Topic 5: Data Cleaning and filtering

2023-08-16

In this topic, you will learn about :

- Introduction to tidy data
- Introduction to tidyr package
- Missing Data

Introduction to tidy data

Introduction to tidy data

Introduction to Tidy Data in R Programming

Tidy data is a structured and standardized format for organizing datasets that makes data manipulation, visualization, and analysis more straightforward and efficient. Tidy data principles, introduced by Hadley Wickham, aim to provide a consistent and intuitive way to handle data in R. In tidy data, each variable forms a column, each observation forms a row, and each type of observational unit forms a table.

Tidy Data Principles:

1. Each variable has its own column: Each variable or attribute of the data is represented by a separate column.
2. Each observation has its own row: Each unique observation or case is represented by a separate row.
3. Each type of observational unit forms a table: Related data should be organized into separate tables.

Benefits of Tidy Data:

1. Simplified data manipulation and analysis with consistent data structures.
2. Easy integration with other R packages and tools.
3. More straightforward data visualization and interpretation.

Example of Tidy Data:

Suppose we have a dataset containing information about the age and height of individuals in different countries. The tidy format of this data would be as follows:

Table 1: Data

Country	Age	Height
USA	25	175
Canada	30	180
Germany	22	168
USA	28	172
Canada	35	185
Germany	27	175

Here, each row represents a unique individual, and each column represents a variable (Country, Age, and Height).

Tidying Data in R:

To convert data into a tidy format, you can use the tidyr package, which provides functions like **gather()** and **spread()** to reshape and tidy data.

Example: Tidying Data using tidyr

```r
# Sample messy data
messy_data <- data.frame(
  Country = c("USA", "Canada", "Germany"),
  Age_2019 = c(25, 30, 22),
  Age_2020 = c(28, 35, 27),
  Height_2019 = c(175, 180, 168),
  Height_2020 = c(172, 185, 175)
)

# Tidying the data using gather() function
tidy_data <- tidyr::gather(messy_data, key = "Year_Height", value = "Value", Age_2019:Height_2020)

print(tidy_data)
```

```
##    Country Year_Height Value
## 1      USA    Age_2019    25
## 2   Canada    Age_2019    30
## 3  Germany    Age_2019    22
## 4      USA    Age_2020    28
## 5   Canada    Age_2020    35
## 6  Germany    Age_2020    27
## 7      USA Height_2019   175
## 8   Canada Height_2019   180
## 9  Germany Height_2019   168
## 10     USA Height_2020   172
## 11  Canada Height_2020   185
## 12 Germany Height_2020   175
```

In this example, the initial dataset messy_data contains age and height information for individuals in different countries for two years (2019 and 2020). We use the **gather()** function from tidyr to tidy the data, combining the "Age" and "Height" variables into a single column, and creating a new column called "Year_Height" to indicate the year of measurement. The resulting tidy_data will follow the principles of tidy data.

By following tidy data principles, you can make your data analysis workflow more organized, efficient, and reproducible in R. Tidy data allows you to take advantage of the full potential of various R packages and tools for data manipulation, visualization, and analysis.

Introduction to tidyr package

Introduction to tidyr package

Introduction to tidyr Package in R Programming

tidyr is a powerful R package designed to help tidy and reshape data, making it easier to work with and analyze. It is part of the tidyverse ecosystem, developed by Hadley Wickham, and complements the dplyr package for data manipulation. tidyr provides functions to convert data between wide and long formats, gather and spread data, and handle missing values efficiently.

Key Functions in tidyr:

1. gather(): This function converts data from wide to long format by gathering columns into key-value pairs. It is particularly useful when you have multiple columns representing different variables, and you want to transform them into rows.

Example of gather():

```
# Sample wide-format data
wide_data <- data.frame(
  ID = c(1, 2, 3),
  Year_2019 = c(100, 120, 80),
  Year_2020 = c(110, 130, 90)
)

# Convert to long format using gather()
tidy_data <- tidyr::gather(wide_data, key = "Year", value = "Value", Year_2019:Year_2020)

print(tidy_data)
```

```
##   ID      Year Value
## 1  1 Year_2019   100
## 2  2 Year_2019   120
## 3  3 Year_2019    80
## 4  1 Year_2020   110
## 5  2 Year_2020   130
## 6  3 Year_2020    90
```

2. spread(): This function converts data from long to wide format by spreading key-value pairs into separate columns. It is useful when you want to reshape your data so that each unique value in a column becomes a new column.

Example of spread():

```
# Sample long-format data
long_data <- data.frame(
  ID = c(1, 1, 2, 2, 3, 3),
  Year = c("Year_2019", "Year_2020", "Year_2019", "Year_2020", "Year_2019", "Year_2020"),
  Value = c(100, 110, 120, 130, 80, 90)
)

# Convert to wide format using spread()
wide_data <- tidyr::spread(long_data, key = "Year", value = "Value")

print(wide_data)
```

```
##   ID Year_2019 Year_2020
## 1  1       100       110
## 2  2       120       130
## 3  3        80        90
```

3. separate() and unite(): These functions are used to split or combine columns containing multiple pieces of information, respectively. **separate()** splits a column into multiple columns based on a separator, while **unite()** combines multiple columns into a single column.

Example of separate() and unite():

```
# Sample data with combined date information
data <- data.frame(
  ID = c(1, 2, 3),
  Date = c("2021-08-01", "2021-08-02", "2021-08-03")
)
```

```r
# Separate the Date column into Year, Month, and Day columns
separated_data <- tidyr::separate(data, col = Date, into = c("Year", "Month", "Day"), sep = "-")

print(separated_data)
```

```
##   ID Year Month Day
## 1  1 2021    08  01
## 2  2 2021    08  02
## 3  3 2021    08  03
```

```r
# Combine Year, Month, and Day columns into a single Date column
united_data <- tidyr::unite(separated_data, col = Date, Year, Month, Day, sep = "-")

print(united_data)
```

```
##   ID       Date
## 1  1 2021-08-01
## 2  2 2021-08-02
## 3  3 2021-08-03
```

These are just a few examples of the functionalities provided by the tidyr package. Tidying and reshaping your data using tidyr can significantly improve data analysis workflows and make your code more expressive and readable. The **tidyr** package, along with other packages in the **tidyverse**, promotes a consistent and efficient approach to data manipulation in R.

Missing Data

Handling Missing Data in R Programming

Missing data refers to the absence of values in a dataset. Dealing with missing data is a crucial step in data analysis, as it can impact the accuracy and validity of your results. In R, there are various approaches to handle missing data effectively.

Common Representations of Missing Data in R:

1. **NA (Not Available):** R uses NA to represent missing values in numeric, character, and logical vectors.
2. **NaN (Not a Number):** Used in specific cases for undefined mathematical operations.
3. **NULL:** Used to represent missing values in lists or objects.

Identifying Missing Data:

1. **is.na():** Checks if elements in a vector or data frame are missing.
2. **complete.cases():** Returns a logical vector indicating complete cases (no missing values) in a data frame.

Example: Identifying Missing Data

```r
# Sample data with missing values
data <- data.frame(ID = 1:5, Value = c(10, NA, 15, NA, 20))

# Detect missing values
missing_values <- is.na(data$Value)
print(missing_values)
```

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

Handling Missing Data:

1. **Omitting Missing Data: na.omit():** Removes rows containing any missing values from a data frame.

Example: Omitting Missing Data

```
# Sample data with missing values
data <- data.frame(ID = 1:5, Value = c(10, NA, 15, NA, 20))
# Remove rows with missing values
clean_data <- na.omit(data)
print(clean_data)
```

```
##   ID Value
## 1  1    10
## 3  3    15
## 5  5    20
```

2. **Replacing Missing Values:**

replace(x, list, values): Replaces values in vector x based on a list of replacement values. **ifelse(test, yes, no)**: Replaces values in a vector based on a logical test.

Example: Replacing Missing Values

```
# Sample data with missing values
data <- data.frame(ID = 1:5, Value = c(10, NA, 15, NA, 20))
# Replace missing values with a specific value
replaced_data <- replace(data$Value, is.na(data$Value), 0)
print(replaced_data)
```

```
## [1] 10  0 15  0 20
```

Example: Replacing Missing Values

```
# Sample data with missing values
data <- data.frame(ID = 1:5, Value = c(10, NA, 15, NA, 20))
# Replace missing values with a specific value
replaced_data <- replace(data$Value, is.na(data$Value), 0)
print(replaced_data)
```

```
## [1] 10  0 15  0 20
```

3. **Imputing Missing Data:**

is.na() combined with subsetting: Replace missing values with a specific value or computed value.

Imputing involves estimating missing values using statistical methods. Popular imputation methods include mean, median, or regression imputation.

- **Example: Imputing Missing Data with mean**

```
# Sample data with missing values
data <- data.frame(ID = 1:5, Value = c(10, NA, 15, NA, 20))
# Impute missing values with mean
mean_value <- mean(data$Value, na.rm = TRUE)
imputed_data <- ifelse(is.na(data$Value), mean_value, data$Value)
print(imputed_data)
```

```
## [1] 10 15 15 15 20
```

- **Using Hmisc Library and imputing with Median value:**

Using the function **impute( )** inside Hmisc library let's impute the column marks2 of data with the median value of this entire column.

Example: Using Hmisc Library and imputing with Median value

```r
# install and load the required packages

install.packages("Hmisc")
library(Hmisc)

# create a adataframe
data <- data.frame(marks1 = c(NA, 22, NA, 49, 75),
                marks2 = c(81, 14, NA, 61, 12),
                marks3 = c(78.5, 19.325, NA, 28,
                           48.002))

# fill missing values of marks2 with median
impute(data$marks2, median)
```

- **Impute with a specific Constant value**

Using the function **impute( )** inside Hmisc library let's impute the column marks2 of data with a constant value.

Example: Impute missing values

```r
# install and load the required packages

install.packages("Hmisc")
library(Hmisc)
# create a adataframe
data <- data.frame(marks1 = c(NA, 22, NA, 49, 75),
                marks2 = c(81, 14, NA, 61, 12),
                marks3 = c(78.5, 19.325, NA, 28,
                           48.002))

# impute with a specific number
# replace NA with 2000
impute(data$marks3, 2000)
```

Handling Missing Data Summary:

1. Missing data is common in datasets and needs to be addressed for accurate analysis.
2. Use **is.na()** and **complete.cases()** to identify missing data.
3. Choose appropriate methods for handling missing data, such as omitting, imputing, or using specialized packages like **imputeTS**.

Remember that the choice of handling missing data depends on the nature of the data and the objectives of the analysis. Always handle missing data thoughtfully and consider the potential impact on the validity of your results.