1.3.1 Data Manipulation using dplyr

`dplyr` is a cornerstone of the tidyverse in R, a suite of packages designed for data science that makes data manipulation straightforward and intuitive.

It provides a concise and consistent set of tools (known as verbs) for manipulating datasets in a way that is both user-friendly and computationally efficient.

This section of the course will explore `dplyr`'s functionality through the analysis of the Tableau Superstore dataset, offering insights into sales data through tidy data principles.

Setting Up

Before diving into `dplyr`, ensure it is installed and loaded into your R environment.

Also install the reader package so that we can read csv files.

```
install.packages("dplyr")
install.packages("readr")

# Alternatively, we can install the entire tidyverse (includes dplyr and readr)
#install.packages("tidyverse")


# In R, you can conditionally install packages if they're not already available using th

# if (!requireNamespace("dplyr", quietly = TRUE)) {
#   install.packages("dplyr")
# }
# library(dplyr)
```

Load the installed libraries

```
# use a separate cell to load the library to accommodate for the eval=FALSE
library(dplyr)
```

```
Attaching package: 'dplyr'

The following objects are masked from 'package:stats':

    filter, lag
```

```
The following objects are masked from 'package:base':

    intersect, setdiff, setequal, union
```

```r
library(readr)
```

Core `dplyr` Verbs

`dplyr` revolves around five primary functions, often referred to as verbs, for most data manipulation tasks:

1. **`select()`** - Picks columns by name.
2. **`filter()`** - Subsets rows based on matching conditions.
3. **`arrange()`** - Changes the ordering of rows.
4. **`mutate()`** - Adds new columns or transforms existing ones.
5. **`summarize()`** - Reduces each group to a smaller summary statistic.
6. **`group_by()`** - Grouping data for aggregation operations.

Open CSV File

```r
file_path <- "data\\superstore.csv"
superstore <- read_csv(file_path, show_col_types = FALSE)
```

Selecting Columns with `select()`

The **`select()`** function is used to select specific columns from a dataset.

The first argument to **`select()`** is the name of the dataset from which you want to select columns.

After the dataset name, you can specify any number of column names that you want to include in your resulting dataset.

The column names should be separated by commas.

Example

The command below creates a new dataset named **`selected_columns`** that contains only the **`Order ID`**, **`Sales`**, and **`Profit`** columns from the **`superstore`** dataset.

```r
selected_columns <- select(superstore, `Order ID`, Sales, Profit)
head(selected_columns)
```

```
# A tibble: 6 × 3
  `Order ID`      Sales  Profit
  <chr>           <dbl>   <dbl>
1 CA-2016-152156 262.     41.9
2 CA-2016-152156 732.    220.
3 CA-2016-152156 732.    220.
4 CA-2016-138688  14.6     6.87
```

```
5 CA-2016-138688  14.6    6.87
6 US-2015-108966 958.  -383.
```

Subsetting Rows with filter()

The `filter()` function from the `dplyr` package is used to extract a subset of rows from a data frame or tibble that meet specific criteria.

It is akin to SQL's `WHERE` clause and base R's subset operation but is more intuitive and consistent within the `tidyverse` framework.

Syntax

The basic syntax of the `filter()` function is as follows:

```
filter(.data, ..., .preserve = FALSE)
```

- `.data`: The dataset from which you want to subset rows. This could be a data frame or a tibble.
- `...`: One or more conditions that rows must meet to be included in the output. These conditions are written using column names directly as if they were variables, thanks to `dplyr`'s non-standard evaluation. Conditions can be combined using logical operators like `&` (and), `|` (or), and `!` (not).

The standard logical operators can be used:

Examples	
<	less than
\	great than
<=	less than or equal
=	greater than or equal
==	equal to
!=	not equal to
|	entry wise or
||	or
!	not
&	entry wise and
&&	and
xor(a,b)	exclusive or

The command below filters the rows in the superstore dataset to return rows where the category is "Technology":

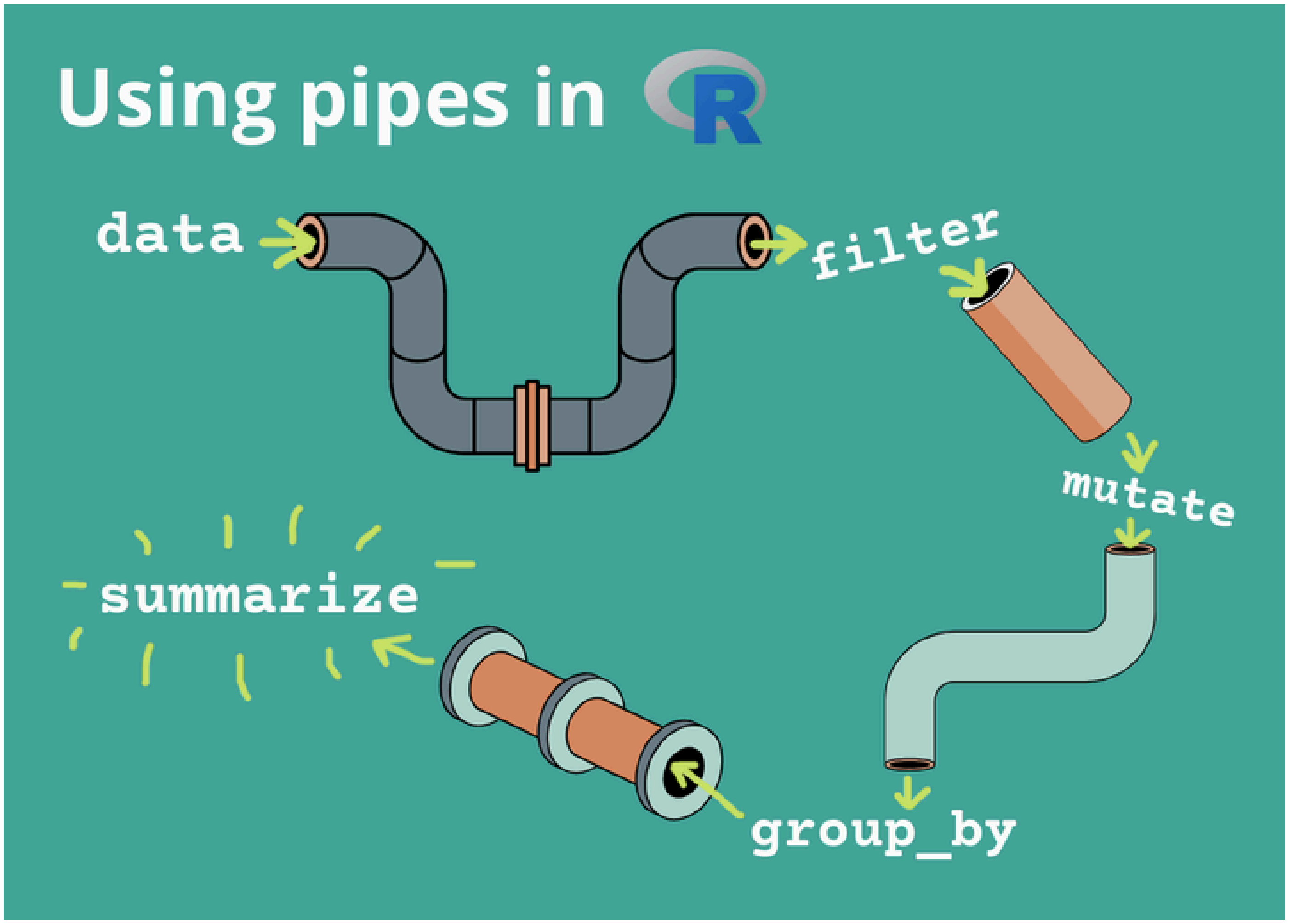
```r
technology_sales <- filter(superstore, Category == "Technology")
head(technology_sales)
```

```
# A tibble: 6 × 21
  `Row ID` `Order ID`     `Order Date` `Ship Date` `Ship Mode`    `Customer ID`
     <dbl> <chr>          <chr>        <chr>       <chr>          <chr>
1        8 CA-2014-115812 6/9/2014     6/14/2014   Standard Class BH-11710
2       12 CA-2014-115812 6/9/2014     6/14/2014   Standard Class BH-11710
3       20 CA-2014-143336 8/27/2014    9/1/2014    Second Class   ZD-21925
4       27 CA-2016-121755 1/16/2016    1/20/2016   Second Class   EH-13945
5       36 CA-2016-117590 12/8/2016    12/10/2016  First Class    GH-14485
6       41 CA-2015-117415 12/27/2015   12/31/2015  Standard Class SN-20710
# ℹ 15 more variables: `Customer Name` <chr>, Segment <chr>, Country <chr>,
#   City <chr>, State <chr>, `Postal Code` <dbl>, Region <chr>,
#   `Product ID` <chr>, Category <chr>, `Sub-Category` <chr>,
#   `Product Name` <chr>, Sales <dbl>, Quantity <dbl>, Discount <dbl>,
#   Profit <dbl>
```

Multiple conditions can be combined. For instance, to select rows where the category is "Technology" and the profit is greater than 1000:

```r
high_profit_tech <- filter(superstore, Category == "Technology" & Profit > 1000)
View(high_profit_tech)
```

Enhancing Workflow with the Pipe Operator `%>%`



The pipe operator `%>%` takes the output of one expression and feeds it into the first argument of the next expression.

It's akin to saying "then" in natural language. This operator simplifies code, reduces the need for intermediate variables, and enhances readability.

Let's apply the pipe operator to the previous examples of `select` and `filter` functions, showcasing its practical benefits.

Subsetting Columns with select() and the Pipe Operator

Without the pipe operator:

```
selected_columns <- select(superstore, `Order ID`, Sales, Profit)
```

With the pipe operator:

```
superstore %>%
  select(`Order ID`, Sales, Profit) -> selected_columns
```

By using the pipe operator, the operation's flow becomes clearer: "Take the `superstore` dataset, then select the `Order ID`, `Sales`, and `Profit` columns." This format is more intuitive, especially as operations become more complex.

Subsetting Rows with filter() and the Pipe Operator

Without the pipe operator:

```
technology_sales <- filter(superstore, Category == "Technology")
```

With the pipe operator:

```
superstore %>%
  filter(Category == "Technology") -> technology_sales
```

This example shows how the pipe operator can streamline operations, making it explicit that we are filtering the `superstore` dataset for technology products. The operation reads naturally from left to right, mirroring how we might describe the process verbally.

Combining Operations

One of the most powerful aspects of the pipe operator is its ability to combine multiple data manipulation steps into a single, coherent pipeline. Here's how you can combine `filter()` and `select()` operations:

```
superstore %>%
  filter(Category == "Technology") %>%
  select(`Order ID`, Sales, Profit) -> tech_sales_summary
```

This code snippet clearly illustrates the sequence of operations: "Take the `superstore` dataset, filter for technology products, and then select specific columns." It demonstrates how the pipe operator facilitates the creation of readable and concise code, especially for complex data manipulation workflows.

The `arrange()` function in `dplyr` is a straightforward yet powerful tool for sorting data frames or tibbles based on one or more columns. It allows users to reorder rows in ascending or descending order, making it easier to analyze data, spot trends, or prepare data for reporting. Here's a brief content segment on the `arrange()` function that you can include in your chapter:

Sorting Data with arrange()

The `arrange()` function from the `dplyr` package enables you to sort a dataset in ascending or descending order based on the values of specified columns.

This functionality is essential for data preparation and analysis, as it allows you to quickly organize your data in a meaningful way.

Syntax

The basic syntax of the `arrange()` function is as follows:

```
arrange(.data, ..., .by_group = FALSE)
```

- `.data`: The dataset you want to sort. This can be a data frame or a tibble.
- `...`: One or more columns to sort by. By default, `arrange()` sorts the data in ascending order. To sort in descending order, wrap the column name(s) with the `desc()` function.
- `.by_group`: A logical value indicating whether to sort within groups. This is relevant when you've used `group_by()` prior to `arrange()`.

Examples

Sorting in Ascending Order

To sort the `superstore` dataset by the `Sales` column in ascending order:

```
superstore %>%
  arrange(Sales) -> superstore_sorted
```

Sorting in Descending Order

To sort the same dataset by `Profit` in descending order:

```
superstore %>%
  arrange(desc(Profit)) -> superstore_sorted_desc
```

Sorting by Multiple Columns

You can also sort by multiple columns. For example, to sort `superstore` first by `Category` (ascending) and then by `Sales` (descending):

```
superstore %>%
  arrange(Category, desc(Sales)) -> superstore_sorted_multi
```

Practical Considerations

- **Handling Ties:** When sorting by multiple columns, `arrange()` uses the second column to break ties in the first, the third to break ties in the second, and so on.
- **Use with Grouping:** When used after `group_by()`, `arrange()` will sort within each group. This is particularly useful for grouped analyses.
- **Efficiency:** Sorting can be computationally intensive, especially for large datasets. However, `dplyr` is optimized for performance, making `arrange()` suitable for most data sizes.

Transforming Data with mutate()

The `mutate()` function from the `dplyr` package is essential for data transformation tasks.

It allows you to add new variables to a dataset or change existing ones while preserving the original rows.

`mutate()` is particularly powerful for creating calculated columns and applying functions to data columns.

Syntax

The basic syntax of the `mutate()` function is as follows:

```
mutate(.data, ..., .keep = "all")
```

- `.data`: The dataset you're modifying. This can be a data frame or a tibble.
- `...`: One or more expressions that create new columns or modify existing ones. New columns are added to the end of the dataset unless an existing column name is used, in which case, the column is modified.
- `.keep`: Controls which columns to keep when adding new ones. The default `"all"` keeps all existing columns.

Examples

Adding a New Column

To add a `ProfitMargin` column to the `superstore` dataset, calculated as profit divided by sales:

```
superstore %>%
  mutate(ProfitMargin = Profit / Sales) -> superstore_with_margin
```

Modifying an Existing Column

To update the `Sales` column to reflect sales in thousands:

```
superstore %>%
  mutate(Sales = Sales / 1000) -> superstore_sales_in_thousands
```

Using Multiple Transformations

`mutate()` can also perform multiple transformations at once. For example, adding both a `ProfitMargin` and a `SalesCategory` column:

```
superstore %>%
  mutate(
    ProfitMargin = Profit / Sales,
    SalesCategory = ifelse(Sales > 1000, "High", "Low")
  ) -> superstore_enhanced
```

Conditional Mutations

The `case_when()` function in `dplyr` is useful for creating new variables based on complex, multiple conditionals. It's akin to a series of if-else statements but is vectorized and integrates seamlessly within `dplyr`'s syntax for data manipulation.

For example, we can categorize the "Sales" in the Superstore dataset into different levels ("Low", "Medium", "High", "Very High") based on the sales amount.

We can use `case_when()` within a `mutate()` call to accomplish this.

```r
library(dplyr)

# Assuming 'superstore' is wer dataset
superstore <- superstore %>%
  mutate(SalesLevel = case_when(
    Sales <= 100  ~ "Low",
    Sales > 100 & Sales <= 500  ~ "Medium",
    Sales > 500 & Sales <= 2000 ~ "High",
    TRUE ~ "Very High" # 'TRUE' acts as the default case (like 'else')
  ))

superstore %>% select(Sales, SalesLevel)
```

```
# A tibble: 9,996 × 2
    Sales SalesLevel
    <dbl> <chr>     
 1 262.   Medium    
 2 732.   High      
 3 732.   High      
 4  14.6  Low       
 5  14.6  Low       
 6 958.   High      
 7  22.4  Low       
 8  48.9  Low       
 9   7.28 Low       
10 907.   High      
# ℹ 9,986 more rows
```

In this example, `SalesLevel` is a new column added to the `superstore` dataset. The `case_when()` function checks each row's "Sales" value against the conditions provided and assigns a corresponding category:

- If "Sales" is less than or equal to 100, `SalesLevel` is "Low".
- If "Sales" is greater than 100 but less than or equal to 500, `SalesLevel` is "Medium".
- If "Sales" is greater than 500 but less than or equal to 2000, `SalesLevel` is "High".
- If none of the above conditions are met, `SalesLevel` is "Very High".

This categorization can be particularly useful for subsequent analyses, such as grouping the data by `SalesLevel` to calculate the average profit or discount for each category. It simplifies the analysis of data across different segments of sales amounts.

Remember, the conditions in `case_when()` are evaluated in order, and the first true condition has its corresponding value assigned to the row. This makes it essential to order conditions correctly, especially when they might overlap.

Summarizing Data with `summarize()`

`summarize()` collapses a data frame to a single row summary per group (when combined with `group_by()`). To find the average sales per category:

```
        average_sales <- superstore %>% group_by(Category) %>% summarize(AverageSales = mean(Sal
        average_sales
```

```
# A tibble: 3 × 2
  Category        AverageSales
  <chr>                  <dbl>
1 Furniture               350.
2 Office Supplies         119.
3 Technology              453.
```

Grouping and Summarizing

Understanding `group_by()` in conjunction with `summarize()` allows for powerful grouped summaries. `summarize()` collapses a data frame to a single row summary per group (when combined with `group_by()`). For instance, calculating total sales by region:

```
        sales_by_region <- superstore %>% group_by(Region) %>% summarize(TotalSales = sum(Sales)
        sales_by_region
```

```
# A tibble: 4 × 2
  Region  TotalSales
  <chr>        <dbl>
1 Central    501240.
2 East       678781.
3 South      392454.
4 West       725472.
```

To find the average sales per category:

```
        superstore %>% group_by(Category,Region) %>% summarize(AverageSales = mean(Sales))
```

```
`summarise()` has grouped output by 'Category'. You can override using the
`.groups` argument.

# A tibble: 12 × 3
# Groups:   Category [3]
   Category        Region  AverageSales
   <chr>           <chr>          <dbl>
 1 Furniture       Central         341.
 2 Furniture       East            347.
 3 Furniture       South           354.
 4 Furniture       West            357.
 5 Office Supplies Central         117.
 6 Office Supplies East            120.
 7 Office Supplies South           126.
 8 Office Supplies West            116.
 9 Technology      Central         406.
```

```
10 Technology     East            495.
11 Technology     South           508.
12 Technology     West            421.
```

```
library(dplyr)
library(tidyr)

superstore %>%
  group_by(Category, Region) %>%
  summarise(TotalSales = sum(Sales)) %>%
  ungroup() %>%
  spread(key = Region, value = TotalSales)
```

```
`summarise()` has grouped output by 'Category'. You can override using the
`.groups` argument.

# A tibble: 3 × 5
  Category        Central    East   South    West
  <chr>             <dbl>   <dbl>   <dbl>   <dbl>
1 Furniture       163797. 208291. 118031. 252613.
2 Office Supplies 167026. 205516. 125651. 220868.
3 Technology      170416. 264974. 148772. 251992.
```

In the code snippet provided, `ungroup()` and `spread()` are functions from the `dplyr` and `tidyr` packages in R, respectively, and they perform the following operations:

- `ungroup()`: This function is used to remove the grouping structure from a data frame. In the context of the provided code, `group_by(Category, Region)` initially groups the data for the summarization operation. After `summarise()` is applied, `ungroup()` is called to remove the grouping metadata. This is a good practice because it prevents accidental grouping from influencing subsequent operations that should not be aware of these groupings.

- `spread()`: The function `spread()` is part of the `tidyr` package and is used to widen a data frame, turning key-value pairs into a full-fledged data frame with a column for each unique key and corresponding values filled in. In the provided code, `spread(key = Region, value = TotalSales)` takes the `Region` column (the key) and creates new columns for each unique region. It then fills these columns with the values from the `TotalSales` column corresponding to each region. The result is a wider format data frame where each region has its own column and the cells contain the total sales figures for that region and category.

The combination of these two functions is part of a typical data transformation workflow in R, where data is grouped and summarized before being reshaped for further analysis or reporting.

Grouping in Base R

We can group data using base r only (without the need for dplyr library), this can be done using the data using the Aggregate function, but the dpyr alternative provides more options and flexibility.

```
aggregate(Sales ~ Category + Region, data = superstore, FUN = sum)
```

```
          Category  Region    Sales
1        Furniture Central 163797.2
2  Office Supplies Central 167026.4
3       Technology Central 170416.3
4        Furniture    East 208291.2
5  Office Supplies    East 205516.1
6       Technology    East 264974.0
7        Furniture   South 118030.6
8  Office Supplies   South 125651.3
9       Technology   South 148771.9
10       Furniture    West 252612.7
11 Office Supplies    West 220867.9
12      Technology    West 251991.8
```

```
# This command reads as:
# aggregate sales by category and region from superstore data
```

Renaming Columns with `rename()`

Consistent and descriptive column names are vital for understandable and maintainable code. The `rename()` function is used to change column names in wer dataset for clarity or consistency.

Example: Renaming Columns for Clarity

Imagine we find some of the column names in the Superstore dataset to be unclear or not aligned with wer analysis conventions. we can use `rename()` to change these names to something more descriptive.

```
# Renaming 'Ship Date' to 'Shipment Date' and 'Segment' to 'Customer Segment'
superstore_renamed <- superstore %>%
  rename(`Shipment Date` = `Ship Date`,
         `Customer Segment` = Segment)

# This changes the column names, making them potentially more understandable for others
```

In this example, `Ship Date` is renamed to `Shipment Date` for clarity, and `Segment` to `Customer Segment` to more accurately describe the data contained in that column.

Self Study Material

Finding and Handling Duplicates

Duplicate rows in a dataset can skew wer analysis, leading to inaccurate results. The `distinct()` function helps in identifying and removing these duplicates, ensuring each data point is unique.

Find Duplicate Orders

Suppose we've noticed some orders might have been recorded multiple times in the Superstore dataset. We can get the duplicate records by grouping values as following:

Counting occurrences of each combination of 'Order ID' and 'Product ID'

```
        duplicates <- superstore %>%
          group_by(`Order ID`, `Product ID`) %>%   count() %>%
          filter(n > 1) # Filtering to keep only those with more than one occurrence duplicates
```

View the Actual Duplicate Rows

If we want to see the actual rows that are duplicates, we can then filter the original dataset based on this information

```
        actual_duplicates <- superstore %>%
          semi_join(duplicates, by = c("Order ID", "Product ID"))

        actual_duplicates
```

```
# A tibble: 20 × 22
   `Row ID` `Order ID`     `Order Date` `Ship Date` `Ship Mode`    `Customer ID`
      <dbl> <chr>          <chr>        <chr>       <chr>          <chr>
 1        2 CA-2016-152156 11/8/2016    11/11/2016  Second Class   CG-12520
 2       NA CA-2016-152156 11/8/2016    11/11/2016  Second Class   CG-12520
 3        3 CA-2016-138688 6/12/2016    6/16/2016   Second Class   DV-13045
 4        3 CA-2016-138688 6/12/2016    6/16/2016   Second Class   DV-13045
 5      351 CA-2016-129714 9/1/2016     9/3/2016    First Class    AB-10060
 6      353 CA-2016-129714 9/1/2016     9/3/2016    First Class    AB-10060
 7      431 US-2016-123750 4/15/2016    4/21/2016   Standard Class RB-19795
 8      432 US-2016-123750 4/15/2016    4/21/2016   Standard Class RB-19795
 9     1301 CA-2016-137043 12/23/2016   12/25/2016  Second Class   LC-17140
10     1302 CA-2016-137043 12/23/2016   12/25/2016  Second Class   LC-17140
11     3184 CA-2017-152912 11/9/2017    11/12/2017  Second Class   BM-11650
12     3185 CA-2017-152912 11/9/2017    11/12/2017  Second Class   BM-11650
13     3406 US-2014-150119 4/23/2014    4/27/2014   Standard Class LB-16795
14     3407 US-2014-150119 4/23/2014    4/27/2014   Standard Class LB-16795
15     6499 CA-2015-103135 7/24/2015    7/28/2015   Standard Class SS-20515
16     6501 CA-2015-103135 7/24/2015    7/28/2015   Standard Class SS-20515
17     7882 CA-2017-118017 12/3/2017    12/6/2017   Second Class   LC-16870
18     7883 CA-2017-118017 12/3/2017    12/6/2017   Second Class   LC-16870
19     9169 CA-2016-140571 3/15/2016    3/19/2016   Standard Class SJ-20125
20     9170 CA-2016-140571 3/15/2016    3/19/2016   Standard Class SJ-20125
# ℹ 16 more variables: `Customer Name` <chr>, Segment <chr>, Country <chr>,
#   City <chr>, State <chr>, `Postal Code` <dbl>, Region <chr>,
#   `Product ID` <chr>, Category <chr>, `Sub-Category` <chr>,
#   `Product Name` <chr>, Sales <dbl>, Quantity <dbl>, Discount <dbl>,
#   Profit <dbl>, SalesLevel <chr>
```

Removing the duplicates

We can use `distinct()` to remove these duplicate entries based on specific columns that would uniquely identify an order, such as `Order ID` and `Product ID`.

```
library(dplyr)

cat("Number of rows before removing duplicate records:", nrow(superstore))
```

```
Number of rows before removing duplicate records: 9996
```

```
# Removing duplicate rows based on 'Order ID' and 'Product ID'
superstore <- superstore %>%
  distinct(`Order ID`, `Product ID`, .keep_all = TRUE)
cat("Number of rows after removing duplicate records:", nrow(superstore))
```

```
Number of rows after removing duplicate records: 9986
```

This operation will retain only unique orders, eliminating any rows where the combination of `Order ID` and `Product ID` is repeated.

In the `distinct()` function from the `dplyr` package in R, the `.keep_all` argument determines whether to keep all columns or only the ones used for identifying unique rows.

When `.keep_all = TRUE`, it means that all columns from the original data frame (`superstore` in this case) will be retained in the output, regardless of whether they were used to identify unique rows or not. This is useful when we want to keep all the original columns in the data frame but remove duplicate rows based on specific columns.

Here's what happens in wer example:

- The `distinct()` function is applied to the `superstore` data frame.
- It looks for unique combinations of values in the columns `Order ID` and `Product ID`.
- If there are duplicate rows with the same combination of values in these columns, only one of them is kept in the output.
- However, with `.keep_all = TRUE`, all other columns in the original data frame (`superstore`) are retained in the output, even if they were not used for identifying unique rows.

Finding and Handling Null Values

To find null values in the `superstore` dataset, we can use the `is.na()` function in R. This function returns a logical vector indicating whether each element of a vector or data frame is missing (NA) or not.

Here's how wecan find null values in the `superstore` dataset:

```
# Check for null values in the entire dataset
null_values <- is.na(superstore)
```

```
# Count the number of null values in each column
null_counts <- colSums(null_values)

# Print the column names and corresponding counts of null values
print(null_counts)
```

```
      Row ID      Order ID    Order Date     Ship Date     Ship Mode 
           1             0             0             0             0 
 Customer ID Customer Name       Segment       Country          City 
           0             0             0             0             0 
       State   Postal Code        Region    Product ID      Category 
           0             0             0             0             0 
Sub-Category  Product Name         Sales      Quantity      Discount 
           0             0             0             0             0 
      Profit    SalesLevel 
           0             0 
```

```
# Sum up all the TRUE values (null values) across all columns
total_null_count <- sum(null_values)

# Print the total number of null values
cat("Sum of all nulls in all columns:", total_null_count)
```

```
Sum of all nulls in all columns: 1
```

In this code:

- `is.na(superstore)` creates a logical data frame where `TRUE` indicates a null value (NA) and `FALSE` indicates a non-null value.
- `colSums(null_values)` calculates the sum of `TRUE` values (which represent null values) for each column, giving you the count of null values in each column.
- Finally, `print(null_counts)` prints the column names and corresponding counts of null values.

This will give usa summary of null values in each column of the `superstore` dataset. We can then decide how to handle these null values based on your analysis requirements.

Handling Null Values

Handling null values in a dataset depends on the nature of your data and the analysis you're conducting. Here are some common strategies for handling null values in R:

1. **Remove rows or columns with null values**: If null values are present in only a small portion of your dataset, you may choose to simply remove the rows or columns containing null values using functions like `na.omit()` or `complete.cases()`. For example:

```
 # Remove rows with any null values
clean_data <- na.omit(superstore)

# Remove rows with any null values
```

```r
clean_data <- superstore[complete.cases(superstore), ]

# Identify columns with any null values
cols_with_null <- colSums(is.na(superstore)) > 0

# Subset the data frame to remove columns with any null values
clean_data <- superstore[, !cols_with_null]

# We can also use the dplyr filter function to filter out null values
data_without_NA <- superstore %>% filter(!is.na(Profit))
```

2. **Impute null values**: If null values are present in a significant portion of your dataset, you may choose to impute (replace) them with a specific value such as the mean, median, or mode of the column, or with a user-defined value. For example:

   First, we need to install the zoo package to able to use the na.fill function

```r
install.packages("zoo")
```

```r
# use a seperate cell to laod the library to accommodate for the eval=FALSE
library(zoo)
```

```
Attaching package: 'zoo'

The following objects are masked from 'package:base':

    as.Date, as.Date.numeric
```

```r
# Impute null values with the mean of the column
filled_data <- na.fill(superstore, "mean")

# Impute null values with a user-defined value
filled_data <- superstore
filled_data[is.na(filled_data)] <- 0
```

3. **Model-based imputation**: Use statistical models, such as linear regression or k-nearest neighbors (KNN), to predict missing values based on other variables in the dataset.

4. **Create a separate category for missing values**: If null values represent a meaningful category in your data, you can create a separate category or indicator variable for missing values.

5. **Domain-specific handling**: Depending on the domain and context of your data, you may have specific methods for handling null values. For example, in time series data, null values might be forward-filled or backward-filled.

6. **Use packages for missing data handling**: R provides several packages, such as `mice`, `missForest`, and `imputeTS`, that offer advanced methods for handling missing data.

Choose the appropriate method based on your specific dataset, analysis goals, and domain knowledge. It’s essential to carefully consider the implications of each method and the potential impact on your analysis results.