Exercises on Preprocessing using R

1. **Handles missing and invalid values using the vtreat package.**

```
install.packages("dplyr")
library(dplyr)
# Set seed for reproducibility
set.seed(123)
# Generate the synthetic dataset
data <- data.frame(
+    ID = 1:100,
+    Age = sample(c(18:75, NA), 100, replace = TRUE),
+    Income = sample(c(30000:120000, NA), 100, replace = TRUE),
+    Number_of_Vehicles = sample(c(0:3, NA), 100, replace = TRUE),
+    Gas_Usage = sample(c(50:200, NA), 100, replace = TRUE)
+ )
# View the first few rows of the dataset
head(data, 10)
treat_plan <- design_missingness_treatment(data, varlist = c("Age", "Income",
"Number_of_Vehicles", "Gas_Usage"))
treated_data <- prepare(treat_plan, data)
print(treated_data)
```

2. **Demonstrates data transformation**

```
(mean_age <- mean(treated_data$Age))
[1] 46.77
(sd_age <- sd(treated_data$Age))
[1] 16.00874
print(mean_age + c(-sd_age, sd_age))
[1] 30.76126 62.77874
```

```
treated_data$scaled_age <- (treated_data$Age -
+                         mean_age) / sd_age
treated_data %>%
+   filter(abs(Age - mean_age) < sd_age) %>%
+   select(Age, scaled_age) %>%
+   head()
treated_data %>%
+   filter(abs(Age - mean_age) < sd_age) %>%
+   select(Age, scaled_age) %>%
+   head()

treated_data %>%
+   filter(abs(Age - mean_age) > sd_age) %>%
+   select(Age, scaled_age) %>%
+   head()
```

3. **Demonstrate Sampling**

```
set.seed(25643)
treated_data$gp <- runif(nrow(treated_data))
treated_test <- subset(treated_data, gp <= 0.1)
treated_train <- subset(treated_data, gp > 0.1)
dim(treated_train)
[1] 90  8
dim(treated_test)
[1] 10  8
```

Group Level Sampling

```
# Create the data frame
household_data <- data.frame(
  household_id = c("000000004", "000000023", "000000023", "000000327", "000000327",
                   "000000328", "000000328", "000000404", "000000424", "000000424"),
```

```
 customer_id = c("000000004_01", "000000023_01", "000000023_02", "000000327_01",
             "000000327_02", "000000328_01", "000000328_02", "000000404_01",
             "000000424_01", "000000424_02"),
 age = c(65, 43, 61, 30, 30, 62, 62, 82, 45, 38),
 income = c(940, 29000, 42000, 47000, 37400, 42500, 31800, 28600, 160000, 250000)
)

# View the data frame
print(household_data)
hh <- unique(household_data$household_id)

set.seed(243674)
households <- data.frame(household_id = hh,
+                  gp = runif(length(hh)),
+                  stringsAsFactors=FALSE)

household_data <- dplyr::left_join(household_data,
+                        households,
+                        by = "household_id")
 print(household_data)
```

4. **More Exercises on Data Cleaning:**

4.1 Handling Missing Values:

```
# Load the airquality dataset
data(airquality)
# Check the structure and summary to see where the missing values are
str(airquality)
summary(airquality)

# 1. Identify the missing values in the dataset.
```

```
sum(is.na(airquality))

# 2. Remove rows with any missing values.
cleaned_data <- na.omit(airquality)

# 3. Impute missing values with the mean of the respective columns.
# Apply mean imputation for each column with missing data
imputed_data <- airquality
for (col in names(imputed_data)) {
  if (any(is.na(imputed_data[[col]]))) {
    imputed_data[[col]][is.na(imputed_data[[col]])] <- mean(imputed_data[[col]], na.rm = TRUE)
  }
}

# Check the result to ensure there are no more missing values
sum(is.na(imputed_data))
```

4.2 Removing Duplicates:

```
# Create a sample dataset
data <- data.frame(
  ID = c(1, 2, 3, 4, 5, 5, 6),
  Value = c(10, 20, 30, 40, 50, 50, 60)
)

# 1. Identify duplicate rows.
duplicates <- duplicated(data)

# 2. Remove the duplicate rows.
cleaned_data <- data[!duplicates,]
```

4.3 Correcting Data Types

```
# Create a sample dataset with incorrect data types
data <- data.frame(
  ID = as.character(1:5),
  Value = as.factor(c(10, 20, 30, 40, 50))
)

# 1. Convert the ID column to numeric.
data$ID <- as.numeric(data$ID)

# 2. Convert the Value column to numeric.
data$Value <- as.numeric(as.character(data$Value))
```

4.4 Scaling and Normalization:

```
# Load the mtcars dataset
data <- mtcars

# 1. Scale the variables (mean = 0, variance = 1).
scaled_data <- scale(data)

# 2. Normalize the variables to a 0-1 range.
normalize <- function(x) {
  return ((x - min(x)) / (max(x) - min(x)))
}
normalized_data <- as.data.frame(lapply(data, normalize))
```

4.5 Data Splitting

```
# Load the iris dataset
data <- iris
```

1. Split the dataset into training (80%) and testing (20%) sets.

```
set.seed(123)
trainIndex <- createDataPartition(data$Species, p = 0.8, list = FALSE)
train_data <- data[trainIndex, ]
test_data <- data[-trainIndex, ]
```

Exercises:

1.Handling Missing Values

Categorical Data:

- **a. Load a dataset with categorical variables and introduce some missing values. How would you handle these missing values by:**
  - **Replacing them with the mode of the variable?**
  - **Replacing them with a new category, such as "Unknown"?**
- **b. How would you use the `vtreat` package to create a treatment plan for handling missing values in categorical data?**

Numerical Data:

- **c. Load a dataset with numerical variables and introduce some missing values. How would you handle these missing values by:**
  - **Replacing them with the mean of the variable?**
  - **Replacing them with the median of the variable?**

2. Log Transformation

- **Load a dataset with a skewed numerical variable. How would you apply a log transformation to reduce skewness?**
- **After applying the log transformation, how would you check if the skewness has been reduced?**

3. Sampling

- **Load a dataset and perform simple random sampling to select 70% of the data for training and 30% for testing. How would you do this in R?**
- **How would you perform stratified sampling based on a categorical variable to ensure each category is proportionately represented in the sample?**

4. Normalization

- Load a dataset with numerical variables. How would you normalize these variables to a range of [0, 1]?
- After normalizing the variables, how would you verify that the transformed variables are within the [0, 1] range?

5. Standardization

- Load a dataset with numerical variables. How would you standardize these variables to have a mean of 0 and a standard deviation of 1?
- After standardizing the variables, how would you check if the mean and standard deviation are as expected (mean = 0, sd = 1)?

Spotting problems using graphics and visualization

File Link: m https://github.com/WinVector/PDSwR2/ tree/master/Custdata

1. **Load the data**

```
file_path <- file.choose()

data <- readRDS(file_path)
print(data)
customer_data <- readRDS(file_path)
summary(customer_data)
summary(customer_data$income)
summary(customer_data$age)
```

2. Plotting a histogram

```
library(ggplot2)
ggplot(customer_data, aes(x=gas_usage)) +
+    geom_histogram(binwidth=10, fill="gray")
```

3. Producing a density plot

```
library(scales) ggplot(customer_data, aes(x=income)) +
geom_density() + scale_x_continuous(labels=dollar)
```

4. Creating a log-scaled density plot

```
ggplot(customer_data, aes(x=income)) + geom_density() +
scale_x_log10(breaks = c(10, 100, 1000, 10000, 100000, 1000000),
labels=dollar) + annotation_logticks(sides="bt", color="gray")
```

5. Producing a horizontal bar chart

```
ggplot(customer_data, aes(x=state_of_res)) +
geom_bar(fill="gray") + coord_flip()
```

6. **Producing a dot plot with sorted categories**

```
library(WVPlots)
ClevelandDotPlot(customer_data, "state_of_res", sort = 1,
title="Customers by state")
 coord_flip()
```

7. **Producing a line plot**

```
x <- runif(100) y <- x^2 + 0.2*x ggplot(data.frame(x=x,y=y),
aes(x=x,y=y)) + geom_line()
```

8. **Examining the correlation between age and income**

```
customer_data2 <- subset(customer_data, 0 < age & age < 100 & 0 <
income & income < 200000)
```

9. **Creating a scatterplot of age and income**

```
set.seed(245566) customer_data_samp <-
dplyr::sample_frac(customer_data2, size=0.1, replace=FALSE)
ggplot(customer_data_samp, aes(x=age, y=income)) + geom_point() +
ggtitle("Income as a function of age")

ggplot(customer_data_samp, aes(x=age, y=income)) + geom_point() +
geom_smooth() + ggtitle("Income as a function of age")

BinaryYScatterPlot(customer_data_samp, "age", "health_ins", title
= "Probability of health insurance by age")
```

10. **Producing a hexbin plot**

```
library(WVPlots) HexBinPlot(customer_data2, "age", "income",
"Income as a function of age") + geom_smooth(color="black",
se=FALSE)
```

11. **Specifying different styles of bar chart**

```
ggplot(customer_data, aes(x=marital_status, fill=health_ins)) +
geom_bar() ggplot(customer_data, aes(x=marital_status,
fill=health_ins)) + geom_bar(position = "dodge")
ShadowPlot(customer_data, "marital_status", "health_ins", title =
"Health insurance status by marital status")
ggplot(customer_data, aes(x=marital_status, fill=health_ins)) +
geom_bar(position = "fill")
```

12. **Plotting a bar chart with and without facets**

```
cdata <- subset(customer_data, !is.na(housing_type))
ggplot(cdata, aes(x=housing_type, fill=marital_status)) +
geom_bar(position = "dodge") + scale_fill_brewer(palette =
"Dark2") + coord_flip() ggplot(cdata, aes(x=marital_status)) +
geom_bar(fill="darkgray") + facet_wrap(~housing_type,
scale="free_x") + coord_flip()
```

13. **Comparing population densities across categories**

```
customer_data3 = subset(customer_data2, marital_status %in%
c("Never married", "Widowed")) ggplot(customer_data3, aes(x=age,
color=marital_status, linetype=marital_status)) + geom_density()
+ scale_color_brewer(palette="Dark2")
```

14. **Comparing population densities across categories with ShadowHist()**

```
ShadowHist(customer_data3, "age", "marital_status", "Age
distribution for never married vs. widowed populations",
binwidth=5)
```

Exercises on dplyr:

1. Select() Function:

The `select()` function in `dplyr` is used to choose a subset of columns from a data frame. It allows you to specify which columns to keep, and you can also rename or reorder them. It's very useful for narrowing down a dataset to just the variables you need.

Key Features of `select()`:

1. Select specific columns: You can select one or more columns by name.
2. Drop columns: You can exclude specific columns by using the minus (-) sign.
3. Rename columns: You can rename columns while selecting them.
4. Use helper functions: `select()` supports several helper functions to match column names, like `starts_with()`, `ends_with()`, `contains()`, `matches()`, etc.

Example 1: Select Specific Columns

You can select specific columns from the mtcars dataset.

```
library(dplyr)

# Select specific columns
mtcars %>%
  select(mpg, hp, cyl)

# Output:
#                      mpg  hp cyl
# Mazda RX4           21.0 110   6
# Mazda RX4 Wag       21.0 110   6
# Datsun 710          22.8  93   4
# ...
```

Example 2: Exclude Specific Columns

To exclude columns, you can use the minus (-) operator before the column name.

```
# Exclude the 'hp' and 'cyl' columns
mtcars %>%
  select(-hp, -cyl)

# Output:
#                      mpg  disp  drat    wt  qsec  vs  am  gear  carb
# Mazda RX4           21.0 160.0 3.90 2.620 16.46   0   1     4     4
# Mazda RX4 Wag       21.0 160.0 3.90 2.875 17.02   0   1     4     4
# Datsun 710          22.8 108.0 3.85 2.320 18.61   1   1     4     1
# ...
```

Example 3: Rename Columns While Selecting

You can rename columns directly inside the select() function by specifying the new name followed by =.

```
# Rename columns while selecting
mtcars %>%
  select(mileage = mpg, horsepower = hp, cylinders = cyl)

# Output:
#                      mileage horsepower cylinders
# Mazda RX4               21.0        110         6
```

```
# Mazda RX4 Wag           21.0        110         6
# Datsun 710              22.8         93         4
# ...
```

Example 4: Select Columns with Helper Functions

You can use helper functions to match columns based on patterns.

- `starts_with()`: Select columns that start with a prefix.
- `ends_with()`: Select columns that end with a suffix.
- `contains()`: Select columns that contain a substring.

```
# Select columns that start with 'd'
mtcars %>%
  select(starts_with("d"))

# Output:
#                       disp  drat
# Mazda RX4            160.0 3.90
# Mazda RX4 Wag        160.0 3.90
# Datsun 710           108.0 3.85
# ...

# Select columns that contain the letter 'a'
mtcars %>%
  select(contains("a"))

# Output:
#                       drat  am  gear  carb
# Mazda RX4            3.90    1     4     4
# Mazda RX4 Wag        3.90    1     4     4
# Datsun 710           3.85    1     4     1
# ...
```

Example 5: Reorder Columns

You can also reorder the columns by specifying their names in the desired order.

```
# Reorder columns
mtcars %>%
  select(cyl, mpg, hp)
```

```
# Output:
#                       cyl  mpg  hp
# Mazda RX4               6 21.0 110
# Mazda RX4 Wag           6 21.0 110
# Datsun 710              4 22.8  93
# ...
```

Example 6: Select Columns by Index

You can select columns by their index position (although it's often better to use names for clarity).

```
# Select first 3 columns
mtcars %>%
  select(1:3)

# Output:
#                       mpg cyl disp
# Mazda RX4            21.0   6 160.0
# Mazda RX4 Wag        21.0   6 160.0
# Datsun 710           22.8   4 108.0
# ...
```

Example 7: Combining `select()` with Other `dplyr` Functions

You can combine `select()` with other `dplyr` functions like `mutate()` or `filter()` for more complex operations.

```
# Mutate to create a new column, then select specific columns
mtcars %>%
  mutate(weight_kg = wt * 453.592) %>%
  select(mpg, cyl, weight_kg)

# Output:
#                       mpg cyl weight_kg
# Mazda RX4            21.0   6    1188.60
# Mazda RX4 Wag        21.0   6    1303.91
# Datsun 710           22.8   4    1052.01
# ...
```

2. filter() Function

In the dplyr package, the filter() function is used to subset rows of a dataset based on logical conditions. This function helps you filter data by retaining rows that meet specific criteria, whether numeric, character, or based on other logical expressions.

Key Features of filter():

1. **Filter rows**: You can filter rows based on one or more conditions.
2. **Multiple conditions**: You can combine multiple conditions using & (and), | (or), and ! (not).
3. **Flexible**: It works with numeric, character, and factor variables, as well as complex logical conditions.
4. **NA handling**: By default, filter() removes rows with NA values in the columns you filter on, unless you explicitly handle them.

Example 1: Basic Filtering with One Condition

You can filter rows based on a simple condition. For example, filter cars with miles per gallon (mpg) greater than 20.

```
library(dplyr)

# Filter rows where mpg is greater than 20
mtcars %>%
  filter(mpg > 20)

# Output:
#                      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
# ...
```

Example 2: Filtering with Multiple Conditions

You can filter rows based on multiple conditions using & (and) or | (or). For example, filter cars with mpg > 20 and hp > 100.

```
# Filter cars where mpg is greater than 20 and hp is greater than 100
mtcars %>%
  filter(mpg > 20 & hp > 100)

# Output:
#                       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
```

Example 3: Filtering with or Condition

You can use the | (or) operator to filter rows where either condition is true.

```
# Filter cars where mpg is greater than 30 or hp is greater than 150
mtcars %>%
  filter(mpg > 30 | hp > 150)

# Output:
#                       mpg  cyl  disp  hp  drat    wt  qsec  vs  am  gear
carb
# Hornet Sportabout     18.7    8 360.0 175  3.15 3.440 17.02   0   0     3
2
# Valiant               18.1    6 225.0 105  2.76 3.460 20.22   1   0     3
1
# Maserati Bora         15.0    8 301.0 335  3.54 3.570 14.60   0   1     5
8
# Toyota Corolla        33.9    4  71.1  65  4.22 1.835 19.90   1   1     4
1
```

Example 4: Filtering Based on Categorical Values

You can filter rows based on categorical (or factor/character) values. For example, filter cars with 6 cylinders.

```
# Filter cars where the number of cylinders is 6
mtcars %>%
  filter(cyl == 6)
```

```
# Output:
#                      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
# ...
```

Example 5: Filtering with != for Exclusion

You can use != to exclude rows based on a condition. For example, filter out all cars with 4 cylinders.

```
# Exclude cars with 4 cylinders
mtcars %>%
  filter(cyl != 4)

# Output:
#                      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
# ...
```

Example 6: Filtering with %in% for Multiple Values

To filter rows based on multiple possible values, you can use %in%. For example, filter cars with 4 or 6 cylinders.

```
# Filter cars where the number of cylinders is either 4 or 6
mtcars %>%
  filter(cyl %in% c(4, 6))

# Output:
#                      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
# ...
```

Example 7: Filtering Rows with Missing Values

By default, `filter()` removes NA values when filtering. To include rows with NA values, you need to handle them explicitly using `is.na()`.

```
# Create a small dataset with NA values
data <- data.frame(a = c(1, 2, NA, 4), b = c(NA, 5, 6, 7))

# Filter rows where column 'a' is not missing (NA)
data %>%
  filter(!is.na(a))

# Output:
#   a b
# 1 1 NA
# 2 2  5
# 3 4  7
```

Example 8: Filtering Using Logical Expressions

You can use logical expressions for more complex filtering conditions. For example, find cars with `mpg` greater than the average `mpg` in the dataset.

```
# Filter cars with mpg greater than the average mpg
mtcars %>%
  filter(mpg > mean(mpg))

# Output:
#                      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
# Datsun 710          22.8    4 108.0  93  3.85 2.320 18.61   1   1     4
1
# ...
```

3. mutate() function

mutate(): Adding or Modifying Columns

The **mutate()** function in **dplyr** is used to create new columns or modify existing ones. The result will include all the original columns and the new or modified columns specified.

Example 1: Adding a New Column

Let's add a new column to the **mtcars** dataset that calculates the weight in kilograms (**wt_kg**) by converting the existing **wt** column (which is in 1000 lbs).

```
library(dplyr)

# Add a new column 'wt_kg' (weight in kilograms)
mtcars %>%
  mutate(wt_kg = wt * 453.592)

# Output:
#                        mpg  cyl  disp  hp  drat    wt  qsec  vs  am  gear
carb   wt_kg
# Mazda RX4            21.0    6 160.0 110  3.90  2.620 16.46   0   1     4
4 1189.43
# Mazda RX4 Wag        21.0    6 160.0 110  3.90  2.875 17.02   0   1     4
4 1304.10
# ...
```

Example 2: Modifying an Existing Column

You can also use **mutate()** to modify an existing column. For instance, converting the **mpg** column to kilometers per liter.

```
# Modify 'mpg' column to represent km per liter (conversion factor: 1 mpg =
0.425144 km/l)
mtcars %>%
  mutate(mpg = mpg * 0.425144)

# Output:
#                       mpg  cyl  disp  hp  drat    wt  qsec  vs  am  gear
carb
# Mazda RX4           8.928    6 160.0 110  3.90  2.620 16.46   0   1     4
4
```

```
# Mazda RX4 Wag       8.928   6 160.0 110  3.90  2.875 17.02   0   1     4
4
# ...
```

Example 3: Creating Multiple Columns

You can create more than one column at the same time.

```
# Create two new columns: 'wt_kg' and 'hp_per_kg'
mtcars %>%
  mutate(wt_kg = wt * 453.592, hp_per_kg = hp / wt_kg)
```

```
# Output:
#                       mpg  cyl  disp  hp  drat     wt  qsec  vs  am  gear
carb   wt_kg  hp_per_kg
# Mazda RX4           21.0    6 160.0 110  3.90  2.620 16.46   0   1     4
4 1189.43   0.092477
# Mazda RX4 Wag       21.0    6 160.0 110  3.90  2.875 17.02   0   1     4
4 1304.10   0.084342
# ...
```

---

4. arrange() Function:

arrange(): Sorting Rows

The arrange() function is used to reorder rows in a data frame based on the values of one or more columns. It works similarly to the ORDER BY clause in SQL.

Example 1: Sorting by One Column (Ascending)

To sort the mtcars dataset by miles per gallon (mpg) in ascending order:

```
# Sort rows by mpg (ascending)
mtcars %>%
  arrange(mpg)
```

```
# Output:
#                       mpg  cyl  disp  hp  drat     wt  qsec  vs  am  gear
carb
# Maserati Bora        15.0    8 301.0 335  3.54  3.570 14.60   0   1     5
8
```

```
# Ford Pantera L       15.8    8 351.0 264  4.22  3.170 14.50   0   1     5
4
# ...
```

Example 2: Sorting by One Column (Descending)

To sort by a column in descending order, you use the **desc()** function.

```
# Sort rows by mpg (descending)
mtcars %>%
  arrange(desc(mpg))

# Output:
#                      mpg  cyl  disp  hp  drat    wt  qsec  vs  am  gear
carb
# Toyota Corolla       33.9    4  71.1  65  4.22  1.835 19.90   1   1     4
1
# Fiat 128             32.4    4  78.7  66  4.08  2.200 19.47   1   1     4
1
# ...
```

Example 3: Sorting by Multiple Columns

You can sort by multiple columns. For example, sort first by number of cylinders (**cyl**) and then by horsepower (**hp**) within each group of cylinders.

```
# Sort rows by 'cyl' (ascending) and 'hp' (descending)
mtcars %>%
  arrange(cyl, desc(hp))

# Output:
#                      mpg  cyl  disp  hp  drat    wt  qsec  vs  am  gear
carb
# Ferrari Dino         19.7    6 145.0 175  3.62  2.770 15.50   0   1     5
6
# Mazda RX4            21.0    6 160.0 110  3.90  2.620 16.46   0   1     4
4
# ...
```

5. summarize() Function

In dplyr, the summarize() function is used to compute summary statistics (like mean, median, sum, etc.) for a dataset. It is typically used along with group_by() to provide aggregate summaries for each group in a dataset.

Key Points:

- summarize() collapses a dataset into a single row or a few rows by calculating aggregate values.
- You can pass multiple summary functions to summarize() (like mean(), sum(), min(), max(), etc.).
- When combined with group_by(), it generates summaries for each group rather than the entire dataset.

Example 1: Basic summarize() Without Grouping

Summarize the entire dataset (e.g., find the average mpg for the whole dataset).

```
library(dplyr)

# Summarize the entire dataset to get the average mpg
mtcars %>%
  summarize(avg_mpg = mean(mpg))

# Output:
# avg_mpg
#  20.09062
```

In this example, summarize() returns a single row with the average miles per gallon (mpg) across all cars.

Example 2: Using group_by() and summarize()

When used with group_by(), summarize() calculates the summary statistics for each group.

```
# Group by number of cylinders (cyl) and calculate average mpg and hp
mtcars %>%
  group_by(cyl) %>%
  summarize(
    avg_mpg = mean(mpg),
    avg_hp = mean(hp)
```

```
  )

# Output:
# # A tibble: 3 × 3
#     cyl avg_mpg avg_hp
#   <dbl>   <dbl>  <dbl>
# 1     4    26.7   82.6
# 2     6    19.7  122.3
# 3     8    15.1  209.2
```

Example 3: Multiple Summary Statistics

You can compute multiple summaries at once, like the mean, median, and standard deviation.

```
# Group by number of cylinders and compute multiple statistics
mtcars %>%
  group_by(cyl) %>%
  summarize(
    avg_mpg = mean(mpg),
    median_mpg = median(mpg),
    sd_mpg = sd(mpg)
  )

# Output:
# # A tibble: 3 × 4
#     cyl avg_mpg median_mpg sd_mpg
#   <dbl>   <dbl>      <dbl>  <dbl>
# 1     4    26.7       26     4.51
# 2     6    19.7       19.7   1.45
# 3     8    15.1       15.2   2.56
```

Example 4: Summarizing More than One Variable

You can summarize more than one variable by including multiple columns in the `summarize()` call.

```
# Group by gear and summarize both mpg and hp
mtcars %>%
  group_by(gear) %>%
  summarize(
    avg_mpg = mean(mpg),
    avg_hp = mean(hp),
    count = n()  # Count the number of rows per group
```

```
  )


# Output:
# # A tibble: 3 × 4
#    gear avg_mpg avg_hp count
#   <dbl>   <dbl>  <dbl> <int>
# 1     3    16.1   176.    15
# 2     4    24.5   89.5    12
# 3     5    21.4   195.     5
```

Example 5: summarize() with n() to Count Rows

You can use n() within summarize() to count the number of observations in each group.

```
# Count how many cars are in each cylinder group
mtcars %>%
  group_by(cyl) %>%
  summarize(count = n())

# Output:
# # A tibble: 3 × 2
#     cyl count
#   <dbl> <int>
# 1     4    11
# 2     6     7
# 3     8    14
```

Example 6: Summarizing with Conditional Logic

You can use conditional statements inside summarize().

```
# Summarize and count cars with mpg > 20 for each cylinder group
mtcars %>%
  group_by(cyl) %>%
  summarize(count_high_mpg = sum(mpg > 20))

# Output:
# # A tibble: 3 × 2
#     cyl count_high_mpg
#   <dbl>          <int>
# 1     4             11
# 2     6              4
```

6. count() Function

In dplyr, the count() function is used to count the number of observations for each group of values in one or more columns. It is a combination of group_by() and summarize(n = n()), making it a simple and quick way to count the occurrences of unique values or combinations of values in a dataset.

Syntax:

```
count(df, vars, wt = NULL, sort = FALSE, name = "n")
```

- df: The dataset (data frame or tibble).
- vars: The variables to group by (the columns whose values you want to count).
- wt: An optional weighting variable. It can be used to count weighted observations.
- sort: If TRUE, sorts the output in descending order of the count.
- name: The name of the new column containing the counts (default is n).

Example 1: Count Occurrences of a Single Column

Let's count how many times each number of cylinders (cyl) appears in the mtcars dataset.

```
library(dplyr)

# Count the occurrences of each value in the 'cyl' column
mtcars %>%
  count(cyl)

# Output:
#    cyl  n
# 1    4 11
# 2    6  7
# 3    8 14
```

Example 2: Count Combinations of Two Columns

You can count how often combinations of values appear. Let's count the number of cars for each combination of cyl and gear.

```
# Count the combinations of 'cyl' and 'gear'
mtcars %>%
```

```
  count(cyl, gear)

# Output:
#   cyl gear  n
# 1    4    3  1
# 2    4    4  8
# 3    4    5  2
# 4    6    3  2
# 5    6    4  4
# 6    6    5  1
# 7    8    3 12
# 8    8    5  2
```

Example 3: Count with Sorting

To sort the results by the count in descending order, use the `sort = TRUE` argument.

```
# Count occurrences of 'cyl' and sort by the count in descending order
mtcars %>%
  count(cyl, sort = TRUE)

# Output:
#   cyl  n
# 1    8 14
# 2    4 11
# 3    6  7
```

Example 4: Using Weighted Counts

The `wt` argument allows you to perform weighted counts. For example, you can use the `hp` (horsepower) column as weights.

```
# Weighted count using the 'hp' column
mtcars %>%
  count(cyl, wt = hp)

# Output:
#   cyl      n
# 1    4   1279
# 2    6   1394
# 3    8   4943
```

Example 5: Rename the Count Column

You can rename the count column using the `name` argument.

```
# Rename the count column to 'count'
mtcars %>%
  count(cyl, name = "count")

# Output:
#   cyl  count
# 1    4     11
# 2    6      7
# 3    8     14
```

Exercises on data.table

`data.table` is an R package that extends `data.frame`, making data manipulation faster and more memory-efficient, especially for large datasets. It's widely used for data manipulation tasks in R, providing a concise syntax and powerful features for filtering, aggregating, joining, and modifying data. Here's a detailed overview of its functions, along with examples and exercises.

Key Features of `data.table`

1. Efficiency: Fast data manipulation, even with large datasets.
2. Syntax: Simplified and expressive syntax for common tasks.
3. Grouping and Aggregation: Easy and efficient operations on grouped data.
4. Key-based Subsetting: Efficient sorting and filtering using keys.
5. Memory Efficiency: Lower memory footprint compared to `data.frame`.

Basic Syntax

The syntax of `data.table` is simple but very powerful. It's structured in three parts: i, j, by, similar to SQL's WHERE, SELECT, GROUP BY:

```
DT[i, j, by]
```

- i: Used for subsetting rows.
- j: Used for selecting/transforming columns.
- by: Used for grouping rows for aggregation.

Converting a data.frame to data.table

```
library(data.table)

# Convert data.frame to data.table
dt <- as.data.table(mtcars)
```

1. Selecting Rows (i)

Row selection is done using the i parameter, similar to filtering in dplyr::filter().

Example: Selecting cars with more than 6 cylinders:

```
dt[cyl > 6]
```

2. Selecting Columns (j)

Column selection is done using j, where columns can be referred to by name.

Example: Selecting specific columns (e.g., mpg and hp):

```
dt[, .(mpg, hp)]
```

3. Adding or Modifying Columns (j)

Use := to add or modify columns within a data.table.

Example: Adding a new column for weight in kilograms:

```
dt[, wt_kg := wt * 453.592]
```

Example: Modifying an existing column:

```
dt[, mpg := mpg * 0.425144]  # Convert mpg to km/l
```

4. Grouping (by)

Aggregation is easy with by, similar to dplyr::group_by().

Example: Calculating the average mpg by the number of cylinders (cyl):

```
dt[, .(avg_mpg = mean(mpg)), by = cyl]
```

5. Chaining

You can chain multiple operations together for concise and readable code.

Example: Filter, select, and aggregate in one go:

```
dt[cyl > 6, .(avg_mpg = mean(mpg), avg_hp = mean(hp)), by = gear]
```

6. Sorting

The setorder() function allows you to sort a data.table.

Example: Sorting by mpg in descending order:

```
setorder(dt, -mpg)
```

7. Joins

data.table allows fast joins, similar to SQL joins.

Example: Inner join of two data.tables on a common column:

```
dt1 <- data.table(id = 1:5, x = letters[1:5])
dt2 <- data.table(id = 3:7, y = letters[3:7])

# Perform an inner join on 'id'
dt1[dt2, on = "id"]
```

Exercises on reshape2

The reshape2 package in R provides a set of functions to easily "melt" and "cast" data between wide and long formats, making it ideal for reshaping and aggregating data.

Here are some key functions in reshape2:

- melt(): Converts wide data into long format.

- dcast(): Converts long data back to wide format, and allows aggregating data while reshaping.

```
install.packages("reshape2")
library(reshape2)
```

Data Example

We'll use the mtcars dataset for the exercises. Let's first convert it into a data.frame for better compatibility:

```
data("mtcars")
df <- as.data.frame(mtcars)
```

Exercise 1: Melting Data (Wide to Long)

Convert the mtcars dataset from wide to long format using the melt() function.

Task:

- Convert mtcars to a long format.
- Treat cyl (cylinders) as the identifying variable.

```
# Melt the mtcars dataset, keeping 'cyl' as an ID variable
long_mtcars <- melt(df, id.vars = "cyl")
head(long_mtcars)

# Output (First 6 rows):
#   cyl variable value
# 1   6      mpg  21.0
# 2   6      mpg  21.0
# 3   6      mpg  22.8
# 4   6      mpg  21.4
# 5   6      mpg  18.1
# 6   6      mpg  19.2
```

Exercise 2: Aggregating Data Using dcast()

Now, let's convert the long-form data back to wide format using dcast(), while also performing an aggregation.

Task:

- Use the melted data (long_mtcars) to find the mean value of each variable for each cyl group.
- Use dcast() to convert it back to wide format.

```
# Cast the data back to wide format, calculating the mean of each variable by
'cyl'
wide_mtcars <- dcast(long_mtcars, cyl ~ variable, mean)
wide_mtcars

# Output:
#   cyl      mpg     disp        hp      drat       wt    qsec        vs
am    gear    carb
# 1   4 26.66364 105.1364  82.63636  4.070909  2.285727 19.13727 0.9090909
0.7272727 4.090909 1.545455
# 2   6 19.74286 183.3143 122.28571  3.585714  3.117143 18.97714 0.5714286
0.4285714 3.857143 3.428571
# 3   8 15.10000 353.1000 209.21429  3.229286  3.999214 16.77286 0.0000000
0.1428571 3.285714 3.500000
```

Exercise 3: Multiple Identifiers

What if we want to keep multiple columns as identifiers? Let's use both cyl and gear as identifiers.

Task:

- Melt the mtcars dataset using both cyl and gear as identifying variables.

```
# Melt the dataset with multiple identifiers
long_mtcars_multi <- melt(df, id.vars = c("cyl", "gear"))
head(long_mtcars_multi)

# Output (First 6 rows):
#   cyl gear variable value
# 1   6    4      mpg  21.0
```

```
# 2   6    4      mpg  21.0
# 3   4    4      mpg  22.8
# 4   6    3      mpg  21.4
# 5   8    3      mpg  18.7
# 6   6    3      mpg  19.2
```

Exercise 4: Reshaping with Multiple Measures

Suppose we have multiple measures like mpg and hp that we want to analyze together.

Task:

- Cast the melted data back to wide format while calculating the sum of both mpg and hp for each cyl group.

```
# Melt specific columns of interest: 'mpg' and 'hp'
melted_mtcars <- melt(df, id.vars = "cyl", measure.vars = c("mpg", "hp"))

# Cast the data back to wide format, summing both 'mpg' and 'hp' by 'cyl'
dcast(melted_mtcars, cyl ~ variable, sum)

# Output:
#   cyl   mpg   hp
# 1   4 293.3  909
# 2   6 138.2  856
# 3   8 211.4 2929
```

Exercise 5: Handling NA values in dcast()

By default, dcast() returns NA values when there is no data for a particular combination. You can handle these NAs using the fill argument.

Task:

- Add some NA values to the dataset and fill them with 0 during the dcast() process.

```
# Introduce some NA values into the dataset
df$mpg[1:5] <- NA

# Melt and cast the data while filling NA values with 0
```

```
long_mtcars_na <- melt(df, id.vars = "cyl")
wide_mtcars_na <- dcast(long_mtcars_na, cyl ~ variable, mean, fill = 0)
wide_mtcars_na

# Output:
#   cyl      mpg     disp        hp      drat       wt    qsec        vs
am    gear    carb
# 1   4 23.09091 105.1364  82.63636  4.070909  2.285727 19.13727 0.9090909
0.7272727 4.090909 1.545455
# 2   6 17.94286 183.3143 122.28571  3.585714  3.117143 18.97714 0.5714286
0.4285714 3.857143 3.428571
# 3   8 13.77500 353.1000 209.21429  3.229286  3.999214 16.77286 0.0000000
0.1428571 3.285714 3.500000
```