Introduction to Data Science Lab
Week -3&4

Topics

1. Data frames
2. CSV
3. TSV
4. Excel Files
5. XML
6. JSon
7. HTML

Data Frames:

Creating Data Frames

```
df <- data.frame(Name = c("Alice", "Bob","ALice","Bob"), Age = c(25, 30,32,39))
```

Viewing Data Frames

```
head(df)
tail(df)
str(df)
```

Accessing Data

```
df$Name
df[1, ]
df[, "Age"]
df[df$Age > 25, ]
```

Modifying Data Frames

```
df2 <- data.frame(Salary = c(50000, 60000))
df_combined <- cbind(df, df2)
df3 <- data.frame(Name = c("Charlie"), Age = c(28), Salary = c(55000))
df_extended <- rbind(df, df3)
```

Summary Statistics

```
summary(df)
mean(df$Age)
```

Subsetting and Filtering

```
subset(df, Age > 25)
```

Data Manipulation

```
library(dplyr)
df_filtered <- df %>% filter(Age > 25)
df_selected <- df %>% select(Name)
```

Data Transformation

```
df_transformed <- transform(df, Age = Age + 1)
```

Merging Data Frames

```
df1 <- data.frame(ID = c(1, 2), Value = c("A", "B"))
df2 <- data.frame(ID = c(1, 2), Score = c(90, 85))
df_merged <- merge(df1, df2, by = "ID")
```

Aggregating Data

```
aggregate(Age ~ Name, data = df, FUN = mean)
```

Combining and Splitting Data

```
split_df <- split(df, df$Name)
library(data.table)
dt1 <- data.table(a = 1:3, b = 4:6)
dt2 <- data.table(a = 7:9, b = 10:12)
dt_combined <- rbindlist(list(dt1, dt2))
```

Adding Two Data Frames

Assuming df1 and df2 have the same structure:

```
# Create example data frames
df1 <- data.frame(A = c(1, 2, 3), B = c(4, 5, 6))
df2 <- data.frame(A = c(7, 8, 9), B = c(10, 11, 12))

# Add the two data frames
df_sum <- df1 + df2
```

Subtracting Two Data Frames

```
# Subtract df2 from df1
df_diff <- df1 - df2
```

Multiplying Two Data Frames

```
# Multiply df1 and df2 element-wise
df_product <- df1 * df2
```

Dividing Two Data Frames

```
# Divide df1 by df2 element-wise
df_quotient <- df1 / df2
```

Arithmetic Operations on Specific Columns

You can perform arithmetic operations on specific columns of a data frame:

Adding a Constant to a Column

```
# Add 10 to column A
df1$A <- df1$A + 10
```

Performing Operations Between Columns

```
# Create a new column C which is the sum of columns A and B
df1$C <- df1$A + df1$B
```

Applying Functions to Columns

```
# Apply a function to each column
df1$A <- sqrt(df1$A)  # Square root of column A
```

Row-Wise Arithmetic Operations

You can apply arithmetic operations across rows using functions like rowSums() and rowMeans().

Calculating Row Sums

```
# Calculate the sum of each row
df1$row_sum <- rowSums(df1)
```

Calculating Row Means

```
# Calculate the mean of each row
df1$row_mean <- rowMeans(df1)
```

Applying Custom Functions

```
# Define a custom function
custom_func <- function(x) { sum(x) * 2 }
```

```
# Apply the custom function to each row
df1$row_custom <- apply(df1, 1, custom_func)
```

Using dplyr for Arithmetic Operations

The dplyr package provides a convenient way to perform arithmetic operations on data frames:

Installing and Loading dplyr

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

Adding New Columns

```
# Add a new column C which is the sum of A and B
df1 <- df1 %>%
  mutate(C = A + B)
```

Performing Operations with mutate

```
# Add 10 to column A and create a new column D
df1 <- df1 %>%
  mutate(D = A + 10)
```

Applying Functions to Columns

```
# Calculate the square root of column A and update it
df1 <- df1 %>%
  mutate(A = sqrt(A))
```

"Apply" Function

apply is used to apply a function over the margins (rows or columns) of a matrix or data frame.

Syntax:

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

- X is the data frame or matrix.
- MARGIN indicates whether to apply the function over rows (1) or columns (2).
- FUN is the function to apply.
- ... are additional arguments to pass to the function.

Applying Functions to Columns

```
# Create a data frame
df <- data.frame(A = c(1, 2, 3), B = c(4, 5, 6))

# Calculate the mean of each column
column_means <- apply(df, 2, mean)
print(column_means)
```

Applying Functions to Rows

```
# Calculate the sum of each row
row_sums <- apply(df, 1, sum)
print(row_sums)
```

Calculate the Standard Deviation of Each Column

```
# Calculate the standard deviation of each column
column_sd <- apply(df, 2, sd)

# Print the result
print(column_sd)
```

Calculate the Mean of Each Row

```
# Calculate the mean of each row
row_means <- apply(df, 1, mean)

# Print the result
print(row_means)
```

You can also use apply with custom functions.

Calculate the Range of Each Column

```
# Define a custom function to calculate range
range_func <- function(x) {
  return(max(x) - min(x))
}

# Apply the custom function to each column
column_ranges <- apply(df, 2, range_func)

# Print the result
print(column_ranges)
```

Applying a Function to Numeric Columns Only

```
# Create a data frame with numeric and non-numeric columns
df_mixed <- data.frame(A = c(1, 2, 3), B = c(4, 5, 6), C = c("x", "y", "z"))

# Apply a function to numeric columns only
numeric_df <- df_mixed[sapply(df_mixed, is.numeric)]
column_means_numeric <- apply(numeric_df, 2, mean)

# Print the result
print(column_means_numeric)
```

Normalize Data

```
# Create a data frame
df <- data.frame(A = c(10, 20, 30), B = c(40, 50, 60))

# Normalize columns (subtract mean and divide by standard deviation)
normalize <- function(x) {
  (x - mean(x)) / sd(x)
}

df_normalized <- apply(df, 2, normalize)
print(df_normalized)
```

Exercises:

1. Creating Data Frames

Exercise 1.1: Create a data frame named students with the following columns:

- StudentID: 101, 102, 103, 104
- Name: "John", "Emma", "Alex", "Sophia"
- Score: 88, 92, 79, 85

Exercise 1.2: Create a data frame sales with the following data:

- Product: "A", "B", "C", "D"
- UnitsSold: 150, 200, 175, 130
- PricePerUnit: 20.5, 15.0, 25.0, 18.0

2. Accessing Data Frames

Exercise 2.1: Access the Name column from the students data frame you created in Exercise 1.1.
Exercise 2.2: Retrieve the first two rows of the sales data frame from Exercise 1.2.
Exercise 2.3: Select all rows where UnitsSold is greater than 150 in the sales data frame.

3. Analyzing Data Frames

Exercise 3.1: Calculate the mean Score of students from the students data frame.
Exercise 3.2: Find the maximum PricePerUnit from the sales data frame.
Exercise 3.3: Create a new column in the sales data frame named TotalRevenue that calculates the total revenue for each product (i.e., UnitsSold multiplied by PricePerUnit).
Exercise 3.4: Use the summary() function to get a summary of the students data frame.

4. Data Manipulation

Exercise 4.1: Add a new row to the students data frame with the following data: StudentID = 105, Name = "Liam", Score = 90.
Exercise 4.2: Add a new column to the sales data frame named DiscountedPrice with a 10% discount applied to each PricePerUnit.
Exercise 4.3: Filter the students data frame to show only those students with a Score greater than or equal to 85.

5. Data Transformation and Reshaping

Exercise 5.1: Transform the students data frame by adding 5 points to each Score.
Exercise 5.2: Reshape the sales data frame from a wide format to a long format, where UnitsSold and PricePerUnit are melted into one column.

CSV Files

Reading CSV Files

```
# Read a CSV file into a data frame
data <- read.csv("filename.csv")
```

Reading with Custom Options:

```
# Read a CSV file with custom options
data <- read.csv("filename.csv", header = TRUE, sep = ",", stringsAsFactors = FALSE)
```

- header = TRUE: Indicates that the first row contains column names.
- sep = ",": Specifies the delimiter used in the CSV file (default is comma).
- stringsAsFactors = FALSE: Prevents automatic conversion of character strings to factors.

Writing CSV Files

```
# Write a data frame to a CSV file
write.csv(data, "output.csv", row.names = FALSE)
```

- row.names = FALSE: Prevents writing row names to the file.

Writing with Custom Options:

```
# Write a data frame with custom options
write.csv(data, "output.csv", quote = TRUE, sep = ",", na = "NA")
```

- quote = TRUE: Quotes character or factor columns.
- sep = ",": Specifies the delimiter (default is comma).
- na = "NA": Specifies the string used for missing values.

Reading CSV Files with readr Package

The readr package provides functions for reading and writing data that are often faster and more flexible.

Reading CSV with readr:

```
library(readr)
# Read a CSV file into a dataframe
data <- read_csv("filename.csv")
```

Writing CSV with readr:

```
# Write a dataframe to a CSV file
write_csv(data, "output.csv")
```

Basic Data Manipulation

Viewing Data:

```
# View the first few rows of the data
head(data)

# View the last few rows of the data
tail(data)

# Get the structure of the data
str(data)
```

Subsetting Data:

```
# Subset data based on a condition
subset_data <- subset(data, ColumnName > 100)

# Select specific columns
selected_columns <- data[, c("Column1", "Column2")]
```

Filtering and Sorting:

```
library(dplyr)

# Filter rows where ColumnName is greater than 100
filtered_data <- filter(data, ColumnName > 100)

# Arrange rows by ColumnName in ascending order
sorted_data <- arrange(data, ColumnName)

# Arrange rows by ColumnName in descending order
sorted_data_desc <- arrange(data, desc(ColumnName))
```

Adding and Modifying Columns:

```
# Add a new column with a constant value
data$new_column <- 10
```

```
# Modify an existing column
data$ColumnName <- data$ColumnName * 2
```

Removing Columns:

```
# Remove a column from the data frame
data$ColumnName <- NULL
```

Reading a Compressed CSV File:

```
# Read a compressed CSV file (e.g., .gz)
data <- read.csv(gzfile("filename.csv.gz"))
```

Writing a Compressed CSV File:

```
# Write a CSV file to a compressed format (e.g., .gz)
write.csv(data, gzfile("output.csv.gz"), row.names = FALSE)
```

Exercises

1.Reading Data from Files

Exercise 1.1: Load a CSV file named employee_data.csv into a data frame. Assume the file contains columns: EmployeeID, Name, Department, Salary.
Exercise 1.2: Load an Excel file named sales_data.xlsx into a data frame. Assume it has a sheet named "2024 Sales" with columns: Product, UnitsSold, Revenue.
Exercise 1.3: Read a tab-delimited file named customer_info.txt into a data frame. Assume it has columns: CustomerID, Name, PurchaseAmount.

2. Exploring and Accessing Data

Exercise 2.1: Display the first 10 rows of the data frame from Exercise 1.1.
Exercise 2.2: Retrieve and display the names of all unique departments in the data frame from Exercise 1.1.
Exercise 2.3: Find and display the highest salary in the data frame from Exercise 1.1.
Exercise 2.4: For the data frame from Exercise 1.2, calculate the total revenue from all products.

3. Data Filtering and Subsetting

Exercise 3.1: From the data frame in Exercise 1.1, filter and display all employees with a salary greater than $60,000.
Exercise 3.2: From the data frame in Exercise 1.2, filter the data to show products with UnitsSold greater than 100.
Exercise 3.3: For the data frame from Exercise 1.3, find all customers with a PurchaseAmount greater than $500.

4. Summarizing Data

Exercise 4.1: Calculate the average salary by department from the data frame in Exercise 1.1.
Exercise 4.2: Determine the product with the maximum revenue from the data frame in Exercise 1.2.
Exercise 4.3: For the data frame from Exercise 1.3, calculate the total purchase amount and the number of customers.

5. Merging and Aggregating Data

Exercise 5.1: Merge the data frame from Exercise 1.1 with another data frame named department_data which has columns Department, Manager. Merge by the Department column.
Exercise 5.2: Aggregate the data from Exercise 1.2 to find the total revenue and average revenue per product.
Exercise 5.3: Create a new data frame from Exercise 1.3 with additional information on whether PurchaseAmount is above or below $300 (add a new column HighValueCustomer).

TSV FIles

Reading TSV Files
To read a TSV file into R, you use the read.table() function with the sep argument set to "\t":

```
data <- read.table("filename.tsv", header = TRUE, sep = "\t",
stringsAsFactors = FALSE)
```

Writing TSV Files
To write a data frame to a TSV file, use the write.table() function with the sep argument set to "\t":

```
write.table(data, "output.tsv", sep = "\t", row.names = FALSE, quote = FALSE)
```

XML Files

XML (eXtensible Markup Language) is a format used to encode documents in a way that is both human-readable and machine-readable. XML files consist of elements, attributes, and nested structures. Each XML file starts with an XML declaration and may contain multiple nested elements.

Common Packages for XML in R

- XML: A package for parsing XML documents.
- xml2: A modern package for XML processing, providing simple and powerful tools to work with XML files.

Creation of XML Files
Using XML Package:

```
library(XML)

# Create XML content
doc <- newXMLDoc()
root <- newXMLNode("root", doc = doc)

# Add child nodes
child1 <- newXMLNode("child1", "Value 1", parent = root)
child2 <- newXMLNode("child2", "Value 2", parent = root)
child3 <- newXMLNode("child3", attrs = c( "attr_value"), parent = root)

# Save to file
saveXML(doc, file = "example.xml")
```

Accessing XML Files

Using XML Package:

```
library(XML)

# Load XML file
doc <- xmlParse("example.xml")

# Parse XML
root <- xmlRoot(doc)

# Access elements
child1_value <- xmlValue(root[["child1"]])
child2_value <- xmlValue(root[["child2"]])

# Access attributes
child3_attr <- xmlAttrs(root[["child3"]])
```

Parsing and Extracting Data:

Using XML Package:

```
library(XML)

# Load XML file
doc <- xmlParse("example.xml")
root <- xmlRoot(doc)

# Extract all child nodes
children <- xmlChildren(root)

# Extract and analyze specific data
child1_value <- xmlValue(children[["child1"]])
```

Transformation:

Using XML Package:

```
library(XML)

# Transform XML to a data frame
df <- xmlToDataFrame(nodes = getNodeSet(doc, "//child1"))
```

Exercises

1. Creation of XML Files

Exercise 1.1: Create an XML file named contacts.xml with the following structure:

```
<contacts>
  <contact id="1">
    <name>John Doe</name>
    <email>john@example.com</email>
  </contact>
  <contact id="2">
    <name>Jane Smith</name>
    <email>jane@example.com</email>
  </contact>
</contacts>
```

Exercise 1.2: Create an XML file named products.xml with the following structure:

```
<products>
  <product id="A1">
    <name>Product A</name>
    <price>10.99</price>
  </product>
  <product id="B2">
    <name>Product B</name>
    <price>20.99</price>
  </product>
</products>
```

2. Accessing XML Files

Exercise 2.1: Load the contacts.xml file and extract the email addresses of all contacts.

Exercise 2.2: Load the products.xml file and extract the names and prices of all products.

3. Analysis of XML Files

Exercise 3.1: Convert the contacts.xml file to a data frame in R with columns for id, name, and email.

Exercise 3.2: Convert the products.xml file to a dataframe in R with columns for id, name, and price.

JSON FIles

Working with JSON (JavaScript Object Notation) files in R is a common task for handling structured data. JSON is a lightweight data-interchange format that is easy for humans to read and write and easy for machines to parse and generate.

JSON Basics

- **Structure**: JSON files consist of key-value pairs. Keys are strings and values can be strings, numbers, arrays, objects, or booleans.
- **Syntax**: JSON objects are enclosed in curly braces {}, arrays are enclosed in square brackets [], and key-value pairs are separated by colons **:**

Common Packages for JSON in R

- jsonlite: A powerful package for parsing and generating JSON.
- rjson: Another package for working with JSON in R.

Creation of JSON Files

Using jsonlite Package:

```
library(jsonlite)
# Create a list
data <- list(
  name = "John Doe",
  age = 30,
  address = list(street = "123 Main St",  city = "Anytown"  ),
  phone_numbers = c("123-456-7890", "987-654-3210")
)

# Convert to JSON
json_data <- toJSON(data, pretty = TRUE)

# Write to file
write(json_data, file = "data.json")
```

Using rjson Package:

```
library(rjson)
# Create a list
data <- list(
  name = "John Doe",
  age = 30,
  address = list( street = "123 Main St",   city = "Anytown"  ),
  phone_numbers = c("123-456-7890", "987-654-3210")
)
```

```
# Convert to JSON
json_data <- toJSON(data)
print(json_data)

# Write to file
write(json_data, file = "data.json")
```

Accessing JSON Files
Using jsonlite Package:

```
library(jsonlite)

# Read JSON file
data <- fromJSON("data.json")
print(data)

# Access elements
name <- data$name
age <- data$age
address_city <- data$address$city
phone_numbers <- data$phone_numbers
```

Using rjson Package:

```
library(rjson)

# Read JSON file
data <- fromJSON(file = "data.json")
print(data)

# Access elements
name <- data$name
age <- data$age
address_city <- data$address$city
phone_numbers <- data$phone_numbers

# Convert to data frame
df <- as.data.frame(data)
```

Exercises

1. **Creation of JSON Files**
Exercise 1.1: Create a JSON file named employees.json with the following structure:

```
[
  {
    "id": 1,
    "name": "Alice",
    "department": "HR",
    "salary": 50000
  },
  {
    "id": 2,
    "name": "Bob",
    "department": "IT",
    "salary": 60000
  }
]
```

Exercise 1.2: Create a JSON file named books.json with the following structure:

```
{
  "books": [
    {
      "title": "To Kill a Mockingbird",
      "author": "Harper Lee",
      "year": 1960
    },
    {
      "title": "1984",
      "author": "George Orwell",
      "year": 1949
    }
  ]
}
```

2. **Accessing JSON Files**
Exercise 2.1: Load the employees.json file and extract the names of all employees.
Exercise 2.2: Load the books.json file and extract the titles and authors of all books.

3. **Analyzing JSON Files**
Exercise 3.1: Convert the employees.json file to a data frame in R with columns for id, name, department, and salary.
Exercise 3.2: Convert the books.json file to a data frame in R with columns for title, author, and year.

4. **Flattening and Working with JSON Arrays**
Exercise 4.1: Flatten the books.json file and access the title of the first book.
Exercise 4.2: Handle the JSON array from employees.json and compute the average salary of all employees.

HTML Tables:

Extracting and working with HTML tables from web pages in R involves several steps. You typically use web scraping techniques to access and parse HTML content, and then extract the tables.

The rvest package is commonly used for this purpose.

Extracting HTML Tables from Web Pages

Using rvest Package

Installation and Loading

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

Reading a Web Page

Use read_html() to load the HTML content from a web page.

```
url <- "https://example.com"
webpage <- read_html(url)
```

Extracting Tables

Use html_table() to extract tables from the HTML content. You can specify which table to extract if there are multiple.

```
# Extract all tables
tables <- html_nodes(webpage, "table")
table1 <- html_table(tables[[1]], fill = TRUE)  # Extract the first table
```

Viewing and Analyzing the Data

Convert the table into a data frame and analyze it.

```
# Convert to data frame
df <- as.data.frame(table1)

# View the first few rows
head(df)
```

Saving Extracted Data

Save to CSV

```
write.csv(df, "extracted_table.csv", row.names = FALSE)
```

Save to Excel

You can use the writexl package to save the data to an Excel file.

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

write_xlsx(df, "extracted_table.xlsx")
```

Example Code
Here is a complete example of extracting and analyzing a table from a web page:

```
library(rvest)

# Define the URL
url <- "https://example.com"

# Read the HTML content
webpage <- read_html(url)

# Extract the first table from the webpage
table_node <- html_node(webpage, "table")
df <- html_table(table_node, fill = TRUE)

# View the data
print(df)

# Save the data to a CSV file
write.csv(df, "extracted_table.csv", row.names = FALSE)
```

Exercises

1. **Extracting Tables**
Exercise 1.1: Extract the first table from the web page https://www.worldometers.info/coronavirus/. Save the table to a CSV file.
Exercise 1.2: Extract all tables from the page https://www.indeed.com/salaries. Convert them to data frames and save each to a separate CSV file.