "2024-10-30"
  date2 <- mdy("October 30, 2024")  # Converts to "2024-10-30"
  date3 <- dmy("30-10-2024")        # Converts to "2024-10-30"
  ```

## Date Manipulation

Once dates are in a recognized format, lubridate makes it easy to manipulate them.

1. Adding/Subtracting Time Intervals:
   - `days()`, `months()`, `years()` can be added to or subtracted from dates.
   - Example:
   ```r
   date <- ymd("2024-01-01")
   date + days(10)         # Adds 10 days, resulting in "2024-01-11"
   date - months(1)        # Subtracts one month, resulting in "2023-12-01"
   ```

2. Extracting Components:
   - `year()`, `month()`, `day()`: Extract specific parts of a date.
   - Example:
   ```r
   date <- ymd("2024-10-30")
   year(date)    # Returns 2024
   month(date)   # Returns 10
   day(date)     # Returns 30
   ```

## Other Ways of Formatting Dates with `as.Date()` and `as.POSIXct()`

Sometimes, you may encounter dates in non-standard formats that lubridate cannot parse directly. In these cases, you can use `as.Date()` or `as.POSIXct()` with custom format strings to convert strings into dates or timestamps.

- Date Conversion with `as.Date()`:
  - Syntax: `as.Date("date_string", format = "format_string")`
  - Examples:
  ```r
  as.Date("2024-10-30", format = "%Y-%m-%d")     # Year-Month-Day
  as.Date("30-10-2024", format = "%d-%m-%Y")     # Day-Month-Year
  as.Date("10-30-2024", format = "%m-%d-%Y")     # Month-Day-Year
  as.Date("30-Oct-2024", format = "%d-%b-%Y")    # Day-AbbrevMonth-Year
  as.Date("2024/10/30", format = "%Y/%m/%d")     # Year/Month/Day
  as.Date("30/10/2024", format = "%d/%m/%Y")     # Day/Month/Year
  as.Date("10/30/2024", format = "%m/%d/%Y")     # Month/Day/Year
  as.Date("30/Oct/2024", format = "%d/%b/%Y")    # Day/AbbrevMonth/Year
  as.Date("Wednesday, 30 October 2024", format = "%A, %d %B %Y")
  ```

- Timestamp Conversion with `as.POSIXct()`:
  - Use `as.POSIXct()` for date-time formats that include time information.
  - Example:
  ```r
  as.POSIXct("2024-10-30 15:30:00", format = "%Y-%m-%d %H:%M:%S")
  ```

### Date Format Codes

Here’s a quick reference for the most commonly used date and time format codes. These codes are used with `as.Date()` and `as.POSIXct()` functions for custom date and time formats.

- Date Components:
  - `%Y`: Year with 4 digits (e.g., `2024`)
  - `%y`: Year with 2 digits (e.g., `24`)
  - `%m`: Month as a number (01-12)
  - `%b`: Abbreviated month name (e.g., `Oct`)
  - `%B`: Full month name (e.g., `October`)

  - `%d`: Day of the month (01-31)

- Day of the Week:
  - `%a`: Abbreviated weekday name (e.g., `Wed`)
  - `%A`: Full weekday name (e.g., `Wednesday`)

- Time Components:
  - `%H`: Hour in 24-hour format (00-23)
  - `%M`: Minute (00-59)
  - `%S`: Second (00-59)

- Week and Day of the Year:
  - `%U`: Week number of the year (00-53), with weeks starting on Sunday
  - `%W`: Week number of the year (00-53), with weeks starting on Monday
  - `%u`: Day of the week as a number (1-7), where Monday is 1 and Sunday is 7

### Summary of Working with Dates and Times

- `lubridate` simplifies date handling in R with functions like `today()` and `now()`.
- Conversion functions `ymd()`, `mdy()`, and `dmy()` allow easy parsing of standard date formats.
- Date manipulation functions like `days()`, `months()`, and `years()` help perform date arithmetic.
- For custom date formats, use `as.Date()` and `as.POSIXct()` with format strings.

# 5. Data Manipulation with Tidyverse 🧩

The tidyverse is a collection of R packages designed for data science, providing a consistent and user-friendly framework for data manipulation, visualization, and analysis. tidyverse includes powerful tools such as dplyr for data manipulation and ggplot2 for visualization. This section focuses on using dplyr functions and pipes to make data manipulation both intuitive and efficient.

### The tidyverse Package

- What is tidyverse?
  - The tidyverse is a suite of R packages that includes `dplyr`, `ggplot2`, `tidyr`, `readr`, `tibble`, `purrr`, and `stringr`.
  - dplyr is the primary package for data manipulation within the tidyverse, offering a range of functions to filter, sort, mutate, and summarize data.

- Installation and Loading:
```r
 install.packages("tidyverse")
 library(tidyverse)
```

- Why Use tidyverse?
  - tidyverse provides a clean, cohesive syntax for data manipulation, making workflows simpler and more readable.
  - It’s particularly useful for working with large datasets, repetitive data transformations, and creating complex, multi-step data analysis pipelines.

## The Pipe Operator `%>%`

One of the most powerful features of the tidyverse is the pipe operator `%>%`, which comes from the `magrittr` package (included in tidyverse). The pipe operator allows you to chain multiple operations together in a logical sequence, making your code easier to read and understand.

- Why Use Pipes?
  - Readability: Pipes allow you to break down complex tasks into a sequence of smaller steps.
  - Streamlined Code: You can pass the output of one function directly into the next without creating intermediate variables.
  - Intuitive Workflow: Pipes let you write code that reads like a series of instructions, making it easy to follow each step.

- How Pipes Work:
  - The `%>%` operator takes the output of the expression on the left and passes it as the first argument to the function on the right.
  - Example Without Pipes:
```r
   # Without pipes, you may need to create intermediate variables
   filtered <- filter(df, Age > 20)
   sorted <- arrange(filtered, Age)
   selected <- select(sorted, Name, Age)
```
  - Example With Pipes:
```r
   # With pipes, you can do this in one readable sequence
```

```
  df %>%
    filter(Age > 20) %>%
    arrange(Age) %>%
    select(Name, Age)
```

## Common Functions for Data Manipulation

The dplyr package provides a range of functions to manipulate data frames or tibbles. Here are some of the most commonly used functions, each explained with simple examples:

### 1. `filter()`: Select Rows Based on Conditions

- Purpose: `filter()` is used to subset rows in a dataset based on specified conditions.
- Syntax: `filter(data, condition)`
- Example:

```r
 # Filter for rows where Age is greater than 20
 df %>%
   filter(Age > 20)
```

- Usage:
  - You can apply multiple conditions using logical operators like `&` (AND) and `|` (OR).

```r
 df %>%
   filter(Age > 20 & Passed == TRUE)  # Select rows where Age > 20 and Passed is TRUE
```

### 2. `select()`: Choose Specific Columns

- Purpose: `select()` is used to select specific columns from a dataset, useful for narrowing down to relevant variables.
- Syntax: `select(data, column1, column2, ...)`
- Example:

```r
 # Select only Name and Age columns
 df %>%
   select(Name, Age)
```

- Usage:
  - You can also exclude columns by using a minus sign (`-`).

```r
df %>%
  select(-Passed)  # Excludes the Passed column
```

## 3. `mutate()`: Create New Columns

- Purpose: `mutate()` is used to add new columns or transform existing ones.
- Syntax: `mutate(data, new_column = expression)`
- Example:

```r
# Add a new column that is 1.1 times the Score column
df %>%
  mutate(AdjustedScore = Score * 1.1)
```

- Usage:
  - You can use `mutate()` to create multiple new columns in one command.

```r
df %>%
  mutate(AdjustedScore = Score * 1.1, PassFail = ifelse(Score >= 50, "Pass", "Fail"))
```

## 4. `arrange()`: Sort Rows by Column Values

- Purpose: `arrange()` sorts the rows of a data frame based on the values of one or more columns.
- Syntax: `arrange(data, column1, column2, ...)`
- Example:

```r
# Sort by Age in ascending order
df %>%
  arrange(Age)
```

- Usage:
  - To sort in descending order, use `desc()`.
  ```r
  df %>%
    arrange(desc(Age))  # Sort by Age in descending order
  ```

### 5. `summarize()`: Aggregate Data

- Purpose: `summarize()` creates summary statistics for data, such as mean, median, sum, or count.
- Syntax: `summarize(data, summary_stat = function(column))`
- Example:
 ```r
 # Calculate the average Age
 df %>%
   summarize(AverageAge = mean(Age, na.rm = TRUE))
 ```

- Usage:
  - `summarize()` is often combined with `group_by()` to create summaries within groups.

### 6. `group_by()`: Group Data by Categories

- Purpose: `group_by()` is used to split data into groups based on the values in one or more columns.
- Syntax: `group_by(data, column1, column2, ...)`
- Example:
 ```r
 # Group data by Passed column and calculate average Age for each group
 df %>%
   group_by(Passed) %>%
   summarize(AverageAge = mean(Age, na.rm = TRUE))
 ```

## Using dplyr Functions Together with Pipes

A typical data manipulation workflow often involves multiple dplyr functions chained together using pipes. This helps create a readable, step-by-step sequence of transformations.

Example Workflow:
Suppose we have a dataset `df` and want to:
1. Filter rows where `Age` is greater than 20.
2. Arrange the results by `Score` in descending order.
3. Add a new column `ScoreCategory` based on `Score`.
4. Select only relevant columns.

```r
df %>%
  filter(Age > 20) %>%
```

```
 arrange(desc(Score)) %>%
 mutate(ScoreCategory = ifelse(Score >= 85, "High", "Low")) %>%
 select(Name, Age, Score, ScoreCategory)
```

In this example:
- Step 1: Filters rows based on age.
- Step 2: Sorts by `Score` in descending order.
- Step 3: Adds a new column categorizing the `Score` into "High" or "Low."
- Step 4: Selects only the columns we’re interested in.

### Summary of Data Manipulation with tidyverse

- tidyverse: A set of packages, with dplyr being the key package for data manipulation.
- Pipe Operator `%>%`: Makes code more readable by chaining operations.
- Common dplyr Functions:
  - `filter()`: Select rows based on conditions.
  - `select()`: Choose specific columns.
  - `mutate()`: Create or transform columns.
  - `arrange()`: Sort rows by column values.
  - `summarize()` and `group_by()`: Aggregate data, especially within groups.

## 6. Data Visualization with ggplot2 📈

ggplot2 is the primary data visualization package in the tidyverse and is based on the principles of The Grammar of Graphics. It provides a structured way to create complex and visually appealing graphics by layering data and aesthetics. ggplot2 is built on a set of building blocks that you can combine to create nearly any type of plot.

### Building Blocks of ggplot2

To create a plot with ggplot2, you need to understand its core components:

1. Data: The dataset you want to visualize. This is usually a data frame or tibble.
2. Geoms: Short for "geometric objects," geoms represent the type of plot you want, such as points, lines, or bars.
3. Aesthetics (aes): The properties of your data that you want to visualize, such as position, color, or size. Aesthetics map variables from your data to visual properties in the plot.

With these building blocks, you can layer elements to construct a wide variety of plots.

## 3 Steps to Create a Plot with ggplot2

The process of creating a ggplot visualization generally follows three steps:

1. Choose a Dataset: Start with the data you want to visualize.
   - Example Dataset: Let's use the built-in `mtcars` dataset in R, which contains information on various car attributes like miles per gallon, horsepower, and weight.

2. Choose a Geom: Select a geom that best represents the data.
   - Common Geoms:
     - `geom_point()`: Scatter plots
     - `geom_line()`: Line plots
     - `geom_bar()`: Bar charts
     - `geom_histogram()`: Histograms

3. Choose Aesthetics: Map variables to aesthetics to define how data points should look.
   - Aesthetics can include x and y positions, color, size, shape, and more.

## Example: Creating a Scatter Plot

Let's create a scatter plot showing the relationship between miles per gallon (mpg) and horsepower (hp) in the `mtcars` dataset, where we color the points by the number of cylinders (cyl).

```r
library(ggplot2)

# Step 1: Choose a dataset
data <- mtcars

# Step 2: Choose a geom (geom_point for scatter plot)
# Step 3: Choose aesthetics (map mpg to x, hp to y, and cyl to color)
ggplot(data, aes(x = mpg, y = hp, color = factor(cyl))) +
  geom_point() +
  labs(
    title = "Scatter Plot of Horsepower vs. MPG",
    x = "Miles per Gallon (mpg)",
    y = "Horsepower (hp)",
    color = "Cylinders"
  )
```

In this example:
- `data = mtcars`: Specifies the dataset.
- `geom_point()`: Specifies a scatter plot, with points representing data points.
- `aes(x = mpg, y = hp, color = factor(cyl))`: Maps `mpg` to the x-axis, `hp` to the y-axis, and uses `cyl` as a color aesthetic to distinguish by cylinder count.

## Facets: Splitting Data into Subplots

Facets allow you to create multiple panels in a single plot based on the levels of a categorical variable. This is especially useful for visualizing subgroups of your data.

1. `facet_wrap()`: Splits data by one variable and lays out the panels in a grid that wraps around.
   - Example:
   ```r
   ggplot(mtcars, aes(x = mpg, y = hp)) +
     geom_point() +
     facet_wrap(~ cyl)  # Creates a separate panel for each cylinder count
   ```

2. `facet_grid()`: Splits data by two variables, arranging them in a grid structure.
   - Example:
   ```r
   ggplot(mtcars, aes(x = mpg, y = hp)) +
     geom_point() +
     facet_grid(gear ~ cyl)  # Creates a grid with gear as rows and cylinders as columns
   ```

In both cases, facets help you compare data across levels of one or two categorical variables.

## Adding Labels and Annotations

Adding labels and annotations helps to enhance the interpretability of your plots by highlighting specific data points or providing additional context.

1. `labs()`: Adds labels to the entire plot, including the title, subtitle, x- and y-axis labels, and legend titles.
   - Example:
   ```r
   ggplot(mtcars, aes(x = mpg, y = hp)) +
     geom_point() +
     labs(
       title = "Horsepower vs. MPG",
   ```

```r
       x = "Miles per Gallon",
       y = "Horsepower",
       color = "Cylinder Count"
     )
```

2. `annotate()`: Adds specific annotations to highlight key points or areas in the plot.
   - Syntax: `annotate("text", x = x_position, y = y_position, label = "text")`
   - Example:

```r
   ggplot(mtcars, aes(x = mpg, y = hp)) +
     geom_point() +
     annotate("text", x = 30, y = 150, label = "High Efficiency", color = "red")
```

In this example, `annotate()` adds a text label at coordinates (30, 150) in the plot, marking it with the label "High Efficiency."

## Saving Plots with ggsave()

Once you create a plot, you may want to save it as an image or PDF file for reports or presentations. The `ggsave()` function provides a simple way to save plots created with ggplot2.

- Usage of `ggsave()`:
  - By default, `ggsave()` saves the last plot displayed.
  - You can specify the filename, width, height, and format of the saved plot.
  - Syntax: `ggsave("filename.extension", plot = last_plot(), width = ..., height = ...)`

- Examples:

```r
 # Save the last plot as a PNG
 ggsave("scatter_plot.png", width = 6, height = 4, dpi = 300)

 # Save as PDF
 ggsave("scatter_plot.pdf", width = 6, height = 4)

 # Save with specified plot object
 my_plot <- ggplot(mtcars, aes(x = mpg, y = hp)) + geom_point()
 ggsave("my_plot.png", plot = my_plot, width = 5, height = 5)
```

Other ways to save plots include:
- Copying to Clipboard: In RStudio, you can right-click the plot and copy it to the clipboard.
- Export from RStudio: Use the “Export” button in the plot window to save as PNG, PDF, or SVG.

## Summary of ggplot2 Basics

- Building Blocks:
  - Dataset: Provides the data.
  - Geom: Determines the type of plot.
  - Aesthetics (aes): Maps data to visual properties.

- Facets:
  - `facet_wrap()`: Creates multiple panels based on one variable.
  - `facet_grid()`: Creates a grid layout based on two variables.

- Labels and Annotations:
  - `labs()`: Adds titles and axis labels.
  - `annotate()`: Adds text or shapes at specific points in the plot.

- Saving Plots:
  - `ggsave()`: Saves plots to a file in various formats and sizes.

# 7. Machine Learning caret Package 🚀

The caret (short for Classification and Regression Training) package in R is a comprehensive toolkit for building and evaluating machine learning models. It simplifies the process of training, testing, and tuning models, making it especially useful for data scientists and analysts who work with predictive models.

## Why Use the caret Package?

The caret package is highly valuable because it:

1. Streamlines Workflow: caret provides a consistent interface for training a wide variety of machine learning algorithms, reducing the need to learn different syntax for each algorithm.
2. Automates Preprocessing: caret offers functions for data preprocessing, including normalization, handling missing values, and encoding categorical variables.
3. Simplifies Model Tuning: It includes tools for hyperparameter tuning, allowing you to easily optimize model performance.

4. Supports Cross-Validation: caret makes it easy to evaluate model performance using cross-validation and other resampling methods.

## Key Concepts in Machine Learning with caret

1. Splitting Data: Dividing data into training and testing sets. The training set is used to build the model, while the testing set is used to evaluate it.
2. Training Models: caret provides access to numerous algorithms (e.g., linear regression, decision trees, k-nearest neighbors) through a unified `train()` function.
3. Tuning Hyperparameters: caret enables parameter tuning through `trainControl()` to improve model performance.
4. Evaluating Performance: caret provides tools to evaluate models using metrics like accuracy, precision, recall, and others, depending on the model type.

## Key Functions in caret

1. `train()`: The core function for training a model with caret.
2. `trainControl()`: Defines the method for cross-validation or resampling, helping to tune and evaluate the model.
3. `createDataPartition()`: Splits the data into training and testing sets.
4. `confusionMatrix()`: Evaluates the performance of classification models by creating a confusion matrix.

## Step-by-Step Guide to Using caret

### 1. Splitting Data into Training and Testing Sets

One of the first steps in any machine learning workflow is to split your dataset into a training set and a testing set. This allows you to build the model on one part of the data (training) and evaluate it on unseen data (testing).

- Using `createDataPartition()`:
  - This function randomly splits the data based on a specified proportion.
  - Example:

~~~
  ```r
  library(caret)

  # Load dataset
  data <- mtcars

  # Split data into training (80%) and testing (20%) sets
  set.seed(123)
~~~

```r
  trainIndex <- createDataPartition(data$mpg, p = 0.8, list = FALSE)
  trainData <- data[trainIndex, ]
  testData <- data[-trainIndex, ]
```

In this example, `createDataPartition()` splits the `mtcars` dataset into 80% training data and 20% testing data. The `list = FALSE` argument returns a matrix instead of a list, making it easy to subset the dataset.

## 2. Training Models with `train()`

The `train()` function is the core of the caret package. It allows you to build a model using a wide variety of algorithms with a consistent interface. You can specify the algorithm, control parameters, and features to train the model.

- Basic Syntax:

```r
 train(formula, data, method, trControl)
```

  - formula: Specifies the relationship between predictors and the outcome (e.g., `mpg ~ .`).
  - data: The dataset to use for training.
  - method: The machine learning algorithm to use (e.g., `"lm"` for linear regression, `"rf"` for random forest).
  - trControl: Specifies control options like cross-validation (created with `trainControl()`).

- Example:

```r
 # Set up cross-validation
 control <- trainControl(method = "cv", number = 5)  # 5-fold cross-validation

 # Train a linear regression model
 model <- train(mpg ~ ., data = trainData, method = "lm", trControl = control)
 print(model)
```

In this example:
- `method = "lm"` specifies linear regression.
- `trControl` is set to perform 5-fold cross-validation to evaluate model performance during training.

## 3. Tuning Hyperparameters with `trainControl()`

Hyperparameter tuning is essential for optimizing model performance. With `trainControl()`, you can set up different resampling techniques to evaluate the model's performance on different subsets of the data.

- Common Tuning Methods:
  - "cv": Cross-validation
  - "repeatedcv": Repeated cross-validation
  - "LOOCV": Leave-One-Out Cross-Validation

- Example:

```r
# Set up repeated cross-validation for tuning
control <- trainControl(method = "repeatedcv", number = 10, repeats = 3)

# Train a random forest model with tuning
model <- train(mpg ~ ., data = trainData, method = "rf", trControl = control)
print(model)
```

In this example:
- `method = "repeatedcv"` sets up 10-fold cross-validation, repeated 3 times.
- This approach provides a more robust estimate of model performance.

## 4. Evaluating Model Performance

After training a model, it's essential to evaluate its performance on the test set to see how well it generalizes to new data. caret provides a `confusionMatrix()` function for evaluating classification models.

- For Classification Models:

```r
# Example with classification
predictions <- predict(model, testData)
confusionMatrix(predictions, testData$mpg)
```

- For Regression Models:

```r
# Predict on test data for regression
predictions <- predict(model, testData)
```

```
# Calculate RMSE (Root Mean Square Error)
rmse <- sqrt(mean((predictions - testData$mpg)^2))
print(rmse)
```

In classification, `confusionMatrix()` provides a range of metrics, including accuracy, sensitivity, and specificity. For regression, you typically calculate error metrics like RMSE (Root Mean Square Error) or MAE (Mean Absolute Error) manually.

## Supported Algorithms in caret

The `train()` function in caret supports many popular algorithms, allowing you to specify them via the `method` argument. Here are a few commonly used algorithms:

1. Linear Regression: `method = "lm"`
2. Random Forest: `method = "rf"`
3. k-Nearest Neighbors: `method = "knn3"`
4. Support Vector Machines: `method = "svmLinear"`
5. Decision Trees: `method = "rpart"`

Each algorithm has its strengths and weaknesses, so the choice of algorithm depends on the type of problem (classification or regression) and the nature of the data.

- Example: Training a k-Nearest Neighbors (kNN) Classifier:

```r
# Set up cross-validation
control <- trainControl(method = "cv", number = 5)

# Train a kNN model
knn_model <- train(Species ~ ., data = iris, method = "knn", trControl = control)
print(knn_model)
```

In this example, `method = "knn"` specifies a k-Nearest Neighbors model, and `Species ~ .` uses all other columns in the `iris` dataset as predictors.

## Summary of caret Package for Machine Learning

- Data Splitting: `createDataPartition()` splits data into training and testing sets.
- Model Training: `train()` trains models using various algorithms.
- Hyperparameter Tuning: `trainControl()` sets up cross-validation and tuning options.

- Model Evaluation: `confusionMatrix()` for classification, manual metrics for regression.

## 8. Essential Machine Learning Concepts 📊

### Confusion Matrix

```r
library(caret)
predicted <- sample(iris$Species, 150, replace = TRUE)
actual <- iris$Species
confusionMatrix(factor(predicted), factor(actual))
```

Sensitivity: True positive rate.

Specificity: True negative rate.

Prevalence: Proportion of the positive class.

### Regression Models

- Linear Regression

```r
model <- lm(Sepal.Length ~ Sepal.Width, data = iris)
summary(model)
```

- Loess Regression

```r
model_loess <- loess(Sepal.Length ~ Sepal.Width, data = iris)
plot(iris$Sepal.Width, iris$Sepal.Length)
lines(iris$Sepal.Width, predict(model_loess), col = "blue")
```

- k-Nearest Neighbors (kNN) with knn3

```r
model_knn <- knn3(Species ~ ., data = iris, k = 5)
predictions <- predict(model_knn, iris, type = "class")
```

Cross-Validation

```r
control <- trainControl(method = "cv", number = 10)
model <- train(Species ~ ., data = iris, method = "knn", trControl = control)
```