**Data cleaning in R**

**Part 1: Introduction to Data Cleaning & Transformation**

**1.1 Understanding Data Structures in R**

Before diving into cleaning data, it’s crucial to understand the structures you're working with.

1. **Data Frames**:  
   A data frame is a table or two-dimensional array-like structure in R. It is the most common data structure for working with datasets.
   * **Example**:

data <- data.frame(name = c("John", "Alice", "Bob"), age = c(25, 30, 22))

print(data)

1. **Tibbles**:  
   A more modern and user-friendly version of data frames, provided by the tibble package. They show the data in a cleaner format and don’t convert strings to factors by default.
   * **Example**:

library(tibble)

data <- tibble(name = c("John", "Alice", "Bob"), age = c(25, 30, 22))

1. **Vectors**:  
   These are one-dimensional arrays that can hold numeric, character, or logical data.
   * **Example**:

age <- c(25, 30, 22)

1. **Lists**:  
   Lists are more complex data structures that can store multiple types of objects.
   * **Example**:

info <- list(name = "John", age = 25, scores = c(85, 90, 78))

**1.2 Loading and Exploring Data**

To start cleaning data, you first need to load it into R.

1. **Reading Data**:
   * **CSV**:  
     The most common file format. Use read.csv() to load it into R.

data <- read.csv("data.csv")

* + **Excel**:  
    Use readxl to read Excel files.

library(readxl)

data <- read\_excel("data.xlsx")

1. **Exploring Data**: After loading the data, you can quickly explore its structure and summary.
   * **str()**: Displays the structure of an object (e.g., columns and types).

str(bamu)

* + **summary()**: Provides a summary of each column (e.g., min, max, mean).

summary(bamu)

* + **head()**: Displays the first few rows of the data.

head(bamu)

**1.3 Common Data Issues to Address**

Here are the common issues you’ll encounter in real-world datasets:

1. **Missing Data (NA)**:  
   Missing data is a significant problem that can be dealt with in several ways.
   * **Detecting missing values**:  
     Use is.na() to check for missing values.

is.na(data$age)

* + **Removing rows with missing values**:  
    You can remove rows with missing values using na.omit().

clean\_data <- na.omit(data)

* + **Replacing missing values**:  
    You can replace NA with the mean, median, or a specified value.

data$age[is.na(data$age)] <- mean(data$age, na.rm = TRUE)

1. **Duplicates**:  
   Duplicate rows can skew analysis.
   * **Identifying duplicates**:  
     Use duplicated() to find duplicates.

duplicated(data)

* + **Removing duplicates**:  
    You can remove duplicates using !duplicated().

clean\_data <- data[!duplicated(data), ]

**Part 2: Basic Data Cleaning and Transformation**

**2.1 Cleaning Text Data**

Working with text data is a common task in data cleaning. Functions from the stringr package make this easier.

1. **Trimming Whitespace**:
   * **str\_trim()** removes any leading or trailing spaces.

library(stringr)

clean\_name <- str\_trim(data$name)

1. **Changing Case**:
   * **str\_to\_upper()** and **str\_to\_lower()** change the case of characters.

data$name <- str\_to\_upper(data$name)

1. **Replacing Text**:
   * **str\_replace()** replaces a specified pattern with a new string.

data$name <- str\_replace(data$name, "Dr.", "")

1. **Extracting Patterns**:
   * **str\_extract()** extracts the first match of a pattern.

email <- str\_extract(data$email, "[A-Za-z0-9]+@[A-Za-z0-9]+\\.[a-z]+")

1. **Splitting Strings**:
   * **str\_split()** splits a string into components.

split\_names <- str\_split(data$name, "\_")

**2.2 Handling Date-Time Data**

Dates often come in various formats. The lubridate package helps you handle date-time data seamlessly.

1. **Parsing Dates**:
   * **ymd()**, **dmy()**, **mdy()**: Functions to parse dates in various formats.

library(lubridate)

data$date <- ymd(data$date)

1. **Extracting Date Components**:
   * You can extract parts like year, month, or day using year(), month(), and day().

year(data$date)

month(data$date)

1. **Time Manipulation**:
   * **floor\_date()** and **ceiling\_date()**: Round dates to the nearest specified unit.

floor\_date(data$date, "month")

**1. Key Text Cleaning Operations**

Below are common tasks and functions for cleaning text data in R.

**1.1 Removing Unwanted Characters**

* **gsub() (Base R)**: Replaces all matches of a pattern in a string.
* **Example**: Remove special characters like punctuation.

text <- "Hello! How are you?"

clean\_text <- gsub("[[:punct:]]", "", text)

# Output: "Hello How are you"

* **str\_remove\_all() (stringr)**: Remove all occurrences of a pattern.

library(stringr)

clean\_text <- str\_remove\_all(text, "[[:punct:]]")

**1.2 Removing Numbers**

* **gsub() or str\_remove\_all()**:

text <- "Product ID: 12345"

clean\_text <- gsub("[0-9]", "", text)

# Output: "Product ID: "

**1.3 Lowercasing Text**

* Text normalization often requires making all text lowercase.
* **Function**: tolower() (Base R) or str\_to\_lower() (stringr).

text <- "HELLO World"

clean\_text <- tolower(text)

# Output: "hello world"

**1.4 Removing Extra Whitespace**

* **Function**: str\_squish() (stringr).  
  This removes leading, trailing, and extra spaces between words.

text <- " Hello World "

clean\_text <- str\_squish(text)

# Output: "Hello World"

**1.5 Stemming and Lemmatization**

* **Stemming**: Reduces words to their base/root form.
  + **Function**: wordStem() from the SnowballC package.

library(SnowballC)

words <- c("running", "runs", "runner")

stems <- wordStem(words)

# Output: "run", "run", "runner"

* **Lemmatization**: Similar to stemming but ensures grammatically correct base forms.
  + **Tool**: Use external packages like textstem.

library(textstem)

lemmatized\_words <- lemmatize\_words(c("running", "better", "children"))

# Output: "run", "good", "child"

**1.6 Tokenization**

Splitting text into individual components (words, sentences, or phrases).

* **Function**: unnest\_tokens() from tidytext package.

library(tidytext)

text\_data <- data.frame(text = c("Hello World", "R is great"))

tokenized <- unnest\_tokens(text\_data, word, text)

# Output: "hello", "world", "r", "is", "great"

**1.7 Removing Stop Words**

Stop words (e.g., "is", "and", "the") are commonly removed during text analysis.

* **Function**: Use the stop\_words dataset from the tidytext package.

R

Copy code

library(tidytext)

data("stop\_words")

cleaned\_data <- anti\_join(tokenized, stop\_words, by = "word")

**2. Advanced Text Cleaning Operations**

**2.1 Regex for Pattern Matching**

**Regular Expressions (Regex)** allow powerful pattern-based text cleaning.

* **Removing Emails**:

text <- "Contact me at john.doe@example.com"

clean\_text <- gsub("[a-zA-Z0-9.\_%+-]+@[a-zA-Z0-9.-]+\\.[a-zA-Z]{2,}", "", text)

# Output: "Contact me at "

* **Extracting Phone Numbers**:

text <- "Call me at (123) 456-7890"

phone <- str\_extract(text, "\\(\\d{3}\\) \\d{3}-\\d{4}")

# Output: "(123) 456-7890"

* **Replacing Multiple Spaces with a Single Space**:

text <- "This is a test"

clean\_text <- gsub("\\s+", " ", text)

# Output: "This is a test"

**2.2 Splitting and Joining Text**

* **Splitting Strings**: Use str\_split() to divide text based on a delimiter.

text <- "apple,banana,cherry"

split\_text <- str\_split(text, ",")[[1]]

# Output: c("apple", "banana", "cherry")

* **Joining Strings**: Use str\_c() to concatenate strings.

words <- c("Hello", "World")

joined\_text <- str\_c(words, collapse = " ")

# Output: "Hello World"

**2.3 Handling Encoding Issues**

Sometimes, text may have encoding issues (e.g., unusual symbols).

* **Function**: iconv() converts text to a specified encoding.

text <- "Müller"

clean\_text <- iconv(text, from = "UTF-8", to = "ASCII//TRANSLIT")

# Output: "Muller"

**3. Workflow for Cleaning Text Data**

Here’s a step-by-step example:

**Raw Data**

text\_data <- c(

" HELLO, my name is John!!! ",

"You can contact me at john.doe@example.com.",

"123 Main Street"

)

**Cleaning Steps**

library(stringr)

cleaned\_text <- text\_data %>%

str\_to\_lower() %>% # Convert to lowercase

str\_squish() %>% # Remove extra spaces

str\_remove\_all("[[:punct:]]") %>% # Remove punctuation

str\_remove\_all("\\d+") # Remove numbers

# Output: "hello my name is john", "you can contact me at", "main street"

**4. Study Focus**

* **Regex Mastery**: Learn the syntax for pattern matching ([a-z], \\s, +, ^, $).
* **Text Packages**:
  + stringr for flexible string manipulation.
  + tidytext for tokenization and sentiment analysis.
  + tm (Text Mining) for advanced cleaning and preprocessing.

### ****Text Transformation Workflows in R****

Text transformation involves modifying, reshaping, and preparing text data for further analysis, such as topic modeling, sentiment analysis, or machine learning. Here's a comprehensive guide to **text transformation workflows** in R:

### ****1. Workflow Overview****

1. **Import and Preprocess**: Load raw data and clean it.
2. **Tokenization**: Split text into words, sentences, or n-grams.
3. **Normalization**: Apply transformations like stemming, lemmatization, and casing.
4. **Feature Engineering**: Generate additional features such as term frequency.
5. **Output**: Save the transformed data for visualization or modeling.

### ****2. Step-by-Step Workflow****

#### ****2.1 Import and Preprocess Text****

* **Importing Text Data**:  
  Use functions like read.csv() or readLines() for importing text files.

text\_data <- readLines("text\_data.txt")

* **Cleaning Text**:  
  Apply functions for removing unwanted characters, whitespace, or special symbols.

library(stringr)

clean\_text <- text\_data %>%

str\_to\_lower() %>%

str\_squish() %>%

str\_remove\_all("[[:punct:]]")

#### ****2.2 Tokenization****

Tokenization breaks text into smaller units, like words or n-grams (sequences of n words).

* **Single Word Tokenization**: Use unnest\_tokens() from tidytext.

library(tidytext)

text\_df <- tibble(text = clean\_text)

tokenized\_text <- text\_df %>%

unnest\_tokens(word, text)

* **N-gram Tokenization**: Create sequences of n words for context.

tokenized\_ngrams <- text\_df %>%

unnest\_tokens(ngram, text, token = "ngrams", n = 2)

#### ****2.3 Text Normalization****

Normalization ensures text consistency and reduces variations.

* **Stemming**: Reduces words to their root form.

library(SnowballC)

tokenized\_text$stem <- wordStem(tokenized\_text$word)

* **Lemmatization**: Produces grammatically correct base forms of words.

library(textstem)

tokenized\_text$lemma <- lemmatize\_words(tokenized\_text$word)

#### ****2.4 Stop Word Removal****

Remove common stop words (e.g., "is", "and") to focus on meaningful terms.

data("stop\_words")

filtered\_text <- tokenized\_text %>%

anti\_join(stop\_words, by = "word")

#### ****2.5 Text Feature Engineering****

* **Term Frequency (TF)**:  
  Count the occurrences of words.

R

Copy code

word\_counts <- filtered\_text %>%

count(word, sort = TRUE)

* **TF-IDF (Term Frequency-Inverse Document Frequency)**:  
  Highlight important words in a document compared to the corpus.

R

Copy code

tf\_idf <- filtered\_text %>%

count(document, word) %>%

bind\_tf\_idf(word, document, n)

#### ****2.6 Advanced Transformations****

* **Sentiment Analysis**: Assign sentiment scores to words using lexicons.

R

Copy code

sentiment <- filtered\_text %>%

inner\_join(get\_sentiments("bing"), by = "word") %>%

count(sentiment, sort = TRUE)

* **Topic Modeling (LDA)**: Extract topics using Latent Dirichlet Allocation.

R

Copy code

library(topicmodels)

dtm <- DocumentTermMatrix(tokenized\_text)

lda\_model <- LDA(dtm, k = 3, control = list(seed = 123))

topics <- tidy(lda\_model, matrix = "beta")

#### ****2.7 Exporting Transformed Text****

Save the transformed data for external tools or further processing.

R

Copy code

write.csv(word\_counts, "word\_counts.csv", row.names = FALSE)

### ****3. Complete Workflow Example****

R

Copy code

library(tidyverse)

library(tidytext)

library(SnowballC)

library(textstem)

# 1. Load and Preprocess

text\_data <- readLines("text\_data.txt")

clean\_text <- text\_data %>%

str\_to\_lower() %>%

str\_squish() %>%

str\_remove\_all("[[:punct:]]")

# 2. Tokenization

text\_df <- tibble(text = clean\_text)

tokenized\_text <- text\_df %>%

unnest\_tokens(word, text)

# 3. Normalization

tokenized\_text <- tokenized\_text %>%

mutate(stem = wordStem(word),

lemma = lemmatize\_words(word))

# 4. Remove Stop Words

data("stop\_words")

filtered\_text <- tokenized\_text %>%

anti\_join(stop\_words, by = "word")

# 5. Feature Engineering (Word Counts)

word\_counts <- filtered\_text %>%

count(word, sort = TRUE)

# 6. Save Output

write.csv(word\_counts, "cleaned\_text.csv", row.names = FALSE)

### ****4. Study Focus****

* **Lexicons**: Learn about bing, nrc, and afinn sentiment lexicons.
* **Text Mining Packages**: Explore tm, textclean, and topicmodels.
* **Visualization**: Use ggplot2 and wordcloud for insights into text data.

Would you like detailed steps for **visualizing transformed text** or other applications like **clustering or summarization**?

4o

##### You said:

Regex Mastery

teach me regex from beginning to advanced in simpler language and then we'll move further

###### ChatGPT said:

ChatGPT

### ****Regex (Regular Expressions): A Beginner to Advanced Guide****

Regular expressions (regex) are patterns used to match, search, and manipulate text efficiently. Here's a step-by-step, **simple-to-advanced guide** to help you master regex in R.

### ****1. Basics of Regex****

#### ****1.1 Literal Characters****

* **What It Does**: Matches exact text.
* **Example**:  
  Pattern: "apple" matches the word "apple" in "I like apple pie.".

R

Copy code

text <- "I like apple pie."

grepl("apple", text) # TRUE

#### ****1.2 Meta-Characters****

Meta-characters are special symbols that hold specific meanings in regex.

| **Meta-Character** | **Meaning** |
| --- | --- |
| . | Matches any single character. |
| \ | Escapes special characters. |
| ^ | Matches the start of a string. |
| $ | Matches the end of a string. |
| \* | Matches 0 or more repetitions. |
| + | Matches 1 or more repetitions. |
| ? | Matches 0 or 1 repetition (optional). |
| {} | Specifies the exact number of repetitions. |
| ` | ` |

**Example**:  
Pattern: "ap.le" matches "apple", "ap-le", "ap\_ble".

R

Copy code

text <- "apple ap-le ap\_ble"

grepl("ap.le", text) # TRUE for all matches

#### ****1.3 Character Classes****

Character classes help match specific groups of characters.

| **Class** | **Matches** |
| --- | --- |
| [abc] | Any one character: a, b, or c. |
| [^abc] | Any character except a, b, or c. |
| [a-z] | Any lowercase letter. |
| [A-Z] | Any uppercase letter. |
| [0-9] | Any digit. |
| [:alnum:] | Any alphanumeric character. |
| [:digit:] | Any digit (equivalent to [0-9]). |
| [:space:] | Any whitespace character. |

**Example**: Match any digit.

R

Copy code

text <- "Order #12345"

grepl("[0-9]", text) # TRUE

### ****2. Intermediate Regex****

#### ****2.1 Quantifiers****

Quantifiers specify how many times a pattern should occur.

| **Quantifier** | **Meaning** |
| --- | --- |
| \* | 0 or more times. |
| + | 1 or more times. |
| ? | 0 or 1 time. |
| {n} | Exactly n times. |
| {n,} | n or more times. |
| {n,m} | Between n and m times. |

**Example**: Match "ha" repeated 2–4 times.

R

Copy code

text <- "hahaha hahah ha"

grepl("ha{2,4}", text) # TRUE for "hahaha" and "hahah"

#### ****2.2 Anchors****

* ^ matches the **start** of a string.
* $ matches the **end** of a string.

**Example**: Match a string that starts with "Hello".

R

Copy code

text <- "Hello World"

grepl("^Hello", text) # TRUE

**Example**: Match strings ending with "World".

R

Copy code

text <- "Hello World"

grepl("World$", text) # TRUE

#### ****2.3 Groups and Backreferences****

* **Groups**: Parentheses () are used to group patterns.
* **Backreferences**: Use \\1, \\2, etc., to refer to captured groups.

**Example**: Match repeated words.

R

Copy code

text <- "This is is a test."

grepl("(\\b\\w+\\b) \\1", text) # TRUE for "is is"

### ****3. Advanced Regex****

#### ****3.1 Lookaheads and Lookbehinds****

Lookaheads and lookbehinds match patterns based on what precedes or follows them, without including those characters in the match.

| **Type** | **Syntax** | **Description** |
| --- | --- | --- |
| Positive Lookahead | (?=...) | Matches if ... follows. |
| Negative Lookahead | (?!...) | Matches if ... does not follow. |
| Positive Lookbehind | (?<=...) | Matches if ... precedes. |
| Negative Lookbehind | (?<!...) | Matches if ... does not precede. |

**Example**: Match "cat" only if followed by "dog".

R

Copy code

text <- "catdog catfish"

grepl("cat(?=dog)", text, perl = TRUE) # TRUE for "catdog"

**Example**: Match "dog" only if not preceded by "cat".

R

Copy code

text <- "catdog doghouse"

grepl("(?<!cat)dog", text, perl = TRUE) # TRUE for "doghouse"

#### ****3.2 Non-Greedy Matches****

By default, regex matches as much as possible (greedy matching). Use ? after a quantifier to make it non-greedy.

**Example**: Extract the smallest tag from HTML.

R

Copy code

text <- "<b>bold</b><i>italic</i>"

matches <- regmatches(text, gregexpr("<.\*?>", text))

# Output: "<b>", "<i>"

#### ****3.3 Escaping Special Characters****

To match a meta-character like ., \*, or ?, you need to escape it with \\.

**Example**: Match a literal period.

R

Copy code

text <- "This is a sentence."

grepl("\\.", text) # TRUE

### ****4. Practical Applications of Regex in R****

#### ****4.1 Extracting Email Addresses****

R

Copy code

text <- "Contact us at support@example.com or sales@example.org"

emails <- regmatches(text, gregexpr("[a-zA-Z0-9.\_%+-]+@[a-zA-Z0-9.-]+\\.[a-zA-Z]{2,}", text))

# Output: "support@example.com", "sales@example.org"

#### ****4.2 Extracting Phone Numbers****

R

Copy code

text <- "Call me at (123) 456-7890 or 123-456-7890."

phones <- regmatches(text, gregexpr("\\(?\\d{3}\\)?[-.\\s]?\\d{3}[-.\\s]?\\d{4}", text))

# Output: "(123) 456-7890", "123-456-7890"

#### ****4.3 Replacing Patterns****

R

Copy code

text <- "Visit https://example.com for details."

clean\_text <- gsub("https?://[a-zA-Z0-9./]+", "[URL]", text)

# Output: "Visit [URL] for details."

### ****5. Regex Testing Tools****

Use online tools to test regex patterns:

* [Regex101](https://regex101.com)
* [RegExr](https://regexr.com)

R programming offers a rich set of functions and libraries for data analysis and data science. Below are some of the most important functions and their categories, which are widely used in data manipulation, visualization, statistical analysis, and machine learning.

**1. Data Manipulation**

These functions are primarily used to clean, transform, and prepare data.

* **read.csv(), read.table(), read\_excel(), fread() (from data.table):** Import datasets from CSV, text, Excel, and other formats.
* **write.csv(), write.table()**: Export data to files.
* **subset()**: Extract subsets of data.
* **merge()**: Combine two data frames by matching rows.
* **rbind(), cbind():** Combine data frames by rows or columns.
* **apply(), lapply(), sapply(), tapply(), mapply():** Apply functions across elements of data.
* **aggregate():** Summarize data by groups.
* **dplyr functions:**
  + filter(): Filter rows based on conditions.
  + select(): Select specific columns.
  + mutate(): Create or transform columns.
  + group\_by() and summarize(): Group data and compute summaries.
  + arrange(): Order rows by columns.
  + join functions: Merge data frames (e.g., inner\_join(), left\_join()).

**2. Data Visualization**

R offers versatile tools for visualizing data.

* **Base R Graphics:**
  + plot(): General plotting function.
  + hist(): Create histograms.
  + boxplot(): Generate box plots.
  + barplot(): Produce bar charts.
  + pairs(): Pairwise scatter plots.
* **ggplot2 (popular visualization package):**
  + ggplot() + geom\_point(): Scatter plots.
  + ggplot() + geom\_histogram(): Histograms.
  + ggplot() + geom\_boxplot(): Box plots.
  + ggplot() + geom\_line(): Line plots.
  + facet\_wrap() and facet\_grid(): Create subplots.
* **Other Libraries:**
  + qplot() (quick plots in ggplot2).
  + corrplot(): Visualize correlation matrices.
  + heatmap(): Create heatmaps.

**3. Statistical Analysis**

Functions for hypothesis testing, regression, and statistical modeling.

* **Descriptive Statistics:**
  + mean(), median(), sd(), var(): Compute mean, median, standard deviation, and variance.
  + summary(): Get a summary of a dataset.
  + quantile(): Compute quantiles.
* **Hypothesis Testing:**
  + t.test(): Perform t-tests.
  + chisq.test(): Perform chi-squared tests.
  + wilcox.test(): Conduct Wilcoxon rank-sum tests.
* **Regression:**
  + lm(): Linear regression.
  + glm(): Generalized linear models.
* **ANOVA:**
  + aov(): Perform analysis of variance.
* **Time Series:**
  + ts(): Create a time series object.
  + forecast package functions: E.g., auto.arima() for ARIMA models.
  + acf(), pacf(): Autocorrelation functions.

**4. Machine Learning**

R provides robust libraries for machine learning algorithms.

* **caret package (unified interface):**
  + train(): Train machine learning models.
* **Supervised Learning:**
  + randomForest(): Build random forest models.
  + rpart(): Recursive partitioning for decision trees.
  + nnet(): Neural networks.
  + e1071 functions: E.g., svm() for support vector machines.
* **Unsupervised Learning:**
  + kmeans(): K-means clustering.
  + hclust(): Hierarchical clustering.
  + prcomp(): Principal Component Analysis (PCA).
* **Evaluation Metrics:**
  + confusionMatrix() (from caret).
  + roc() (from pROC).

**5. Data Reshaping**

Functions to transform and rearrange data.

* **reshape2 package:**
  + melt(): Convert data into long format.
  + dcast(): Reshape data into wide format.
* **tidyr package:**
  + gather(): Convert columns to rows.
  + spread(): Convert rows to columns.

**6. Text Mining and Natural Language Processing (NLP)**

Key functions for processing and analyzing text data.

* **tm package:**
  + Corpus(): Create text corpus.
  + tm\_map(): Apply transformations to text.
  + DocumentTermMatrix(): Create document-term matrices.
* **text2vec and tidytext packages**: Advanced text analytics.

**7. Miscellaneous Utilities**

* **Debugging and Profiling:**
  + debug(), traceback(): Debugging tools.
  + system.time(): Measure execution time.
* **Utilities:**
  + sample(): Random sampling.
  + table(): Create frequency tables.
  + unique(), duplicated(): Find unique or duplicate elements.
  + sort(), order(): Sort and reorder data.

**Popular R Packages for Data Science**

* **data.table:** High-performance data manipulation.
* **dplyr and tidyr:** Data wrangling.
* **ggplot2:** Data visualization.
* **caret:** Machine learning.
* **forecast:** Time series analysis.
* **shiny:** Interactive web applications.
* **lubridate:** Date and time manipulation.

These functions and libraries make R an indispensable tool for data scientists and analysts. Let me know if you'd like help implementing specific functions or packages!

The merge() function in R is used to combine two data frames by matching rows based on one or more common columns (keys). It performs database-style join operations like inner join, left join, right join, or full outer join.

**Syntax:**

merge(x, y, by, by.x, by.y, all, all.x, all.y, sort)

* **x**: The first data frame.
* **y**: The second data frame.
* **by**: Column(s) common to both data frames used for merging. Defaults to column names common in both.
* **by.x**: Column(s) in x to match with y. Used if the column names differ.
* **by.y**: Column(s) in y to match with x.
* **all**: Logical, if TRUE, performs a full outer join (includes all rows from both).
* **all.x**: Logical, if TRUE, performs a left join (all rows from x).
* **all.y**: Logical, if TRUE, performs a right join (all rows from y).
* **sort**: Logical, if TRUE, sorts the output by the by columns.

**Examples:**

**1. Inner Join (Default)**

Keeps only rows with matching keys in both data frames.

df1 <- data.frame(ID = c(1, 2, 3), Name = c("Alice", "Bob", "Charlie"))

df2 <- data.frame(ID = c(2, 3, 4), Score = c(90, 85, 88))

merge(df1, df2, by = "ID")

**Result:**

ID Name Score

1 2 Bob 90

2 3 Charlie 85

**2. Left Join**

Keeps all rows from df1 and adds matching rows from df2.

merge(df1, df2, by = "ID", all.x = TRUE)

**Result:**

ID Name Score

1 1 Alice NA

2 2 Bob 90

3 3 Charlie 85

**3. Right Join**

Keeps all rows from df2 and adds matching rows from df1.

merge(df1, df2, by = "ID", all.y = TRUE)

**Result:**

ID Name Score

1 2 Bob 90

2 3 Charlie 85

3 4 <NA> 88

**4. Full Outer Join**

Includes all rows from both data frames.

merge(df1, df2, by = "ID", all = TRUE)

**Result:**

ID Name Score

1 1 Alice NA

2 2 Bob 90

3 3 Charlie 85

4 4 <NA> 88

**5. Using Different Key Names (by.x and by.y)**

df3 <- data.frame(UserID = c(2, 3, 4), Score = c(90, 85, 88))

# Merge with different key names

merge(df1, df3, by.x = "ID", by.y = "UserID")

**6. Merging Multiple Columns**

df4 <- data.frame(ID = c(1, 2, 3), Name = c("Alice", "Bob", "Charlie"), Age = c(25, 30, 35))

# Merge by both `ID` and `Name`

merge(df1, df4, by = c("ID", "Name"))

**Notes:**

* **Duplicates in Key Columns:** merge() creates all combinations of matching rows if there are duplicates in the key columns.
* **Sorting:** By default, the result is sorted by the by columns. Use sort = FALSE to disable sorting.
* **NA Handling:** If there are unmatched rows, they will have NA for the missing values in the result.

merge() is a versatile and powerful function for combining datasets in R.

The functions rbind() and cbind() in R are used to combine data structures by rows and columns, respectively.

**rbind()**

* **Purpose:** Combines data structures (e.g., vectors, matrices, data frames) **row-wise**.
* **Syntax:**

R

Copy code

rbind(object1, object2, ...)

* **Key Requirements:**
  + Objects must have the same number of columns (or elements if vectors).
  + Column names (if applicable) should match, or rbind() will try to align columns by name.

**Example 1: Combine Vectors**

R

Copy code

v1 <- c(1, 2, 3)

v2 <- c(4, 5, 6)

# Combine vectors as rows

rbind(v1, v2)

**Result:**

css

Copy code

[,1] [,2] [,3]

v1 1 2 3

v2 4 5 6

**Example 2: Combine Data Frames**

R

Copy code

df1 <- data.frame(ID = 1:3, Name = c("Alice", "Bob", "Charlie"))

df2 <- data.frame(ID = 4:5, Name = c("David", "Eve"))

# Combine data frames row-wise

rbind(df1, df2)

**Result:**

Copy code

ID Name

1 1 Alice

2 2 Bob

3 3 Charlie

4 4 David

5 5 Eve

**Handling Column Names:**

If column names differ, rbind() will align by names and insert NA for missing columns.

R

Copy code

df3 <- data.frame(ID = 6:7, Age = c(25, 30))

rbind(df1, df3)

**Result:**

r

Copy code

ID Name Age

1 1 Alice NA

2 2 Bob NA

3 3 Charlie NA

4 6 <NA> 25

5 7 <NA> 30

**cbind()**

* **Purpose:** Combines data structures (e.g., vectors, matrices, data frames) **column-wise**.
* **Syntax:**

R

Copy code

cbind(object1, object2, ...)

* **Key Requirements:**
  + Objects must have the same number of rows.

**Example 1: Combine Vectors**

R

Copy code

v1 <- c(1, 2, 3)

v2 <- c(4, 5, 6)

# Combine vectors as columns

cbind(v1, v2)

**Result:**

csharp

Copy code

v1 v2

[1,] 1 4

[2,] 2 5

[3,] 3 6

**Example 2: Combine Data Frames**

R

Copy code

df1 <- data.frame(ID = 1:3, Name = c("Alice", "Bob", "Charlie"))

df2 <- data.frame(Age = c(25, 30, 35), Score = c(90, 80, 85))

# Combine data frames column-wise

cbind(df1, df2)

**Result:**

Copy code

ID Name Age Score

1 1 Alice 25 90

2 2 Bob 30 80

3 3 Charlie 35 85

**Different Row Counts:**

If the row counts differ, cbind() will throw an error:

R

Copy code

v3 <- c(7, 8)

cbind(v1, v3) # Error: number of rows must match

**Key Differences:**

| **Function** | **Action** | **Dimension** |
| --- | --- | --- |
| rbind() | Binds objects row-wise | Increases rows |
| cbind() | Binds objects column-wise | Increases columns |

**Notes:**

* Both functions are versatile and work with vectors, matrices, and data frames.
* Use data.table::rbindlist() for efficient row binding of large lists of data frames.

The apply(), lapply(), sapply(), tapply(), and mapply() functions in R are part of the **apply family** of functions. They are used to apply a function across elements of data structures like vectors, lists, matrices, and data frames. These functions are alternatives to loops, often making the code shorter and more readable.

**1. apply()**

Applies a function over the margins (rows or columns) of a matrix or array.

**Syntax:**

apply(X, MARGIN, FUN, ...)

* **X**: A matrix or array.
* **MARGIN**: 1 for rows, 2 for columns.
* **FUN**: The function to apply.
* **...**: Additional arguments passed to FUN.

**Example:**

mat <- matrix(1:9, nrow = 3, byrow = TRUE)

# Apply sum over rows (MARGIN = 1)

apply(mat, 1, sum)

# Apply mean over columns (MARGIN = 2)

apply(mat, 2, mean)

**2. lapply()**

Applies a function over elements of a list or vector and returns a list.

**Syntax:**

lapply(X, FUN, ...)

* **X**: A list or vector.
* **FUN**: The function to apply.
* **...**: Additional arguments passed to FUN.

**Example:**

lst <- list(a = 1:5, b = 6:10)

# Apply sum to each element of the list

lapply(lst, sum)

**3. sapply()**

Applies a function over elements of a list or vector and returns a simplified result (vector or matrix).

**Syntax:**

sapply(X, FUN, ..., simplify = TRUE)

* **simplify = TRUE**: Attempts to simplify the result (default is TRUE).

**Example:**

# Apply sum to each element, return a vector

sapply(lst, sum)

**4. tapply()**

Applies a function to subsets of a vector, split by factors.

**Syntax:**

tapply(X, INDEX, FUN, ...)

* **X**: A vector to split.
* **INDEX**: A factor (or list of factors) defining groups.
* **FUN**: The function to apply.

**Example:**

# Data

ages <- c(25, 30, 35, 40, 45)

genders <- c("M", "F", "M", "F", "M")

# Apply mean to age subsets grouped by gender

tapply(ages, genders, mean)

**5. mapply()**

Applies a function to multiple arguments in parallel.

**Syntax:**

mapply(FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE)

* **FUN**: The function to apply.
* **...**: Arguments to be passed to the function.
* **SIMPLIFY**: Simplifies the result (default is TRUE).

**Example:**

# Parallel application of sum

mapply(sum, 1:5, 6:10)

**Comparison Table:**

| **Function** | **Input** | **Output** | **Use Case** |
| --- | --- | --- | --- |
| apply() | Matrix or array | Vector or array | Apply a function to rows/columns of a matrix. |
| lapply() | List or vector | List | Apply a function to each element of a list. |
| sapply() | List or vector | Simplified result | Similar to lapply(), but simplifies to vector/matrix. |
| tapply() | Vector with factors | Array | Apply a function to groups of a vector split by factors. |
| mapply() | Multiple vectors or lists | Simplified result | Apply a function to multiple inputs in parallel. |

**When to Use:**

* Use **apply()** for matrix-like operations.
* Use **lapply()** for lists and keep the result as a list.
* Use **sapply()** for lists/vectors when you want a simplified result.
* Use **tapply()** for grouped operations.
* Use **mapply()** for parallel operations across multiple inputs.

The aggregate() function in R is used to apply a function to subsets of data, grouped by one or more factors. It is particularly useful for summarizing data (e.g., calculating mean, sum, or other statistics) by groups.

**Syntax:**

aggregate(x, by, FUN, ..., simplify = TRUE)

* **x**: A data frame, matrix, or formula.
* **by**: A list of grouping variables (factors or vectors) that define groups.
* **FUN**: The function to apply (e.g., mean, sum).
* **...**: Additional arguments passed to the function.
* **simplify**: Logical; if TRUE, tries to simplify the result (default).

**Examples:**

**1. Aggregating a Vector by Groups**

# Data

values <- c(10, 20, 30, 40, 50)

groups <- c("A", "A", "B", "B", "B")

# Calculate the mean for each group

aggregate(values, by = list(Group = groups), FUN = mean)

**Result:**

Group x

1 A 15

2 B 40

**2. Aggregating a Data Frame**

# Data frame

df <- data.frame(

ID = 1:6,

Group = c("A", "A", "B", "B", "C", "C"),

Score = c(85, 90, 78, 88, 92, 95)

)

# Calculate the mean Score by Group

aggregate(Score ~ Group, data = df, FUN = mean)

**Result:**

Group Score

1 A 87.5

2 B 83.0

3 C 93.5

**3. Multiple Grouping Variables**

# Data frame with multiple grouping variables

df <- data.frame(

Group = c("A", "A", "B", "B", "C", "C"),

Gender = c("M", "F", "M", "F", "M", "F"),

Score = c(85, 90, 78, 88, 92, 95)

)

# Calculate mean Score by Group and Gender

aggregate(Score ~ Group + Gender, data = df, FUN = mean)

**Result:**

Group Gender Score

1 A F 90

2 A M 85

3 B F 88

4 B M 78

5 C F 95

6 C M 92

**4. Aggregating Multiple Columns**

# Data frame with multiple columns

df <- data.frame(

Group = c("A", "A", "B", "B", "C", "C"),

Score1 = c(85, 90, 78, 88, 92, 95),

Score2 = c(70, 75, 80, 85, 90, 95)

)

# Aggregate multiple columns by Group

aggregate(. ~ Group, data = df, FUN = mean)

**Result:**

Group Score1 Score2

1 A 87.5 72.5

2 B 83.0 82.5

3 C 93.5 92.5

**5. Aggregating with Custom Functions**

You can use custom functions for aggregation.

# Custom aggregation: Calculate range

aggregate(Score ~ Group, data = df, FUN = function(x) range(x))

**Notes:**

1. **Formula Interface:** aggregate(y ~ x, data, FUN) is a common pattern, where y is the variable to summarize and x is the grouping factor.
2. **Grouping by Multiple Columns:** Use + in the formula interface or provide a list of grouping columns in by.
3. **Efficient Aggregation:** For large datasets, consider dplyr::group\_by() and summarize() for better performance and flexibility.

aggregate() is ideal for quick, grouped calculations.

Top of Form

Bottom of Form