COMP453 Data Science

Lab manual

College of Computer Science and Information Technology

Jazan University

Prepared by

Dr. Hani Alnami

03/11/2023

Introduction

This lab will discuss the basic content of data science tools such as data manipulation, visualization, discussion, and some machine learning tasks. Machine learning is strongly related to data science since it depends directly on data. Building a machine learning on trash data will provide a trash model, so cleaning data to improve its quality is essential.

Recently, python programming language utilized to build different machine learning model and manipulate data sets using several libraries such as Pandas And Scikit-learn. Both libraries will be utilized deeply in this lab.

Pandas is a popular open-source Python library that provides powerful data manipulation and analysis tools. It is widely used in data science, data analysis, and data engineering for working with structured data. Here's a detailed explanation of key concepts and functionalities in Pandas:

1. Data Structures:
   -DataFrame: The primary Pandas data structure is the DataFrame, which is essentially a 2-dimensional tabular data structure with labeled axes (rows and columns). It can be thought of as a spreadsheet or SQL table in memory. Columns can have different data types (e.g., integers, strings, floats) and are labeled.

   - Series: A Series is a one-dimensional labeled array that can hold various data types. It is like a single column or row in a DataFrame. Series are the building blocks of DataFrames.

2. Data Input/Output:
   - Pandas can read data from various sources such as CSV, Excel, SQL databases, JSON, and more. The `read_csv()`, `read_excel()`, `read_sql()`, and other functions are used to import data into Pandas DataFrames.
   - Similarly, Pandas can write DataFrames back to these formats using functions like `to_csv()` and `to_excel()`.

3. Data Selection and Indexing:
   - You can access specific rows and columns of a DataFrame using labels or numeric indices. The `loc` and `iloc` methods are commonly used for this purpose.
   - Boolean indexing allows you to filter rows based on conditions.

4. Data Manipulation:
   - Pandas provides various methods for data manipulation, including filtering, sorting, joining, merging, and reshaping DataFrames.
   - You can use functions like `groupby()` for aggregation operations and `pivot_table()` for reshaping data.

5. Missing Data Handling:
   - Pandas offers tools for dealing with missing data, such as the `fillna()` and `dropna()` functions.

6. Data Visualization:
   - Although not a primary visualization library, Pandas has built-in integration with Matplotlib, making it easy to create basic plots and visualizations directly from DataFrames.

7. Statistical Analysis:
   - Pandas includes functions for basic statistical analysis of data, like `mean()`, `std()`, `min()`, `max()`, and more.

8. Time Series Handling:
   - Pandas has excellent support for time series data, including date and time indexing, resampling, and time-based calculations.

9. Performance and Efficiency:
   - Pandas is designed to handle large datasets efficiently, but it's important to be mindful of memory usage and performance, especially when dealing with very large datasets. Vectorized operations are generally faster than using loops.

10. Integration with Other Libraries:
   - Pandas works well with other data science libraries like NumPy, SciPy, Scikit-Learn, and more, enabling comprehensive data analysis and machine learning workflows.

11. Community and Documentation:
   - Pandas has a vibrant community, extensive documentation, and numerous online resources, including tutorials and forums, making it easy to find help and support.

In summary, Pandas is an essential tool for data manipulation and analysis in Python. Its versatile data structures and functions make it a go-to choice for cleaning, transforming, and analyzing data in a wide range of data-related tasks.

Scikit-Learn, often referred to as sklearn, is a powerful open-source machine learning library for Python. It provides a wide array of tools for performing various machine learning tasks, including classification, regression, clustering, dimensionality reduction, and more. Here's a detailed explanation of the Scikit-Learn library:

1. Consistent API:
   - Scikit-Learn has a consistent and easy-to-use API, making it simple to implement machine learning algorithms. Its uniform interface allows you to switch between different algorithms effortlessly.

2. Supervised Learning:
   - Scikit-Learn supports a wide range of supervised learning algorithms, including:
     - Classification: Algorithms for predicting categorical labels.
     - Regression: Algorithms for predicting continuous numeric values.

3. Unsupervised Learning:
   - Scikit-Learn offers various unsupervised learning techniques, such as:

- Clustering: Algorithms for grouping similar data points together.
- Dimensionality Reduction: Techniques like Principal Component Analysis (PCA) and t-distributed Stochastic Neighbor Embedding (t-SNE) for reducing the dimensionality of data.
- Anomaly Detection: Methods for identifying outliers or anomalies in data.

4. Preprocessing and Feature Engineering:
   - Scikit-Learn provides tools for data preprocessing, including scaling, normalization, encoding categorical variables, and feature selection.

5. Model Selection and Evaluation:
   - Scikit-Learn includes functions for splitting datasets into training and testing sets, cross-validation, and hyperparameter tuning using techniques like Grid Search and Random Search.
   - Evaluation metrics for classification and regression tasks are readily available, such as accuracy, precision, recall, F1-score, Mean Squared Error (MSE), etc.

6. Ensemble Methods:
   - Scikit-Learn supports ensemble methods like Random Forests, Gradient Boosting, and AdaBoost, which combine multiple models to improve predictive accuracy.

7. Integration with NumPy and Pandas:
   - Scikit-Learn seamlessly integrates with NumPy and Pandas, allowing easy data manipulation and integration with other data science libraries.

8. Handling Imbalanced Data:
   - The library provides techniques to address imbalanced datasets, including oversampling and undersampling methods.

9. Text Data Processing:
   - Scikit-Learn offers tools for text data preprocessing, including feature extraction techniques like TF-IDF and Count Vectorization.

10. Integration with Other Libraries:
    - Scikit-Learn works well with other Python libraries such as Pandas, Matplotlib, and Seaborn for data visualization and analysis.

11. Extensive Documentation and Community Support:
    - Scikit-Learn has comprehensive documentation, tutorials, and an active community, making it easy to find resources and get help when needed.

12. Scalability and Performance:
    - While Scikit-Learn is primarily designed for small to medium-sized datasets, it can handle larger datasets efficiently. However, for big data, more specialized libraries like Apache Spark's MLlib might be more suitable.

13. Customization and Extensibility:

- Advanced users can extend Scikit-Learn's functionality by creating custom estimators and transformers.

In summary, Scikit-Learn is an essential library for machine learning practitioners and data scientists. Its simplicity, wide range of algorithms, and strong community support make it a valuable tool for building and deploying machine learning models in Python.

Google Colaboratory

Google Colaboratory, commonly known as Colab, is a cloud-based platform provided by Google that allows users to write and execute Python code in a collaborative and interactive environment. Here are some key details about Google Colaboratory:

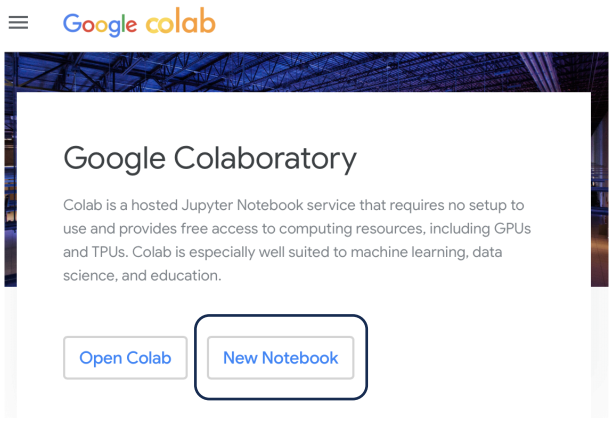
- Cloud-Based Jupyter Notebooks: Colab is built on Jupyter Notebooks, which are interactive documents that combine code, text, and visualizations. It runs entirely in the cloud, eliminating the need to install and configure software on your local machine.
- Free Access: Google Colab is free to use and provides access to a virtual machine (VM) with CPU and GPU options. The GPU option is particularly useful for machine learning and deep learning tasks, as it accelerates training times.
- Integration with Google Drive: Colab seamlessly integrates with Google Drive, making it easy to save and share notebooks. You can also access your existing notebooks stored in Google Drive.
- Preinstalled Libraries: Colab comes with a wide range of preinstalled Python libraries commonly used in data science and machine learning, such as NumPy, pandas, TensorFlow, PyTorch, and Matplotlib.
- Hardware Acceleration: As mentioned earlier, Colab offers free GPU and TPU (Tensor Processing Unit) support. This is a significant advantage for data scientists and machine learning practitioners, as it allows for faster model training.
- Sharing and Collaboration: Colab makes it easy to share your notebooks with others. You can collaborate in real-time, just like Google Docs, by sharing a link to your Colab notebook with colleagues or friends.
- Code Execution: You can execute code cells individually or run the entire notebook at once. This makes it a versatile tool for experimenting, prototyping, and developing Python code.
- Markdown Support: Colab supports Markdown cells, allowing you to add formatted text, headings, images, and hyperlinks to your notebooks for documentation and explanation.
- Extensions and Customization: Colab offers extensions and add-ons that enhance its functionality. You can also install additional Python packages if they are not preinstalled.
- Limited Resources: While Colab provides free GPU and TPU access, there are limitations on usage. Users may experience resource limitations if they use excessive computing power or storage, and sessions may be disconnected after a period of inactivity.
- Data Access: You can easily access and manipulate data stored on Google Drive or by importing datasets from the web directly into your Colab notebook.
- Export Options: Colab allows you to export your notebooks in various formats, including PDF, HTML, and plain Jupyter Notebook format.

Overall, Google Colaboratory is a powerful tool for data analysis, machine learning, and collaborative coding that leverages the convenience of the cloud and Google's infrastructure. It's particularly popular in the data science and machine learning communities for its accessibility and GPU/TPU support.

How to access google colab? You only need a gmail account to access google colab. Then follow the below steps:

1. Click on this link: https://colab.google

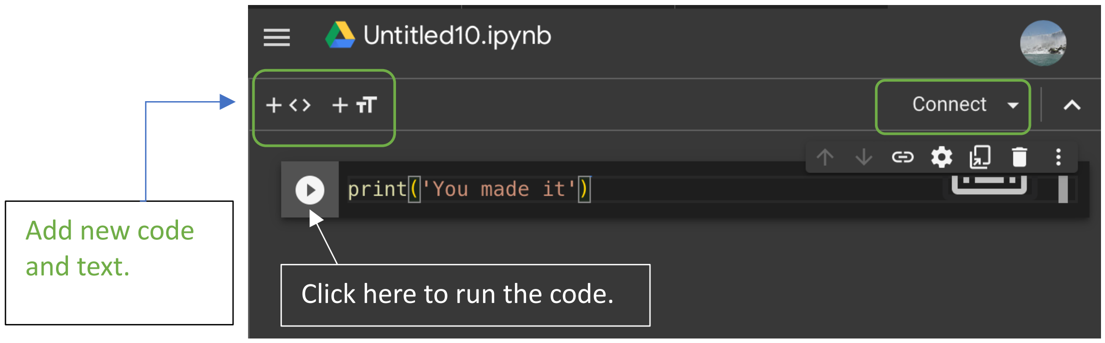
2. Click on open new Notebook. You may need to enter your username and password information for your gmail account.



Google colab

Google Colaboratory

Colab is a hosted Jupyter Notebook service that requires no setup to use and provides free access to computing resources, including GPUs and TPUs. Colab is especially well suited to machine learning, data science, and education.



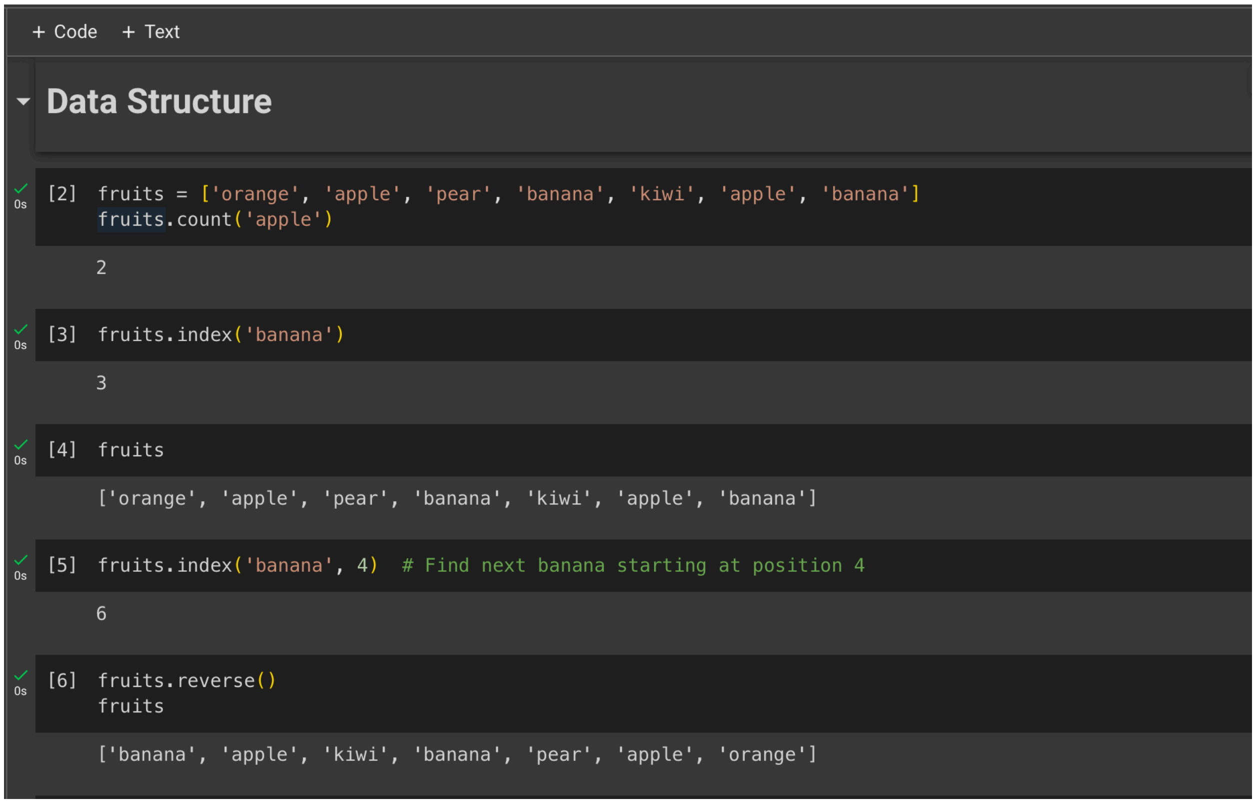
```
print('You made it')
```

3. One you access you must be connected by clicking on ‘connect’.
4. You are ready for coding.
5. Repeat the process every time you open the colab.

List of topics to be covered in this lab.

Topics	Description
1	Introduce Data Structure basic operations, utilize panda's functions, and numpy.
2	Apply different visualization techniques on IRIS datasets using Matplotlib and seaborn.
3	Data cleaning and transformation. Apply all the essential data preparation methods on different datasets. Methods such as (handling missing value, check and drop duplicated values, fix class imbalance, and detect and drop outliers.)
4	Build machine learning classification models and analyze their results. Models such as Random Forest, KNN, Naïve Bayes, decision tree, and support vector machine.
5	Build machine learning regression models and analyze their results. Models such as random forest regression, linear regression, support vector regression, and logistic regression.
6	Implement evaluation metrics for classification and regression machine learning model and analyze the outcomes.
7	Utilize NLTK function to mining and analyze text where steaming, tokenization, lemmatization, stop words, and part of speech can be introduced.
8	Implement apriori algorithm using store dataset.

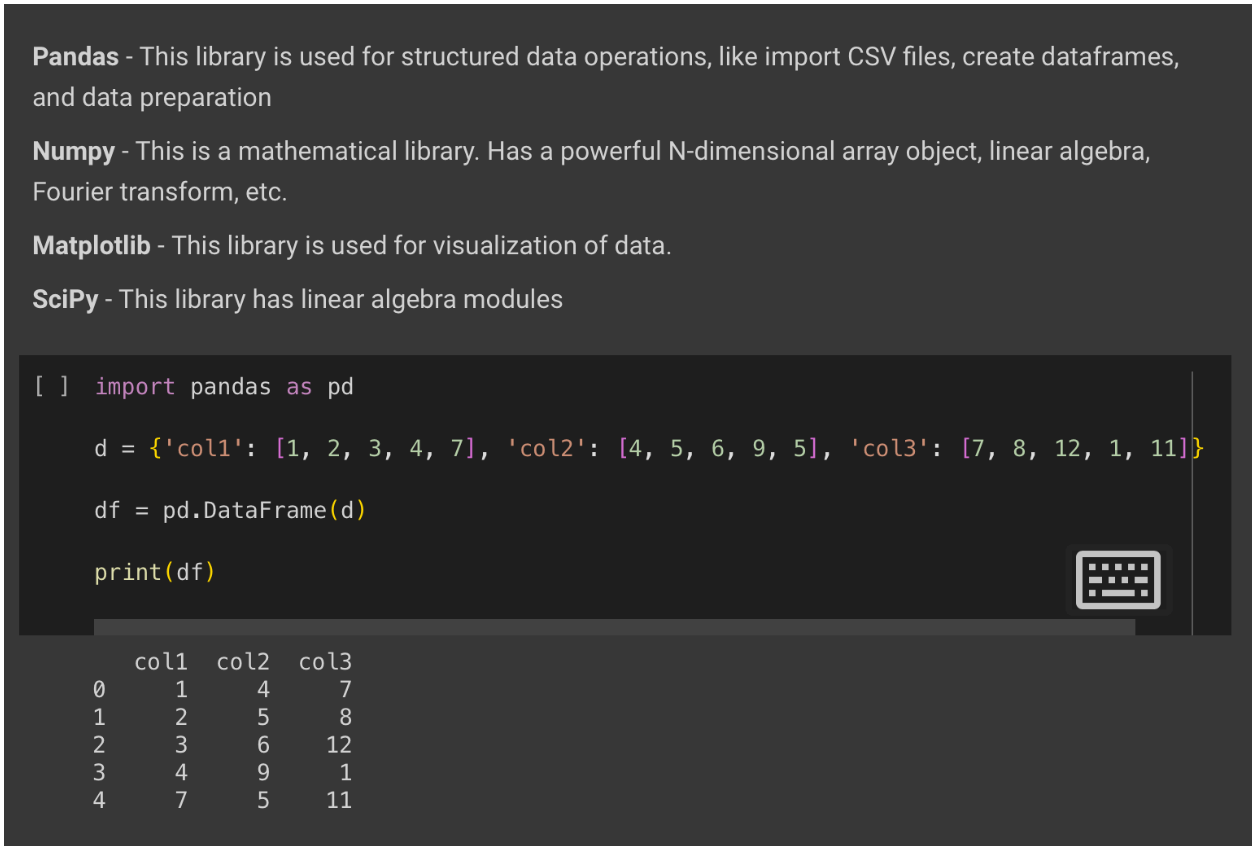
Topic 1: Introduce Data Structure



```
[2] fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
    fruits.count('apple')

    2

[3] fruits.index('banana')

    3

[4] fruits

    ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']

[5] fruits.index('banana', 4)  # Find next banana starting at position 4

    6

[6] fruits.reverse()
    fruits

    ['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange']
```

Data.count('apple ') is used to tell how munch time the word apple repeated in the array.
Data.index() used to determine the position of an individual in an array.
Data.reverse() reverse the order of the array.
Data.append('grape') used to add new item in the array.
Data.sort() sort the array elements in an alphabetical order

Pandas is a popular Python library for data manipulation and analysis. It provides data structures like DataFrame and Series, along with functions to read, write, and perform various operations on tabular data. You can use Pandas for tasks such as data cleaning, exploration, filtering, aggregation, and more. Here's a simple example of how to import Pandas and create a DataFrame.



Pandas - This library is used for structured data operations, like import CSV files, create dataframes, and data preparation

Numpy - This is a mathematical library. Has a powerful N-dimensional array object, linear algebra, Fourier transform, etc.

Matplotlib - This library is used for visualization of data.

SciPy - This library has linear algebra modules

```
[ ]  import pandas as pd

     d = {'col1': [1, 2, 3, 4, 7], 'col2': [4, 5, 6, 9, 5], 'col3': [7, 8, 12, 1, 11]}

     df = pd.DataFrame(d)

     print(df)
```

```
   col1  col2  col3
0     1     4     7
1     2     5     8
2     3     6    12
3     4     9     1
4     7     5    11
```

Apply the following commands to analyze this simple data.

```
count_column = df.shape[1] #Count the number of columns
print(count_column)
```

```
count_rows = df.shape[0] #Count the number of rows
print(count_rows)
```

```
import numpy as np
np.max(df)
np.min(df)
np.mean(df)
df.drop('col3', axis = 'columns') #delete a columns
df.drop(df.index[4]) # delete a row
df.at[0,'col1'] #select value in dataFrame
df.at[0,'col2']
df.loc[:,'col2']
df.set_index('col1') # set columns 1 as index
df.loc[5] = [10,20,30] #add a new row
df.loc[:, 'col4'] = pd.Series(['5', '6', '7','8','9','10'], index=df.index)
```

```
newcolumns = {'col1' : 'colA'} # rename columns
df.rename(columns= newcolumns, inplace=True)
df=df.rename(index={1: 'a'}) # rename rows
df.rename(index={2: 'a'}) # rename rows
```

NumPy is a fundamental Python library for numerical and mathematical operations. It provides support for large, multi-dimensional arrays and matrices, as well as a collection of mathematical functions to operate on these arrays efficiently. NumPy is the foundation for many scientific and data science libraries in Python.

Here's a simple example of how to use NumPy:

```
import numpy as np

# Create a NumPy array
arr = np.array([1, 2, 3, 4, 5])

# Perform operations on the array
mean = np.mean(arr)
sum = np.sum(arr)

print("Array:", arr)
print("Mean:", mean)
print("Sum:", sum)
```

In this example, we import NumPy as `np`, create a NumPy array, and then calculate the mean and sum of the elements in the array using NumPy functions.

NumPy is particularly useful for tasks involving numerical data, scientific computing, and linear algebra operations. It's a fundamental tool in the Python data science ecosystem, often used in conjunction with libraries like Pandas and Matplotlib for data analysis and visualization. If you have specific questions about NumPy or need more guidance, please feel free to ask! NumPy is a fundamental Python library for numerical and mathematical operations. It provides support for large, multi-dimensional arrays and matrices, as well as a collection of mathematical functions to operate on these arrays efficiently. NumPy is the foundation for many scientific and data science libraries in Python.

Here's a simple example of how to use NumPy:

```
import numpy as np

# Create a NumPy array
arr = np.array([1, 2, 3, 4, 5])

# Perform operations on the array
mean = np.mean(arr)
sum = np.sum(arr)

print("Array:", arr)
print("Mean:", mean)
print("Sum:", sum)
```

In this example, we import NumPy as `np`, create a NumPy array, and then calculate the mean and sum of the elements in the array using NumPy functions.

NumPy is particularly useful for tasks involving numerical data, scientific computing, and linear algebra operations. It's a fundamental tool in the Python data science ecosystem, often used in conjunction with libraries like Pandas and Matplotlib for data analysis and visualization. If you have specific questions about NumPy or need more guidance, please feel free to ask!

Topic 2: Visualization Techniques

Matplotlib is a popular Python library for creating static, animated, and interactive visualizations in data science and scientific computing. It provides a wide range of tools for creating high-quality plots and charts. Here's a detailed explanation of key aspects of Matplotlib:

1. Installation:
   You can install Matplotlib using pip:

```
pip install matplotlib
```

2. Importing:
   To use Matplotlib in your Python code, you typically start by importing it:

```
import matplotlib.pyplot as plt
```

3. Figure and Axes:
   Matplotlib plots are organized into two main components: the `Figure` and `Axes`. The `Figure` is the container that holds all elements of the plot, and `Axes` represent the actual plotting area or individual subplots within the figure.

4. Basic Plotting:
   You can create simple line plots, scatter plots, bar plots, and more using `plt.plot()`, `plt.scatter()`, `plt.bar()`, and related functions. For example:

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
y = [10, 12, 5, 8, 9]

plt.plot(x, y)
plt.xlabel('X-axis Label')
plt.ylabel('Y-axis Label')
plt.title('Simple Line Plot')
plt.show()
```

5. Customizing Plots:
   You can customize various aspects of the plot, such as labels, titles, colors, line styles, and markers. Matplotlib provides a wide range of options to control the appearance of your plots.

6. Subplots:

You can create multiple plots within the same figure using `plt.subplots()`. This allows you to arrange and display multiple plots in a grid.

7. Annotations and Text:
   Matplotlib provides functions like `plt.text()` and `plt.annotate()` to add text and annotations to your plots.

8. Saving Plots:
   You can save your plots as image files (e.g., PNG, JPEG, PDF) using `plt.savefig()`.

9. Common Plot Types:
   - Line Plots: `plt.plot()`
   - Scatter Plots: `plt.scatter()`
   - Bar Plots: `plt.bar()`, `plt.barh()`
   - Histograms: `plt.hist()`
   - Pie Charts: `plt.pie()`
   - Box Plots: `plt.boxplot()`
   - Heatmaps: `plt.imshow()`
   - And many more...

10. Advanced Features:
    Matplotlib also supports advanced features such as 3D plotting, polar plots, and contour plots.

11. Matplotlib Styles:
    You can apply different styles to your plots using `plt.style.use()`. There are various pre-defined styles available, or you can create your own custom styles.

12. Interactive Plots:
    Matplotlib can be integrated with interactive backends like Jupyter Notebook or used with libraries like `mpld3` for creating interactive visualizations.

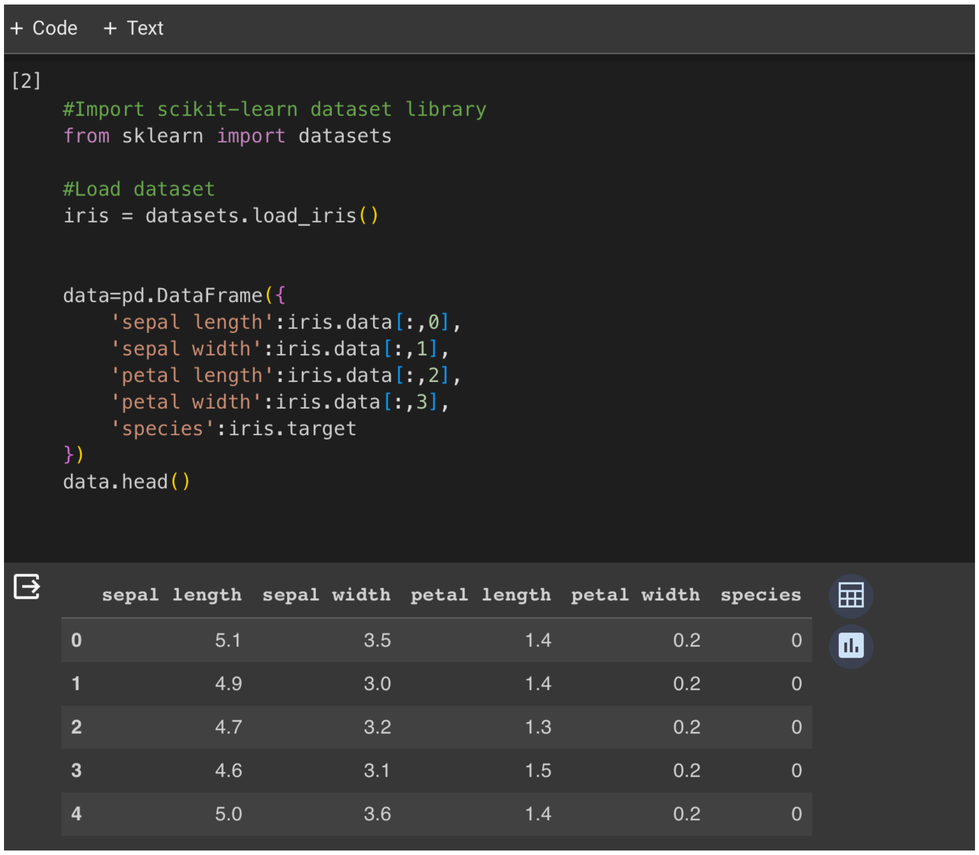
13. Matplotlib and Pandas:
    Matplotlib integrates well with the Pandas library, allowing you to easily create plots from Pandas DataFrames.

14. Documentation:
    The official Matplotlib documentation (https://matplotlib.org/stable/contents.html) is an excellent resource for detailed information on all aspects of the library.

Matplotlib is a versatile library, and while this overview covers the basics, there is much more to explore based on your specific plotting needs.

Here are some examples of using matplotlib on IRIS Dataset:

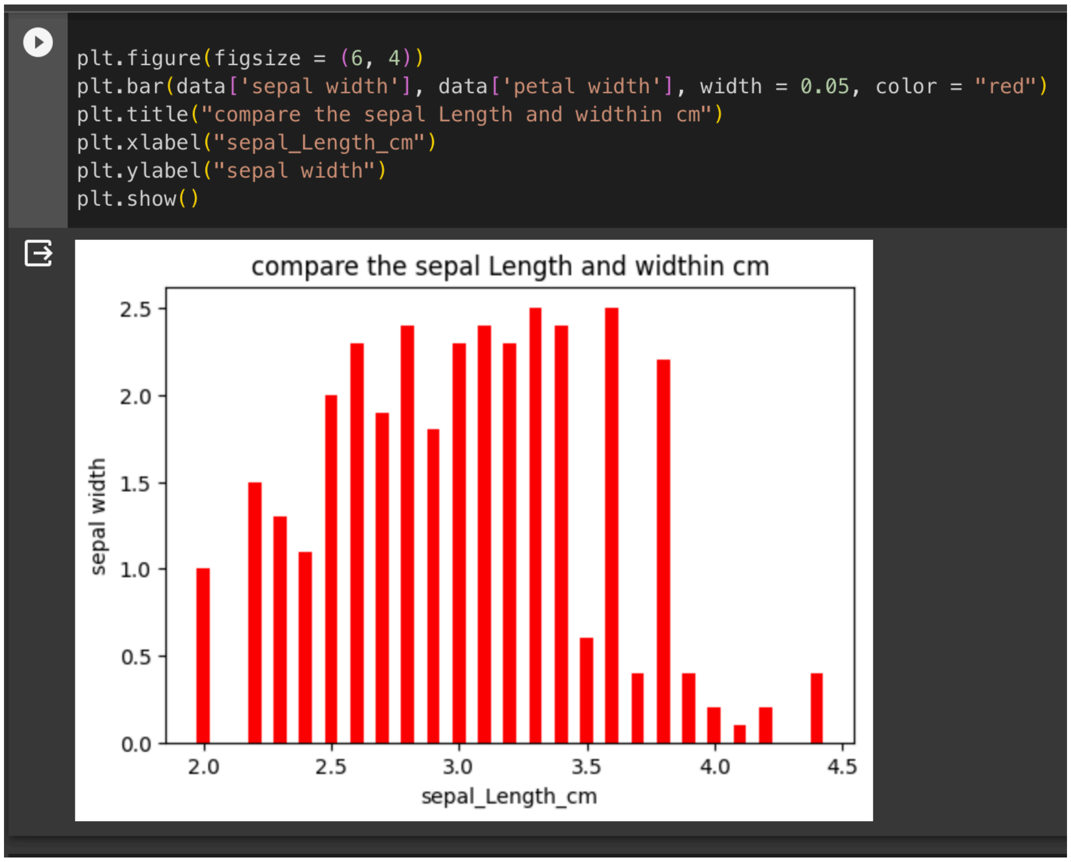
First load IRIS data as following:



```
#Import scikit-learn dataset library
from sklearn import datasets

#Load dataset
iris = datasets.load_iris()


data=pd.DataFrame({
    'sepal length':iris.data[:,0],
    'sepal width':iris.data[:,1],
    'petal length':iris.data[:,2],
    'petal width':iris.data[:,3],
    'species':iris.target
})
data.head()
```

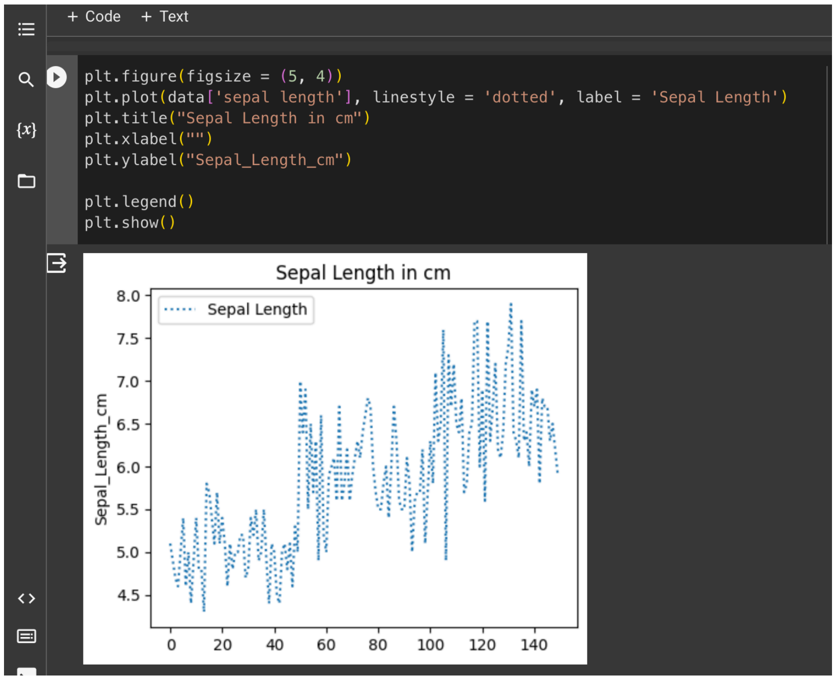
	sepal length	sepal width	petal length	petal width	species
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0

After importing matplotlib we can start implementing some visualizations such as bar char:



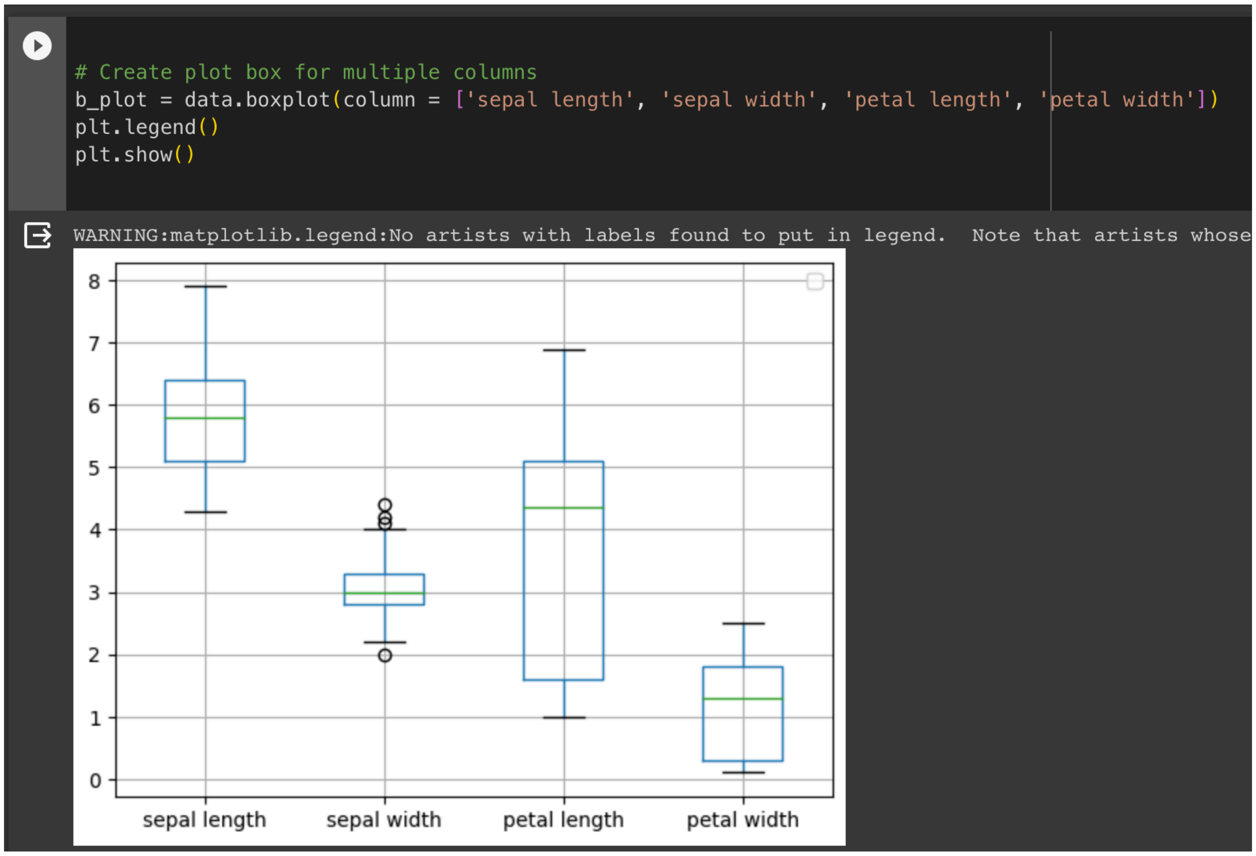
```
plt.figure(figsize = (6, 4))
plt.bar(data['sepal width'], data['petal width'], width = 0.05, color = "red")
plt.title("compare the sepal Length and widthin cm")
plt.xlabel("sepal_Length_cm")
plt.ylabel("sepal width")
plt.show()
```

Line chart:



```
plt.figure(figsize = (5, 4))
plt.plot(data['sepal length'], linestyle = 'dotted', label = 'Sepal Length')
plt.title("Sepal Length in cm")
plt.xlabel("")
plt.ylabel("Sepal_Length_cm")

plt.legend()
plt.show()
```

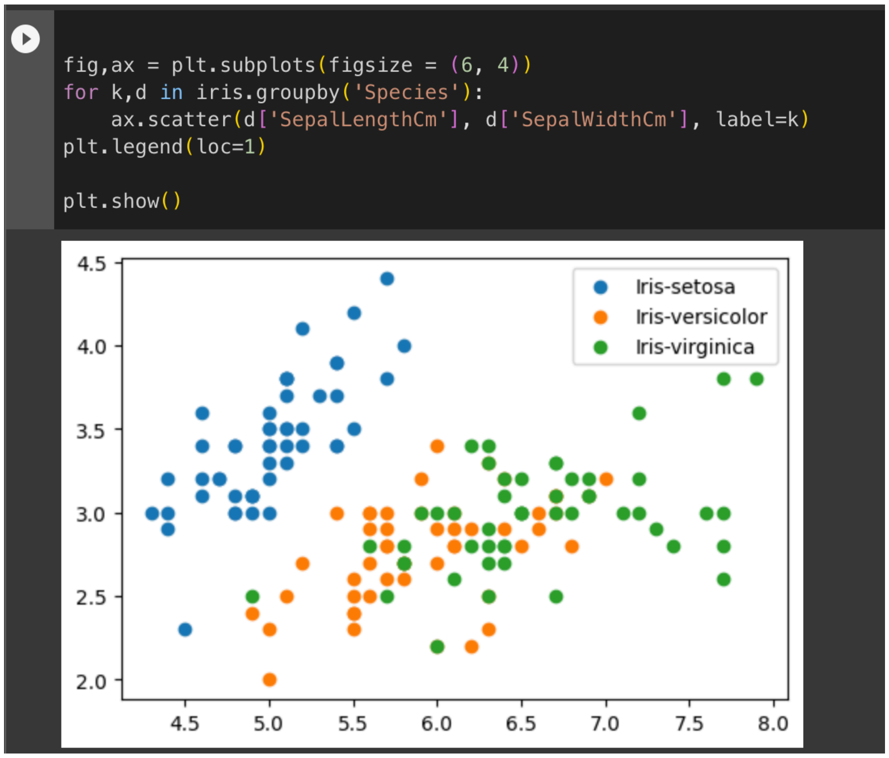
Box plot:



```
# Create plot box for multiple columns
b_plot = data.boxplot(column = ['sepal length', 'sepal width', 'petal length', 'petal width'])
plt.legend()
plt.show()
```

```
WARNING:matplotlib.legend:No artists with labels found to put in legend.  Note that artists whose
```

Scatter:



```
fig,ax = plt.subplots(figsize = (6, 4))
for k,d in iris.groupby('Species'):
    ax.scatter(d['SepalLengthCm'], d['SepalWidthCm'], label=k)
plt.legend(loc=1)

plt.show()
```

Seaborn is a popular Python data visualization library built on top of Matplotlib. It is specifically designed for creating aesthetically pleasing and informative statistical graphics. Seaborn simplifies the process of creating complex visualizations by providing high-level functions and a high-level interface to manipulate data. Here are the key features and details of Seaborn:

1. Statistical Data Visualization: Seaborn is primarily used for visualizing statistical relationships in data. It can create various types of plots like scatter plots, bar plots, histograms, box plots, and more, which are often used for exploring and communicating patterns in data.

2. Integration with Pandas: Seaborn works seamlessly with Pandas DataFrames, making it easy to plot data directly from your datasets. This integration simplifies data preprocessing and manipulation.

3. Attractive Aesthetics: Seaborn comes with built-in themes and color palettes that improve the aesthetics of your plots. It also allows you to customize the appearance of your plots easily.

4. Statistical Estimation: Seaborn offers functions for plotting regression models and estimating relationships between variables using methods like linear regression, logistic regression, and more.

5. Categorical Plots: Seaborn provides functions for visualizing categorical data, such as bar plots, count plots, and box plots, making it easy to explore and compare categories in your data.

6. Distribution Plots: You can create distribution plots like histograms and kernel density plots to understand the data's distribution and underlying patterns.

7. Matrix Plots: Seaborn supports matrix plots, including heatmaps and cluster maps, which are helpful for visualizing relationships and correlations in large datasets.

8. Time Series Visualization: It offers tools for visualizing time series data, including line plots with date or time on the x-axis.

9. Grids and Subplots: Seaborn allows you to create grids of plots, making it convenient to compare multiple visualizations side by side.

10. Seaborn vs. Matplotlib: While Seaborn is built on Matplotlib, it abstracts away some of Matplotlib's complexity and provides a higher-level API for common statistical plots. This makes Seaborn an excellent choice for data analysts and scientists who want to create attractive and informative visualizations with less code.

To use Seaborn, you typically follow these steps:
1. Import the Seaborn library.
2. Load your dataset into a Pandas DataFrame.
3. Use Seaborn's functions to create visualizations, customize them as needed, and display the plots.

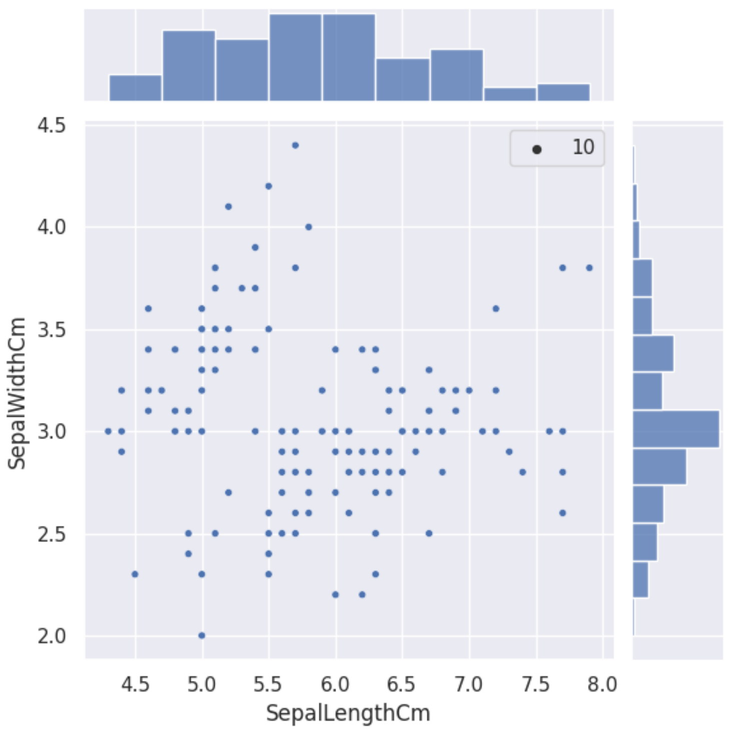
Here's a simple example of creating a scatter plot using Seaborn:

```
import seaborn as sns
import pandas as pd

# Load a sample dataset
data = pd.read_csv('your_dataset.csv')

# Create a scatter plot
sns.scatterplot(x='x_column', y='y_column', data=data)

# Show the plot
import matplotlib.pyplot as plt
plt.show()
```

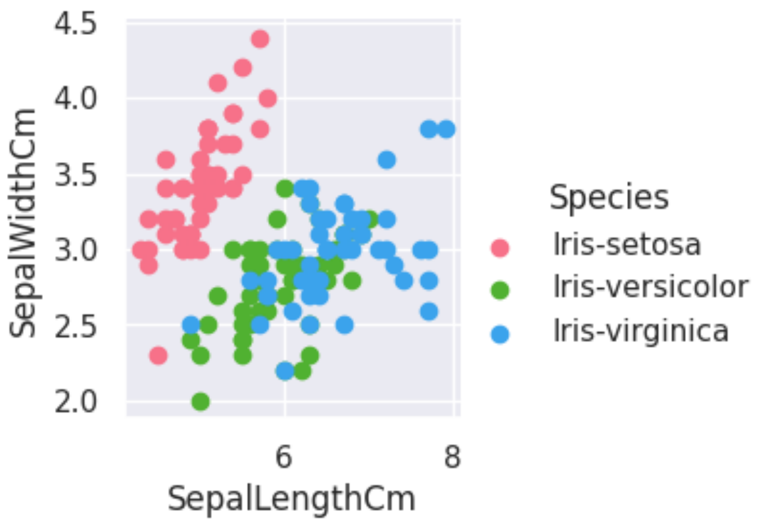
Seaborn's rich functionality and ease of use make it a valuable tool for data visualization in Python, especially for exploring and presenting data in a visually appealing way.

Some example of using seaborn on IRIS dataset:

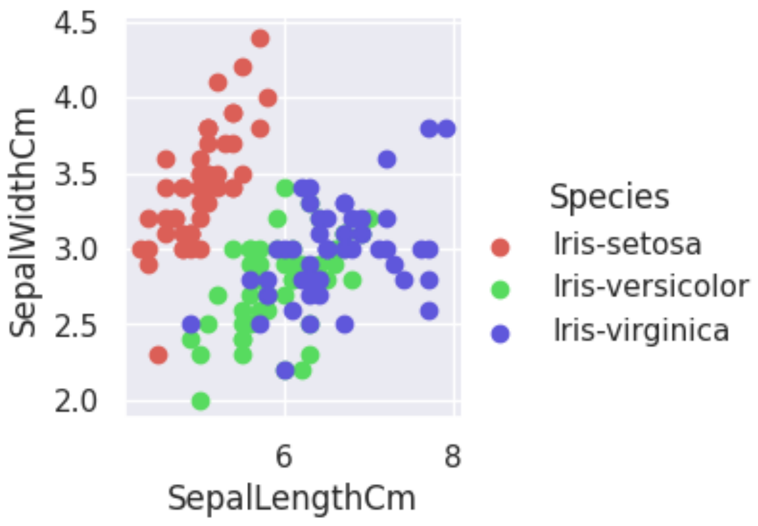
```
import seaborn as sns
sns.set(color_codes=True)
import warnings
warnings.filterwarnings("ignore")
sns.jointplot(x="SepalLengthCm", y="SepalWidthCm", data=iris, size=10) #The additional detail provided by the histograms shows us there are many data points with a sepal width of 3 spread across the sepal length axis.
```



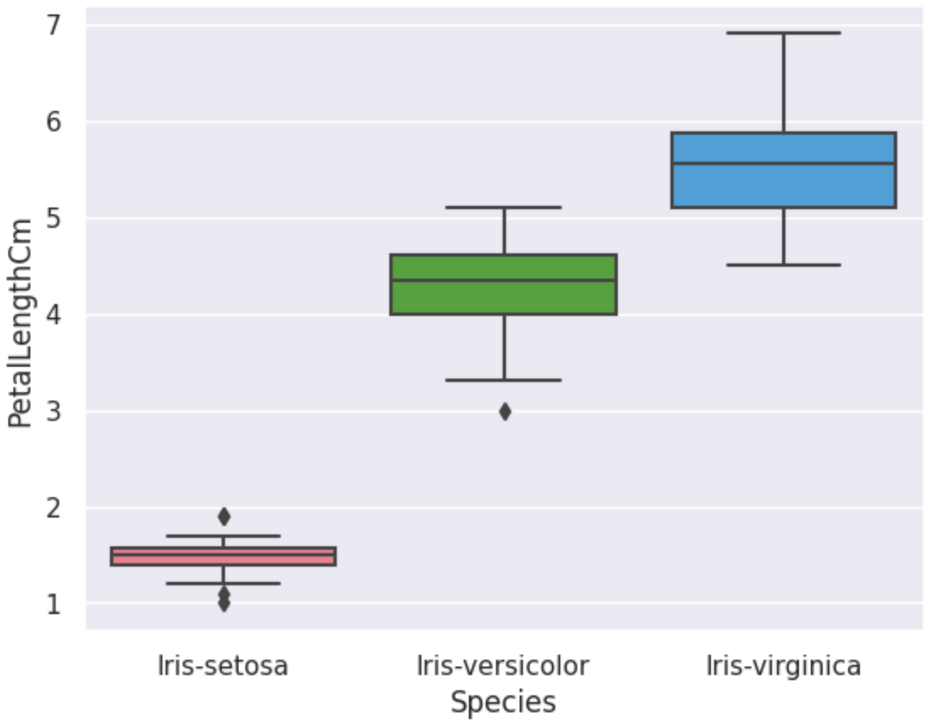
```
sns.FacetGrid(iris, hue="Species", palette="husl") \
  .map(plt.scatter, "SepalLengthCm", "SepalWidthCm") \
  .add_legend()
```



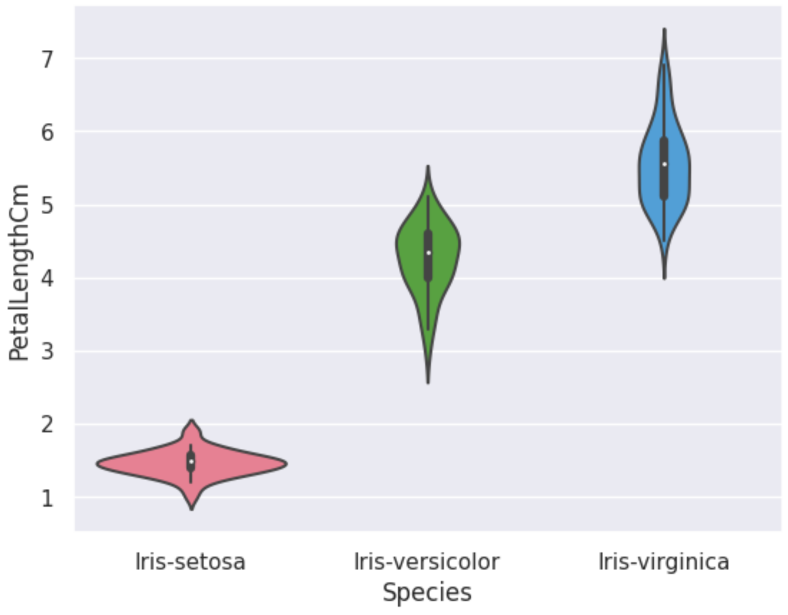
```
sns.FacetGrid(iris, hue="Species", palette="hls") \
  .map(plt.scatter, "SepalLengthCm", "SepalWidthCm") \
  .add_legend()
```



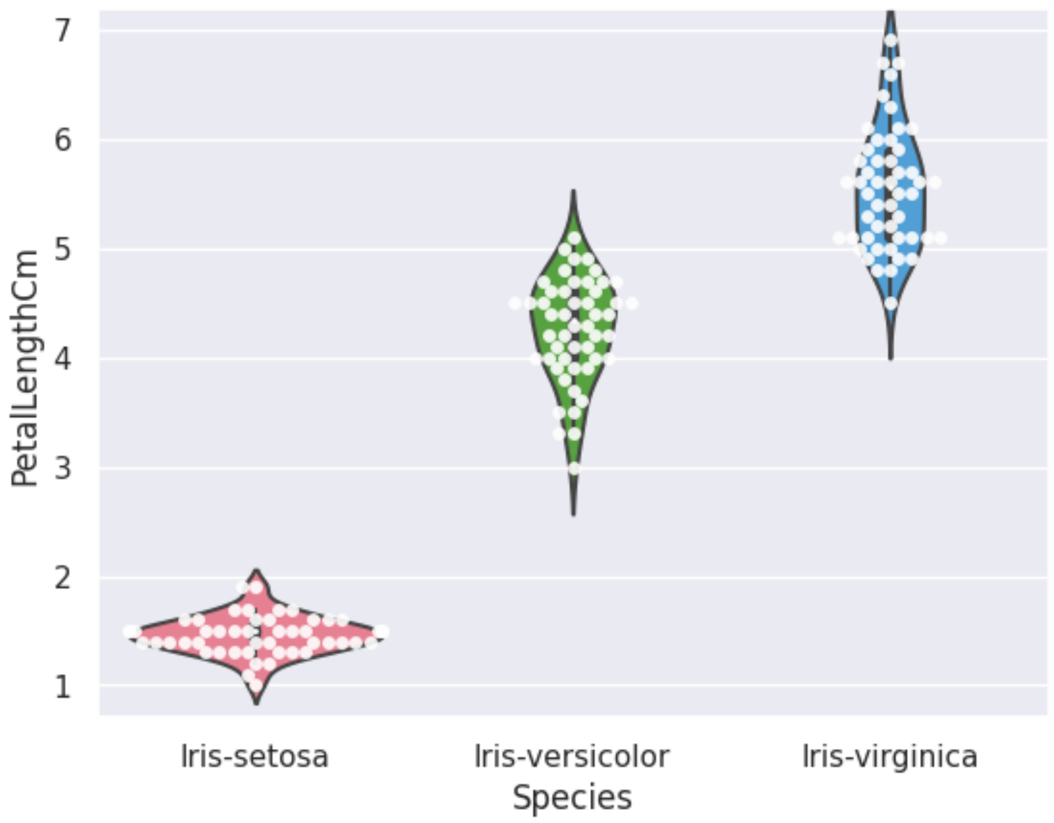
```
sns.boxplot(x="Species", y="PetalLengthCm", palette="husl", data=iris)
```



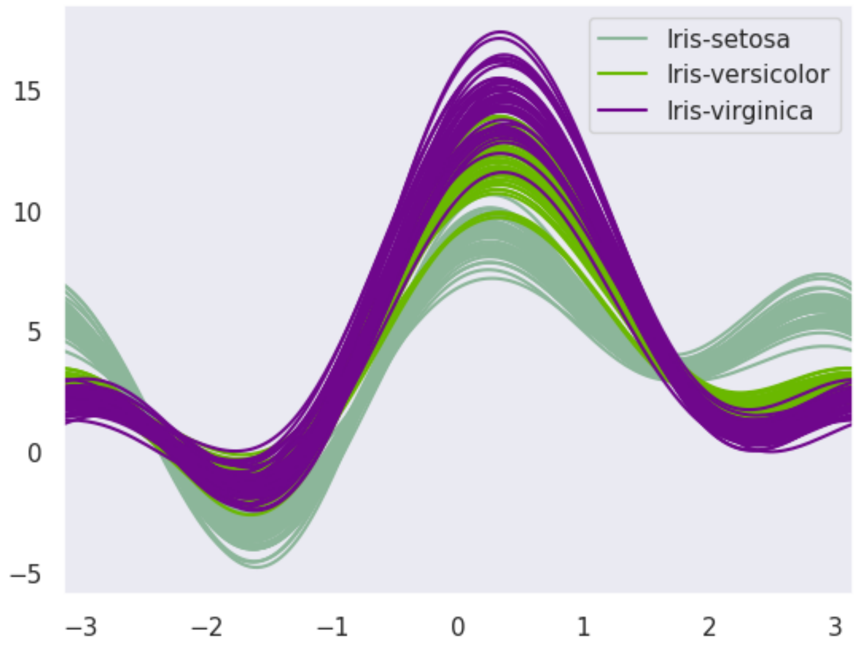
```
sns.violinplot(x="Species", y="PetalLengthCm", palette="husl", data=iris)
```



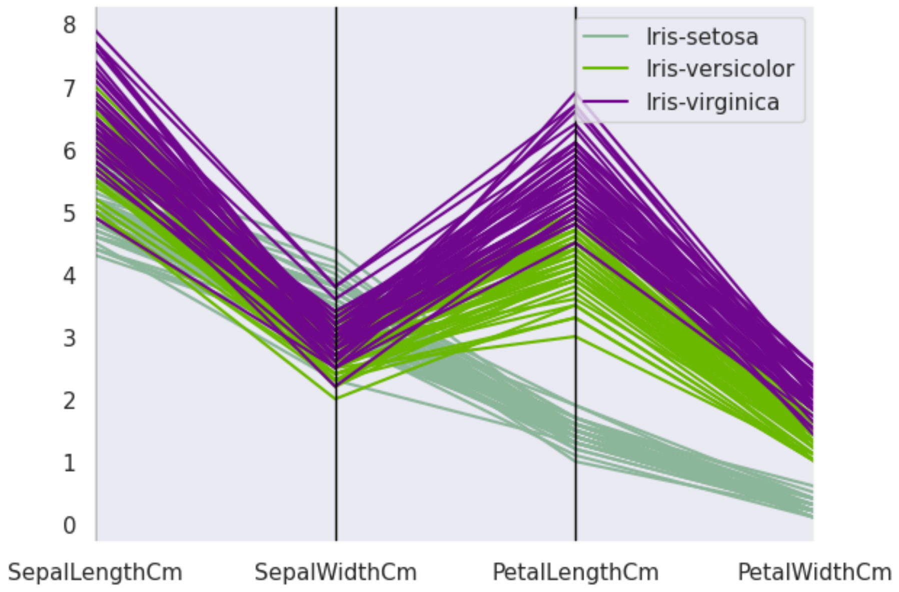
```
sns.violinplot(x="Species", y="PetalLengthCm", palette="husl", data=iris)
sns.swarmplot(x="Species", y="PetalLengthCm", data=iris, color="w", alpha=.9)
```



```
# Andrews Curves can also be created using pandas
# Andrews Curves involve using attributes of samples as coefficients for Fourier series
# and then plotting the series
from pandas.plotting import andrews_curves
andrews_curves(iris.drop("Id", axis=1), "Species")
```



```
# Another multivariate visualization technique pandas has is parallel_coordinates
# Parallel coordinates plots each feature on a separate column & then draws lines
# connecting the features for each data sample
from pandas.plotting import parallel_coordinates
parallel_coordinates(iris.drop("Id", axis=1), "Species")
```



Topic 3: Data Cleaning

Data cleaning is a crucial step in data analysis. It involves identifying and correcting errors, inconsistencies, and missing values in a dataset to ensure that the data is accurate and reliable for analysis. Common data cleaning tasks include:

1. Handling missing data: Addressing missing values by either imputing them with appropriate values or removing rows/columns with too many missing values.

2. Removing duplicates: Identifying and removing duplicate records to avoid skewing analysis results.

3. Standardizing data: Ensuring consistent formats, units, and scales for variables, such as dates and numerical values.

4. Handling outliers: Identifying and addressing outliers that can distort statistical analyses.

5. Correcting data types: Ensuring that variables have the correct data types (e.g., numerical, categorical) for analysis.

6. Dealing with typos and inconsistencies: Identifying and correcting data entry errors, typos, and inconsistencies in naming conventions.

7. Data transformation: Performing necessary transformations like normalization or scaling for variables as needed.

Data cleaning helps analysts and data scientists obtain more accurate insights and make informed decisions based on reliable data.

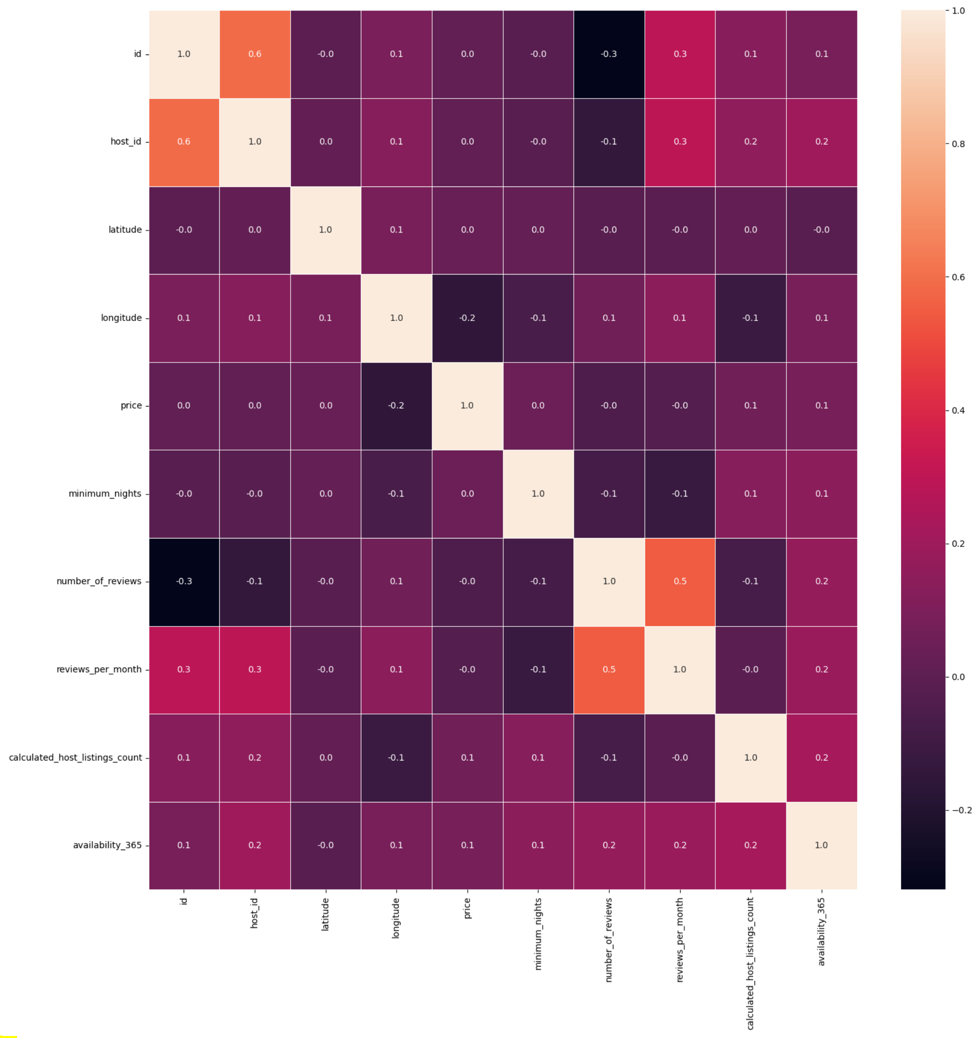
In this lab we are going to demonstrate basic data cleaning methods such as feature selection using correlation, handling missing value, check duplicated data and etc.

Correlation:

```
import time
import matplotlib.pyplot as plt
from matplotlib.pyplot import figure
import seaborn as sns
```

```
#Correlation map. Find correlation between features
f,ax = plt.subplots(figsize=(18, 18))
sns.heatmap(df.corr(), annot=True, linewidths=.5, fmt= '.1f',ax=ax)
```

In Python's Pandas library, the corr() function is used to compute the correlation between columns (variables) in a DataFrame. Correlation measures the statistical association between two variables and is commonly used to understand relationships in data.



Info()

In Python's Pandas library, the `info()` method is used to obtain concise information about a DataFrame, including the data types of each column, the number of non-null values, and memory usage. Here's how to use the `info()` method:

import pandas as pd

```
# Create a sample DataFrame
data = {'Name': ['Alice', 'Bob', 'Charlie', 'David'],
        'Age': [25, 30, 22, 35],
        'City': ['New York', 'San Francisco', 'Los Angeles', 'Chicago']}

df = pd.DataFrame(data)

# Get information about the DataFrame
df.info()
```

When you run `df.info()`, it will provide you with a summary of the DataFrame's structure, including:

- The number of rows (entries) and columns (attributes).
- The data type of each column.
- The count of non-null values in each column.
- The total memory usage of the DataFrame.

This method is useful for quickly assessing the completeness of your data and understanding the data types, especially when dealing with large datasets.

Duplicated data:

In Python's Pandas library, the `duplicated()` method is used to identify duplicate rows in a DataFrame. It returns a Boolean Series where each row is marked as `True` if it is a duplicate of a previous row and `False` if it is not a duplicate. Here's how to use the `duplicated()` method:

```
import pandas as pd

# Create a sample DataFrame with duplicates
data = {'Name': ['Alice', 'Bob', 'Alice', 'Charlie', 'David', 'Alice'],
        'Age': [25, 30, 25, 22, 35, 25]}

df = pd.DataFrame(data)
```

```
# Check for duplicate rows
duplicates = df.duplicated()

# Show the DataFrame with the 'duplicates' column
df['IsDuplicate'] = duplicates
print(df)
```

In this example, the `duplicates` variable will be a Boolean Series indicating which rows are duplicates based on the entire row's contents. You can see that rows 0, 2, and 5 are marked as duplicates because they have the same values in both the 'Name' and 'Age' columns.

You can also remove duplicate rows from a DataFrame using the `drop_duplicates()` method if needed:

```
# Remove duplicate rows
df_no_duplicates = df.drop_duplicates()
print(df_no_duplicates)
```

This will create a new DataFrame (`df_no_duplicates`) that excludes the duplicate rows.

Missing Value:
In Python's Pandas library, the `isna()` function (or its alias `isnull()`) is used to identify missing or NaN (Not a Number) values in a DataFrame. It returns a DataFrame of the same shape as the original, with `True` where a value is missing and `False` where the value is present. Here's how to use the `isna()` function:

```
import pandas as pd
import numpy as np

# Create a sample DataFrame with missing values
data = {'Name': ['Alice', 'Bob', 'Charlie', np.nan, 'David'],
        'Age': [25, np.nan, 22, 35, 30]}

df = pd.DataFrame(data)

# Check for missing values
missing_values = df.isna()

# Show the DataFrame with missing value indicators
print(missing_values)
```

In this example, `missing_values` will be a DataFrame of the same shape as `df`, where `True` represents a missing value, and `False` represents a non-missing value. You can see that 'Name' has a missing value in row 3, and 'Age' has a missing value in row 1.

You can also count the number of missing values in each column using the `sum()` function:

```
# Count missing values in each column
missing_count = df.isna().sum()
print(missing_count)
```

This will provide you with a Series that shows the count of missing values in each column of the DataFrame.

Data.dropna() function id used to delete all the missing value.

Outlier:

In data analysis, an outlier is an observation or data point that significantly differs from the other observations in a dataset. Outliers can skew statistical analyses and should be carefully identified and handled. Here's how you can identify and handle outliers in a DataFrame using Python's Pandas library and visualization tools:

1. Visualize the data: Start by visualizing your data using scatter plots, box plots, or histograms. These visualizations can help you identify potential outliers.

2. Descriptive statistics: Calculate summary statistics such as mean, median, standard deviation, and quartiles for your numerical columns to understand the distribution of the data.

3. Box plots: Box plots are effective for visualizing outliers. You can use the `boxplot()` function in Pandas or libraries like Seaborn and Matplotlib to create box plots for your data. Outliers are typically represented as individual points beyond the "whiskers" of the box plot.

```
  import pandas as pd
  import matplotlib.pyplot as plt

  # Create a DataFrame
  data = {'Values': [10, 15, 20, 22, 23, 200]}

  df = pd.DataFrame(data)

  # Create a box plot
```

```
   df.boxplot(column='Values')
   plt.show()
```

4. IQR Method: One common method for identifying outliers is the Interquartile Range (IQR) method. You calculate the IQR by finding the difference between the third quartile (Q3) and the first quartile (Q1). Outliers are typically values that fall below Q1 - 1.5 * IQR or above Q3 + 1.5 * IQR.

```
   Q1 = df['Values'].quantile(0.25)
   Q3 = df['Values'].quantile(0.75)
   IQR = Q3 - Q1

   lower_bound = Q1 - 1.5 * IQR
   upper_bound = Q3 + 1.5 * IQR

   outliers = df[(df['Values'] < lower_bound) | (df['Values'] > upper_bound)]
```

5. Handle outliers: Depending on your analysis and the nature of the data, you can choose to handle outliers in various ways, including:

- Removing outliers: You can choose to remove the rows containing outliers from your DataFrame.
- Transforming data: Apply data transformations like log transformations to reduce the impact of outliers.
- Winsorizing: Replace outliers with a specified percentile value (e.g., replace values above the 99th percentile with the 99th percentile value).
- Use robust statistical methods: Some statistical methods are less sensitive to outliers, such as the median instead of the mean.

The approach to handling outliers should be based on your specific analysis goals and domain knowledge. It's essential to carefully consider the impact of outliers on your results before deciding on an appropriate strategy.

Class imbalance is a common issue in machine learning when one or more classes in a classification problem have significantly fewer examples than others. Dealing with class imbalance is crucial because it can lead to biased models that perform poorly on the minority class. Here are some strategies to address class imbalance in a Python DataFrame:

Resampling:

- Oversampling: Increase the number of instances in the minority class by duplicating or generating synthetic samples. You can use techniques like SMOTE (Synthetic Minority Over-sampling Technique) to create synthetic samples.

- Undersampling: Reduce the number of instances in the majority class by randomly removing samples. Be cautious with this approach, as you might lose important information.

```
Undersampling implementation (NearMiss Algorithm):
features = ['WBN_LP_H_1.00',  'XLogP',  'PSA',  'NumRot', 'NumHBA', 'NumHBD', 'MW',
'BBB' ,'BadGroup']
X = df.loc[:, features]
y = df.loc[:, ['Outcome']]

# Split the data to train and test before fixing class imbalance
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0, train_size = .75)
```

```
#Near Miss Undersampling Version 1
from imblearn.under_sampling import NearMiss
from collections import Counter
undersampling = NearMiss(version=3, n_neighbors=3)

X_undersampling, Y_undersampling = undersampling.fit_resample(X_train, y_train)

Y_undersampling['Outcome'].value_counts()
```

Implementation of the oversampling:

```
from imblearn.over_sampling import BorderlineSMOTE

oversampling = BorderlineSMOTE()

X_res, Y_res = oversampling.fit_resample(X_train, y_train)

Y_res['Outcome'].value_counts()
```

Topic 4: Classification

Classification in machine learning is a type of supervised learning task where the goal is to categorize input data into predefined classes or categories. It's used to make predictions or decisions based on input features. Common algorithms for classification include logistic regression, decision trees, random forests, support vector machines, and neural networks. It's widely used in applications like spam detection, image recognition, and medical diagnosis.

To perform a Random Forest classification using Scikit-learn in Python, follow these steps:

1. Import Necessary Libraries:
   Start by importing the required libraries, including Scikit-learn (usually imported as `sklearn`) and the RandomForestClassifier class.

```
   from sklearn.ensemble import RandomForestClassifier
   from sklearn.model_selection import train_test_split
   from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
```

2. Load and Prepare the Data:
   Load your dataset, preprocess it if needed (handle missing values, encode categorical features, etc.), and split it into training and testing sets.

```
   import pandas as pd

   # Load your data (e.g., from a CSV file)
   data = pd.read_csv('your_dataset.csv')

   # Separate features and target variable
   X = data.drop('target_column', axis=1)
   y = data['target_column']

   # Split the data into training and testing sets
   X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

3. Create and Train the Random Forest Classifier:
   Initialize the Random Forest classifier and train it using the training data.

```
   # Initialize the Random Forest Classifier
   rf_classifier = RandomForestClassifier(n_estimators=100, random_state=42)
```

```
# Train the classifier
rf_classifier.fit(X_train, y_train)
```

Adjust the hyperparameters as needed. `n_estimators` is the number of decision trees in the forest, and you can fine-tune other hyperparameters for better performance.

4. Make Predictions:
   Use the trained Random Forest classifier to make predictions on the test data.

```
y_pred = rf_classifier.predict(X_test)
```

5. Evaluate the Model:
   Assess the performance of the Random Forest classifier using various metrics such as accuracy, precision, recall, F1-score, or confusion matrix.

```
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy:.2f}')

# Other evaluation metrics
print(classification_report(y_test, y_pred))
print(confusion_matrix(y_test, y_pred))
```

6. Tune Hyperparameters (if needed):
   You can fine-tune the hyperparameters of the Random Forest classifier using techniques like cross-validation to optimize its performance.

That's how you can perform Random Forest classification using Scikit-learn. Make sure to adjust hyperparameters and preprocessing steps based on your specific dataset and problem.

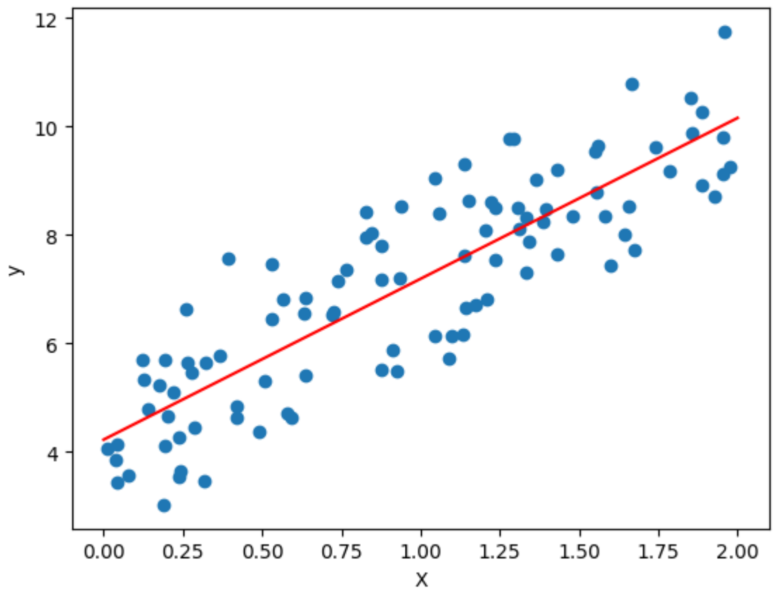
Other classification algorithms can be implemented following the same previous steps

Topic 5: Regression

Regression is a machine learning algorithm used for predicting a continuous numeric outcome (dependent variable) based on one or more input features (independent variables). It aims to find a mathematical relationship that best fits the data. Linear regression, for example, tries to find a linear equation that minimizes the difference between the predicted values and the actual data points. The result is a line (in simple linear regression) or a hyperplane (in multiple linear regression) that can be used for making predictions. Other types of regression algorithms, like polynomial regression or support vector regression, extend this concept to capture more complex relationships between variables.

A simple example of linear regression model:

```
import numpy as np
from sklearn.linear_model import LinearRegression
import matplotlib.pyplot as plt

# Generate some sample data
np.random.seed(0)
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)

# Create a Linear Regression model
model = LinearRegression()

# Fit the model to the data
model.fit(X, y)

# Make predictions
X_new = np.array([[0], [2]])
y_pred = model.predict(X_new)

# Plot the data and regression line
plt.scatter(X, y)
plt.plot(X_new, y_pred, "r-")
plt.xlabel("X")
plt.ylabel("y")
plt.show()
```



Another example of regression model:

Random Forest Regressor

```
from sklearn.ensemble import RandomForestRegressor
```

```
#Create a Gaussian Classifier
clf=RandomForestRegressor(n_estimators=100)
```

Fit the model on training data after splitting the data using trainTestSplit() function, which done after doing all the necessary preprocessing techniques.

```
#Train the model using the training sets y_pred=clf.predict(X_test)
clf.fit(X_train,y_train)
```

```
# prediction on test set
y_predrf=clf.predict(X_test)
```

```
#Import scikit-learn metrics module for accuracy calculation
from sklearn.metrics import r2_score
r2_score(y_test, y_predrf)
```

You can also evaluate your regression models using MSE, RMSE, and MAE.

```
import numpy as np
from sklearn.metrics import mean_squared_error, mean_absolute_error
from math import sqrt

# Calculate Mean Squared Error (MSE)
mse = mean_squared_error(actual, predicted)

# Calculate Mean Absolute Error (MAE)
mae = mean_absolute_error(actual, predicted)

# Calculate Root Mean Squared Error (RMSE)
rmse = sqrt(mse)

print(f"Mean Squared Error (MSE): {mse:.2f}")
print(f"Mean Absolute Error (MAE): {mae:.2f}")
print(f"Root Mean Squared Error (RMSE): {rmse:.2f}")
```

Topic 7: Text Mining

Text mining, also known as text analysis or text data mining, is the process of extracting meaningful information and knowledge from unstructured textual data. It involves various techniques and methods to uncover patterns, insights, and relationships within large collections of text. Here are some key aspects and techniques involved in text mining:

1. Text Preprocessing: This is the initial step where raw text data is cleaned and transformed. Common preprocessing tasks include tokenization (splitting text into words or phrases), removing stopwords, stemming (reducing words to their base form), and handling special characters.
2. Text Retrieval: Text mining can involve searching and retrieving specific documents or passages of text based on queries or keywords. This is commonly seen in information retrieval systems and search engines.
3. Text Classification: Text data can be categorized into predefined classes or categories. This is commonly used in spam detection, sentiment analysis, and topic categorization.
4. Text Clustering: Clustering involves grouping similar documents or pieces of text into clusters or categories without predefined labels. It's useful for discovering patterns or topics within large text datasets.
5. Named Entity Recognition (NER): NER is a technique used to identify and classify named entities such as names of people, places, organizations, and dates within text.
6. Sentiment Analysis: This involves determining the sentiment or emotional tone of a piece of text, typically as positive, negative, or neutral. It's often used in analyzing social media data and customer reviews.
7. Topic Modeling: Topic modeling techniques, like Latent Dirichlet Allocation (LDA), aim to discover hidden topics or themes within a collection of documents.
8. Text Summarization: Text summarization techniques generate concise and meaningful summaries of longer texts. This is useful for news articles, research papers, and more.
9. Text Mining Tools: Various tools and libraries, such as NLTK, spaCy, and Gensim in Python, and commercial solutions like IBM Watson and Google Cloud Natural Language, are used for text mining tasks.
10. Natural Language Processing (NLP): NLP is a field of artificial intelligence that plays a crucial role in text mining. It involves developing models and algorithms to understand and work with human language.

Text mining has a wide range of applications, including information retrieval, content recommendation, market research, fraud detection, and many others. It is a valuable technique for deriving insights from the vast amount of textual data available in documents, emails, social media, and other sources.

Tokenization is a fundamental natural language processing (NLP) technique that involves breaking down a text or a sequence of characters (usually a sentence or document) into smaller units called tokens. Tokens are typically words, phrases, symbols, or other meaningful elements that are useful for further text analysis. Here are some key points about tokenization:

1. Word Tokenization: This is the most common form of tokenization, where a text is divided into individual words. For example, the sentence “I love NLP” would be tokenized into three tokens: [“I”, “love”, “NLP”].
2. Sentence Tokenization: In some cases, text needs to be segmented into sentences. This is useful for tasks such as language translation and text summarization. For example, the text “This is sentence one. This is sentence two.” would be tokenized into two sentence tokens.
3. Phrase Tokenization: In specific contexts, it might be necessary to extract meaningful phrases or n-grams (combinations of n words) from the text. For instance, “New York City” could be considered a single token instead of three separate words.
4. Token Types: Tokens can represent different types of elements, including words, punctuation marks, numbers, or symbols, depending on the specific use case.
5. Tokenization Libraries: Tokenization is typically performed using natural language processing libraries like NLTK (Natural Language Toolkit), spaCy, or tokenizer tools provided by machine learning frameworks such as TensorFlow and PyTorch.
6. Cleaning and Preprocessing: Tokenization is often a part of text preprocessing, where you remove unwanted characters, convert text to lowercase, and handle special cases like contractions and possessives.
7. Applications: Tokenization is a crucial step in various NLP tasks, including text classification, sentiment analysis, named entity recognition, part-of-speech tagging, and more.

Here’s a simple Python example using NLTK for word tokenization:

```python
# Importing necessary library
import pandas as pd
import numpy as np
import nltk
nltk.download('punkt')
import os
import nltk.corpus
# sample text for performing tokenization
text = "In Brazil they drive on the right-hand side of the road. Brazil
has a large coastline on the eastern side of South America"
# importing word_tokenize from nltk
from nltk.tokenize import word_tokenize
# Passing the string text into word tokenize for breaking the sentences
token = word_tokenize(text)
token
```

```
['In',
 'Brazil',
 'they',
 'drive',
 'on',
 'the',
 'right-hand',
 'side',
 'of',
 'the',
 'road',
 '.',
 'Brazil',
 'has',
 'a',
 'large',
 'coastline',
 'on',
 'the',
 'eastern',
 'side',
 'of',
 'South',
 'America']
```

```
# finding the frequency distinct in the tokens
# Importing FreqDist library from nltk and passing token into FreqDist
from nltk.probability import FreqDist
fdist = FreqDist(token)
fdist
```

The output:

```
FreqDist({'the': 3, 'Brazil': 2, 'on': 2, 'side': 2, 'of': 2, 'In': 1,
'they': 1, 'drive': 1, 'right-hand': 1, 'road': 1, ...})
```

```
# To find the frequency of top 10 words
fdist1 = fdist.most_common(10)
fdist1
```

Steaming:

```
# Importing Porterstemmer from nltk library
# Checking for the word 'giving'
from nltk.stem import PorterStemmer
pst = PorterStemmer()
pst.stem('reading')
```

The output: read

```python
# Checking for the list of words
stm = ['giving', 'given', 'given', 'gave']
for word in stm :
   print(word+ ":" +pst.stem(word))
```

The output:

```
giving:give
given:given
given:given
gave:gave
```

```python
# Importing LancasterStemmer from nltk
from nltk.stem import LancasterStemmer
lst = LancasterStemmer()
stm = ['giving', 'given', 'given', 'gave']
for word in stm :
 print(word+ ':' +lst.stem(word))
```

The Porter Stemmer and the Lancaster Stemmer are both algorithms for stemming in natural language processing, but they have some differences:

1. Porter Stemmer:
   - Developed by Martin Porter in 1980.
   - It's a more widely used and cited stemming algorithm.
   - Tends to be more gentle in its stemming. It doesn't aggressively reduce words, which makes it suitable for some applications like information retrieval.
   - It is based on a series of heuristic rules that remove suffixes from words.
2. Lancaster Stemmer:
   - Developed by Chris D. Paice in 1990.
   - Known for its aggressiveness in stemming, often resulting in very short and sometimes less recognizable stems.
   - It uses a more iterative and rule-based approach compared to the Porter Stemmer

```
#A list of words to be stemmed and compare between porter and lancer
word_list = ["friend", "friendship", "friends",
"friendships","stabil","destabilize","misunderstanding","railroad","moonli
ght","football"]
print("{0:20}{1:20}{2:20}".format("Word","Porter Stemmer","lancaster
Stemmer"))
for word in word_list:

print("{0:20}{1:20}{2:20}".format(word,porter.stem(word),lancaster.stem(wo
rd)))
```

```
Word                Porter Stemmer      lancaster Stemmer
friend              friend              friend
friendship          friendship          friend
friends             friend              friend
friendships         friendship          friend
stabil              stabil              stabl
destabilize         destabil            dest
misunderstanding    misunderstand       misunderstand
railroad            railroad            railroad
moonlight           moonlight           moonlight
football            footbal             footbal.
```

Lemmatization

In simpler terms, it is the process of converting a word to its base form. The difference between stemming and lemmatization is, lemmatization considers the context and converts the word to its meaningful base form, whereas stemming just removes the last few characters, often leading to incorrect meanings and spelling errors.

For example, lemmatization would correctly identify the base form of ‘caring’ to ‘care,’ whereas stemming would cutoff the ‘ing’ part and convert it into a car.

Importing and downloading necessary tools

```
# Importing Lemmatizer library from nltk
from nltk.stem import WordNetLemmatizer
lemmatizer = WordNetLemmatizer()
nltk.download('wordnet')
nltk.download('omw-1.4')
```

```python
import nltk
from nltk.stem import WordNetLemmatizer
wordnet_lemmatizer = WordNetLemmatizer()

sentence = "He was running and eating at same time. He has bad habit of 
swimming after playing long hours in the Sun."
punctuations="?:!.,;"
sentence_words = nltk.word_tokenize(sentence)
for word in sentence_words:
    if word in punctuations:
        sentence_words.remove(word)

sentence_words
print("{0:20}{1:20}".format("Word","Lemma"))
for word in sentence_words:
    print ("{0:20}{1:20}".format(word,wordnet_lemmatizer.lemmatize(word, 
pos="v")))
```

The output:

```
Word                Lemma
He                  He
was                 be
running             run
and                 and
eating              eat
at                  at
same                same
time                time
He                  He
has                 have
bad                 bad
habit               habit
of                  of
swimming            swim
after               after
playing             play
long                long
hours               hours
in                  in
the                 the
Sun                 Sun
```

Stop words:

Stop words” are the most common words in a language like “the”, “a”, “at”, “for”, “above”, “on”, “is”, “all”. These words do not provide any meaning and are usually removed from texts. We can remove these stop words using nltk library.

```python
# importing stopwors from nltk library
from nltk import word_tokenize
from nltk.corpus import stopwords
nltk.download('stopwords')
a = set(stopwords.words('english'))
text = 'Cristiano Ronaldo was born on February 5, 1985, in Funchal, Madeira, Portugal.'
text1 = word_tokenize(text.lower())
print(text1)
stopwords = [x for x in text1 if x not in a]
print(stopwords)
```

Part of speech tagging

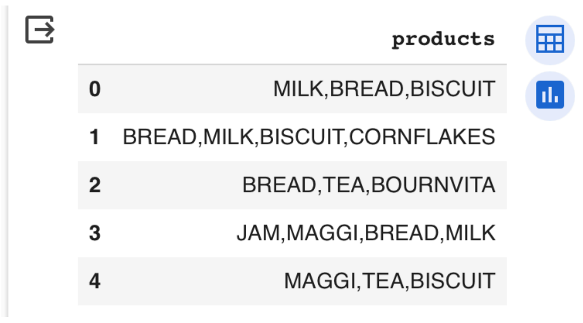
```python
text = 'vote to choose a particular man or a group (party) to represent them in parliament'
#Tokenize the text
nltk.download('averaged_perceptron_tagger') # download the tags
tex = word_tokenize(text)
for token in tex:
  print(nltk.pos_tag([token]))
```

Topic 8: Association

Apriori Algorithm implementation:

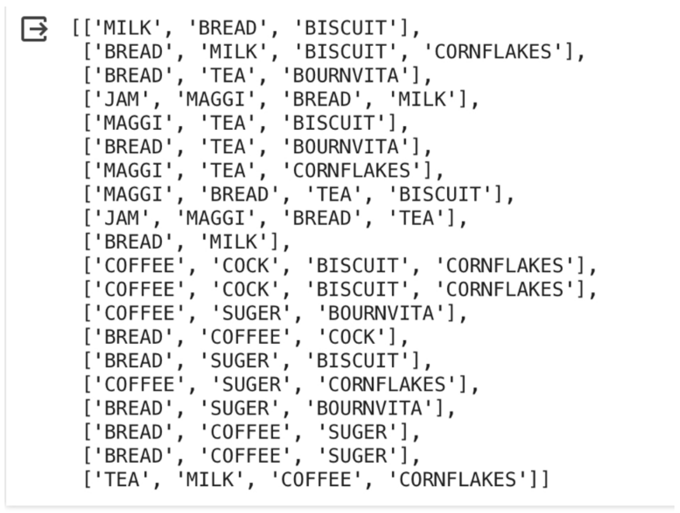
1- **Load the dataset**

```
import pandas
df=pandas.read_csv('/content/drive/MyDrive/COMP453_DataScience_CS/CO
MP435 Data Science/LecturesSlides/Datasets/GroceryStoreDataSet.csv',
names = ['products'], sep = ',')
df.head()
```



	products
0	MILK,BREAD,BISCUIT
1	BREAD,MILK,BISCUIT,CORNFLAKES
2	BREAD,TEA,BOURNVITA
3	JAM,MAGGI,BREAD,MILK
4	MAGGI,TEA,BISCUIT

```
2- #Let’s split the products and create a list called by ‘data’,
   data= list(df["products"].apply(lambda x:x.split(",") ))
   data
```



```
[['MILK', 'BREAD', 'BISCUIT'],
 ['BREAD', 'MILK', 'BISCUIT', 'CORNFLAKES'],
 ['BREAD', 'TEA', 'BOURNVITA'],
 ['JAM', 'MAGGI', 'BREAD', 'MILK'],
 ['MAGGI', 'TEA', 'BISCUIT'],
 ['BREAD', 'TEA', 'BOURNVITA'],
 ['MAGGI', 'TEA', 'CORNFLAKES'],
 ['MAGGI', 'BREAD', 'TEA', 'BISCUIT'],
 ['JAM', 'MAGGI', 'BREAD', 'TEA'],
 ['BREAD', 'MILK'],
 ['COFFEE', 'COCK', 'BISCUIT', 'CORNFLAKES'],
 ['COFFEE', 'COCK', 'BISCUIT', 'CORNFLAKES'],
 ['COFFEE', 'SUGER', 'BOURNVITA'],
 ['BREAD', 'COFFEE', 'COCK'],
 ['BREAD', 'SUGER', 'BISCUIT'],
 ['COFFEE', 'SUGER', 'CORNFLAKES'],
 ['BREAD', 'SUGER', 'BOURNVITA'],
 ['BREAD', 'COFFEE', 'SUGER'],
 ['BREAD', 'COFFEE', 'SUGER'],
 ['TEA', 'MILK', 'COFFEE', 'CORNFLAKES']]
```

```
3- #Let's transform the list, with one-hot encoding
   from mlxtend.preprocessing import TransactionEncoder
   a = TransactionEncoder()
   a_data = a.fit(data).transform(data)
   df = pandas.DataFrame(a_data,columns=a.columns_)
   df = df.replace(False,0)
   df
```

	BISCUIT	BOURNVITA	BREAD	COCK	COFFEE	CORNFLAKES	JAM	MAGGI	MILK	SUGER	TEA
0	True	0	True	0	0	0	0	0	True	0	0
1	True	0	True	0	0	True	0	0	True	0	0
2	0	True	True	0	0	0	0	0	0	0	True
3	0	0	True	0	0	0	True	True	True	0	0
4	True	0	0	0	0	0	0	True	0	0	True
5	0	True	True	0	0	0	0	0	0	0	True
6	0	0	0	0	0	True	0	True	0	0	True
7	True	0	True	0	0	0	0	True	0	0	True
8	0	0	True	0	0	0	True	True	0	0	True
9	0	0	True	0	0	0	0	0	True	0	0
10	True	0	0	True	True	True	0	0	0	0	0
11	True	0	0	True	True	True	0	0	0	0	0
12	0	True	0	0	True	0	0	0	0	True	0
13	0	0	True	True	True	0	0	0	0	0	0
14	True	0	True	0	0	0	0	0	0	True	0
15	0	0	0	0	True	True	0	0	0	True	0

The next step is to create the Apriori Model. We can change all the parameters in the Apriori Model in the mlxtend package. I will try to use minimum support parameters for this modeling. For this, I set a min_support value with a threshold value of 20%.

	support	itemsets
0	0.35	(BISCUIT)
1	0.2	(BOURNVITA)
2	0.65	(BREAD)
3	0.4	(COFFEE)
4	0.3	(CORNFLAKES)
5	0.25	(MAGGI)
6	0.25	(MILK)
7	0.3	(SUGER)
8	0.35	(TEA)
9	0.2	(BREAD, BISCUIT)
10	0.2	(BREAD, MILK)
11	0.2	(BREAD, SUGER)
12	0.2	(BREAD, TEA)
13	0.2	(COFFEE, CORNFLAKES)
14	0.2	(SUGER, COFFEE)
15	0.2	(TEA, MAGGI)

I chose the 60% minimum confidence value. In other words, when product X is purchased, we can say that the purchase of product Y is 60% or more.

```
#Let's view our interpretation values using the Associan rule function.
df_ar = association_rules(df, metric = "confidence", min_threshold = 0.6)
df_ar
```

	antecedents	consequents	antecedent support	consequent support	support	confidence	lift	leverage	conviction
0	(MILK)	(BREAD)	0.25	0.65	0.2	0.800000	1.230769	0.0375	1.75
1	(SUGER)	(BREAD)	0.30	0.65	0.2	0.666667	1.025641	0.0050	1.05
2	(CORNFLAKES)	(COFFEE)	0.30	0.40	0.2	0.666667	1.666667	0.0800	1.80
3	(SUGER)	(COFFEE)	0.30	0.40	0.2	0.666667	1.666667	0.0800	1.80
4	(MAGGI)	(TEA)	0.25	0.35	0.2	0.800000	2.285714	0.1125	3.25

For example, if we examine our 1st index value;

The probability of seeing sugar sales is seen as 30%. Bread intake is seen as 65%. We can say that the support of both of them is measured as 20%. 67% of those who buys sugar, buys bread as well. Users who buy sugar will likely consume 3% more bread than users who don’t buy sugar. Their correlation with each other is seen as 1.05.