

# SRI VASAVI ENGINEERING COLLEGE

PEDATADEPALLI, TADEPALLIGUDEM



## Certificate

*This is to certify that this is a bonafide record of Practical Work done in **Machine Learning Lab using Python** by Mr./Miss \_\_\_\_\_ bearing Roll No. \_\_\_\_\_ of CSE Branch of **VI-Semester** during the academic year 2022 to 2023.*

No. of Experiments Done: \_\_\_\_

**Faculty In-Charge**

**Head of the Department**

**External Examiner**

Exp. No.	Date	Experiment Name	Page No.	Sign
1		Introduction to required python libraries such as Numpy, Pandas, Scipy, Matplotlib, and Scikit-learn.	1	
2		Import, preprocess, and split the datasets using scikit-learn.	17	
3		Construct a classification model using the Bayes classifier using Python Programming.	21	
4		Implement a Logistic Regression algorithm for binary classification using Python Programming.	24	
5		Implement the KNN algorithm for classification and demonstrate the process of finding out the optimal "K" value using Python Programming.	26	
6		Construct an SVM classifier using python programming.	29	
7		Demonstrate the process of the Decision Tree construction for classification problems using python programming.	31	
8		Implement an Ensemble Learner using Random Forest Algorithm using python programming.	34	
9		Implement an Ensemble Learner using Adaboost Algorithm using Python programming.	36	
10		Demonstrate the working of Multi-layer perceptron with MLPClassifier() using Python programming.	39	
11		Demonstrate the K-Means algorithm for the given data set using Python programming	43	

## **System Requirements:**

- **OS**
  - Windows 10 x86\_64 or newer
  - macOS 10.14+, 64-bit
  - Ubuntu 14+/Centos7+, 64-bit
- **Hardware**
  - 16GB RAM
  - 200GB HDD
  - Quad Core Processor
  - NVIDIA GPU (Optional)
- **Python**
  - 2.7 and 3.5+.
- **IDE**
  - Anaconda Navigator
  - Jupyter Notebook

## **Installation Process (Windows):**

### **Python:**

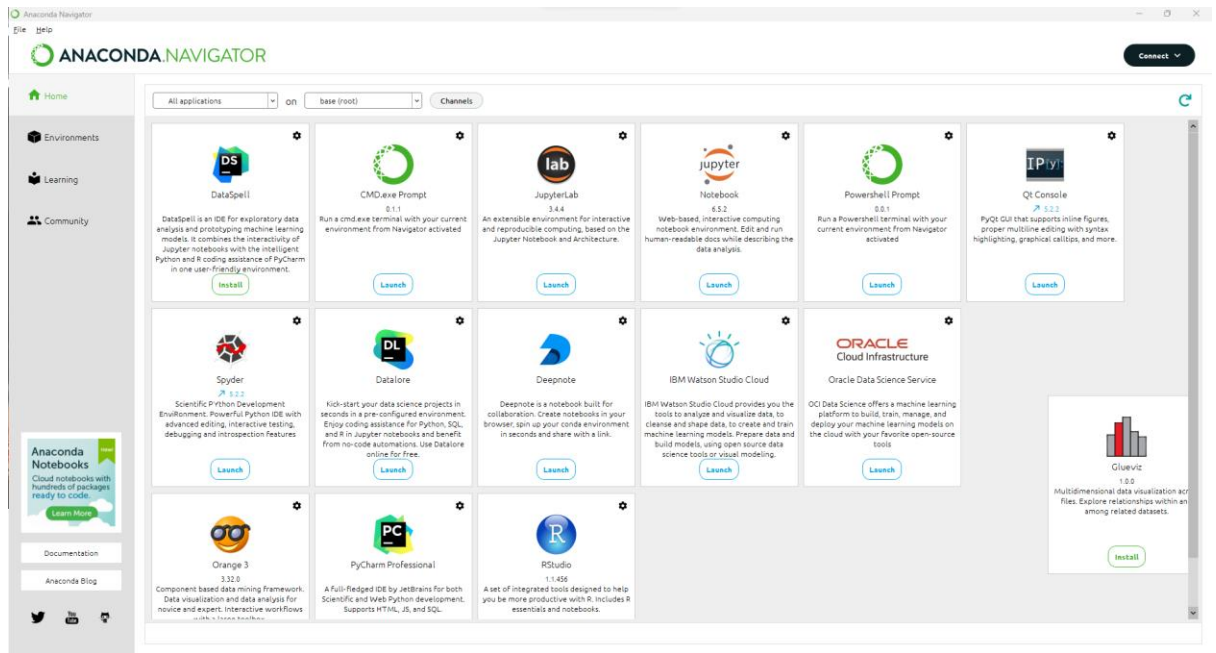
- Download the installer from <https://www.python.org/downloads/>.
- Run the downloaded "python xxxxx.exe" and follow the instructions.

### **Anaconda & Its navigator:**

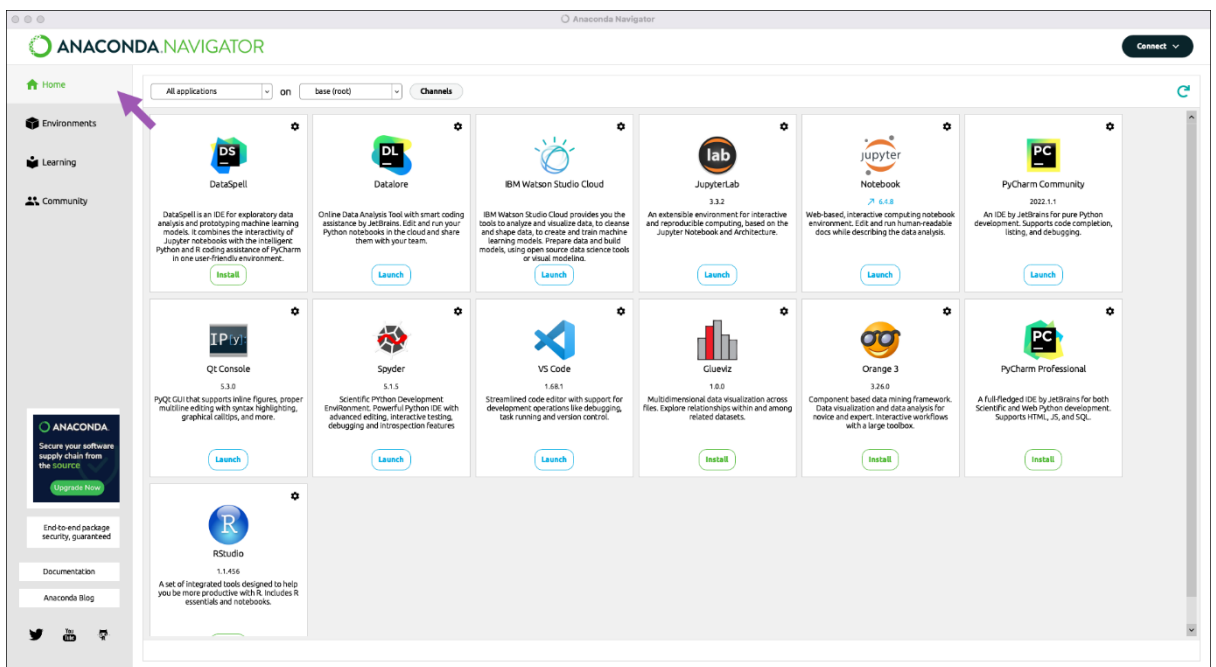
Anaconda is an open-source distribution of the Python and R programming languages for data science that aims to simplify package management and deployment.

Whereas an Anaconda Navigator is a desktop graphical user interface (GUI) included in Anaconda® distribution that allows you to launch applications and manage conda packages, environments, and channels without using command line interface (CLI) commands.

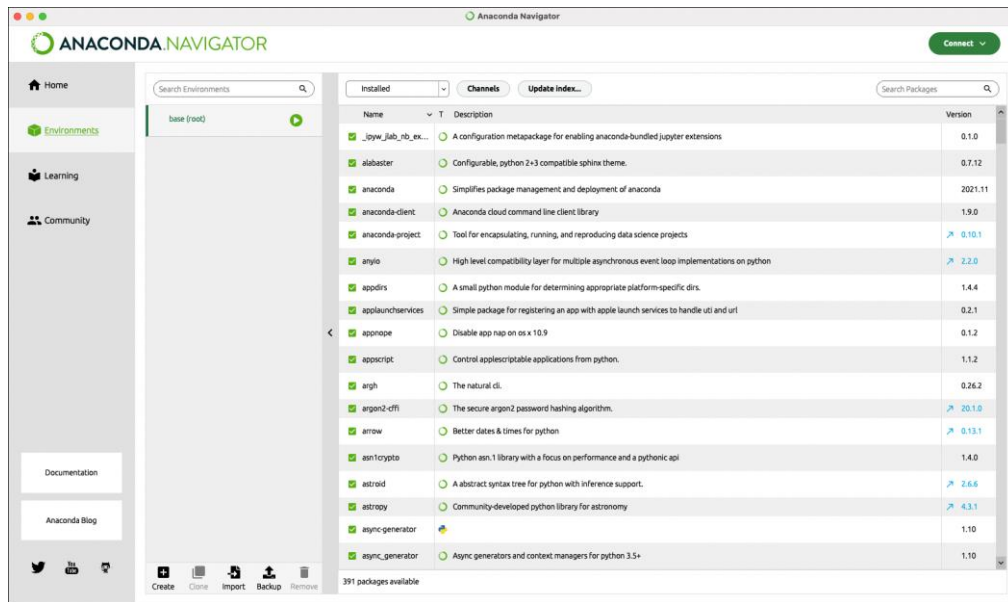
- Once Python is installed in the system, get the installer from <https://www.anaconda.com/>.
- Run the downloaded "Anaconda3-XXXXX.exe" and follow the instructions.
- Once the installation is done, open the anaconda navigator from the windows start menu.
- The Navigator will look likes as:



- As one can see, the anaconda comes with many tools to work with Python, R, etc.
- On the left side of the Navigator, we can see four tabs: Home, Environments, Learning, and Community.



- The Home page is open by default when Navigator starts. Home displays all of the available applications that you can manage with Navigator.
- The first time you open Navigator, many popular graphical Python applications are already installed or are available to install (**Jupyter Notebook**).
- The Environments page allows you to manage installed environments, packages, and channels.



- The left column lists your environments. Click an environment to activate it.
- Environments
  - With Navigator, you can create, export, list, remove, and update environments that have different versions of Python and other packages installed. Switching or moving between environments is called activating the environment. Only one environment is active at any point in time. For more information, see Managing environments.
- Packages
  - The right column lists packages installed in the currently-activated environment. The default view is Installed packages. To change which packages are displayed, click the arrow next to the list, then select Installed, Not Installed, Updatable, Selected, or All packages. For more information, see Managing packages.
- Channels
  - Channels are locations where Navigator or conda looks for packages. Click the Channels button to open the Channels Manager and modify which channels Navigator uses. For more information, see Managing channels.

**Note: It is recommended to create an environment for you to work with ML with Python.**

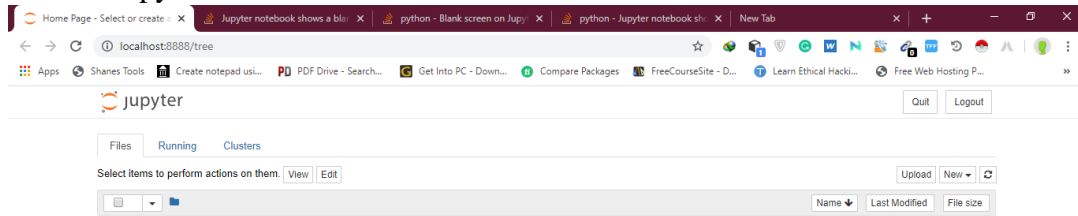
1. Click on "crate" at the bottom of the environments column.
2. Give your registration number as "SECTION\_REGNO," select Python checkbox and click create.  
Ex: CSEA\_20A81A0501
3. Once the environment is created, go back home and launch Jupyter notebook.  
Note: If needed, install Jupyter notebook.

## Jupyter Notebook:

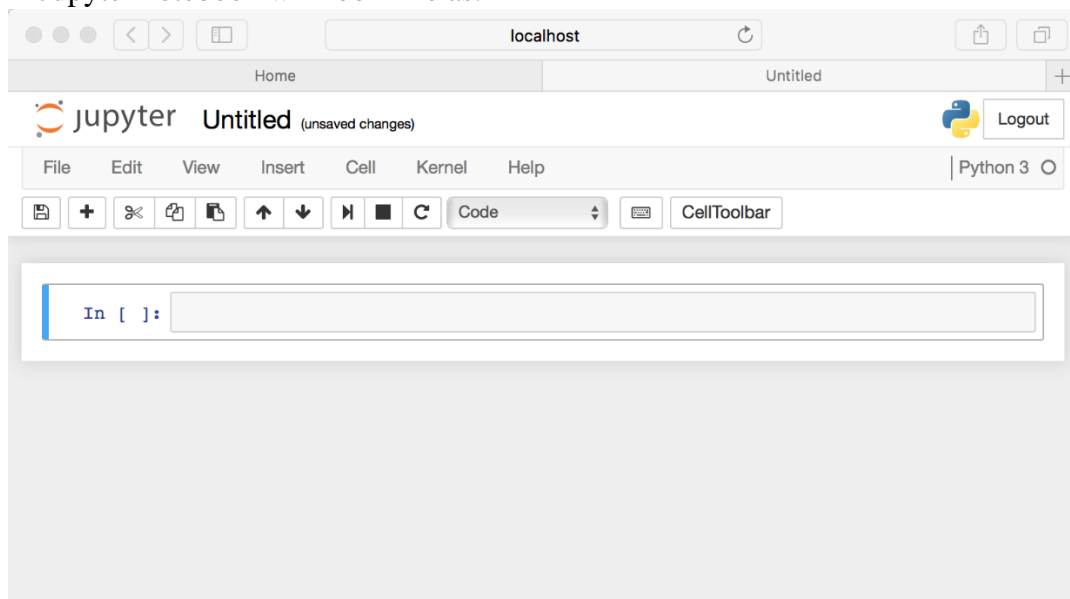
Jupyter Notebook (formerly IPython Notebook) is a web-based interactive computational environment for creating notebook documents. A Jupyter Notebook application is a browser-based REPL containing an ordered list of input/output cells which

can contain code, text (using Markdown), mathematics, plots and rich media. Jupyter Notebook is built using several open-source libraries, including IPython, ZeroMQ, Tornado, jQuery, Bootstrap, and MathJax.

- A Jupyter notebook will be launched in the browser with the home page that shows all the directories/folders in the default directory.
- A fresh Jupyter notebook home will look like as:

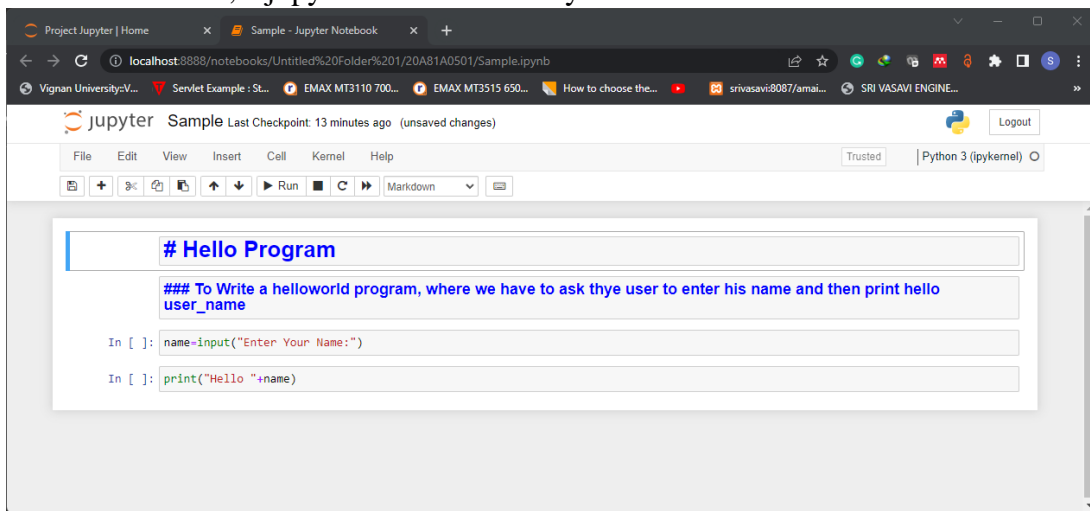


- In that home, on the left top, click on the new and select folder and rename the created **Untitled Folder** by selecting it (**check box**) and clicking on **rename** on top.  
**Note: Rename the folder with your reg\_no.**
- Click on a folder to get into it.
- Once you are in the desired directory, click on new, select **python 3** to open a new notebook, or just click on an already created notebook.
- A Jupyter notebook will look like as:

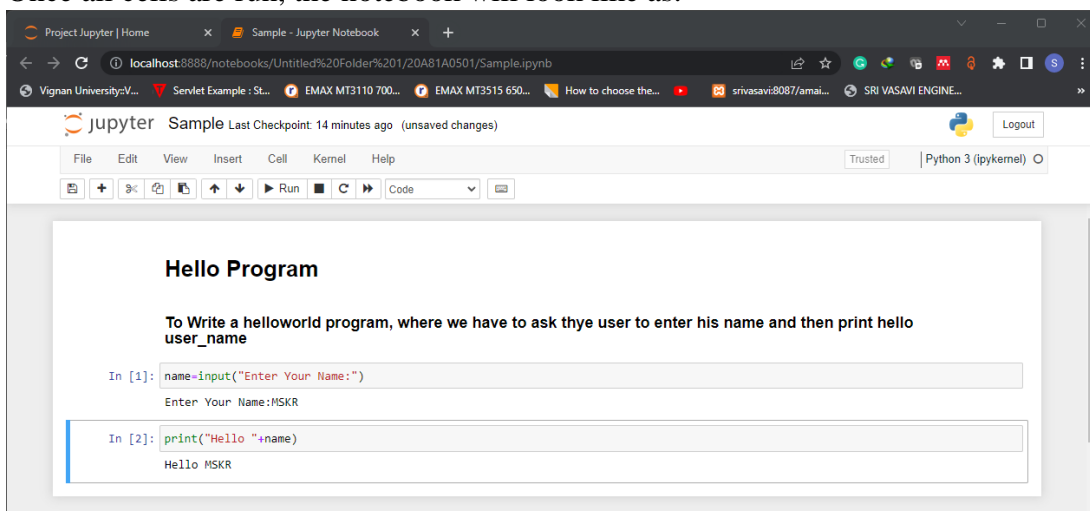


- A Jupyter notebook allows users to split the code into multiple segments and add some textual description or headings in between.

- One can use the menus or buttons or the shortcuts (<https://towardsdatascience.com/jupyter-notebook-shortcuts-bf0101a98330>) to create, delete, move, or run segments.
- As shown bellow, a jupyter notebook mainly have markdown cells and code cells.



- Once all cells are run, the notebook will look like as.



### Note:

- One can install python libraries/packages in **the anaconda navigator: environments** or by using the `!pip install <Package name>` command in jupyter notebook.
- One can also install a package from the command prompt using `"py -m pip install <Package name>"` or `"conda install <Package name>."`

## **Exp 1: Introduction to required python libraries such as Numpy, Pandas, Scipy, Matplotlib, and Scikit-learn.**

**Aim:** To familiarize with various advanced python libraries that are used in Machine Learning implementation.

### **What is a python library?**

Python libraries are collections of modules that contain useful codes and functions, eliminating the need to write them from scratch. There are tens of thousands of Python libraries that help machine learning developers, as well as professionals working in data science, data visualization, and more.

Python is the preferred language for machine learning because its syntax and commands are closely related to English, making it efficient and easy to learn. Compared with C++, R, Ruby, and Java, Python remains one of the simplest languages, enabling accessibility, versatility, and portability. It can operate on nearly any operating system or platform.

Some of the most important python libraries for machine learning are:

- **Numpy**
- **Pandas**
- **SciPy**
- **Matplotlib**
- **Sklearn**
- **Seaborn**
- TensorFlow
- PyTorch
- Keras

### **NumPy:** (Install Numpy if needed)

NumPy (Numerical Python) is an open-source Python library that's used in almost every field of science and engineering. It's the universal standard for working with numerical data in Python and is at the core of the scientific Python and PyData ecosystems. NumPy users include everyone from beginning coders to experienced researchers doing state-of-the-art scientific and industrial research and development. The NumPy API is used extensively in Pandas, SciPy, Matplotlib, scikit-learn, scikit-image, and most other data science and scientific Python packages.

**The NumPy library contains a multidimensional array and matrix data structures** (you'll find more information about this in later sections). It provides **ndarray**, a homogeneous n-dimensional array object, with methods to efficiently operate on it. NumPy can be used to perform a wide variety of mathematical operations on arrays. It adds powerful data structures to Python that guarantee efficient calculations with arrays and matrices, and it supplies an enormous library of high-level mathematical functions that operate on these arrays and matrices.



- To access NumPy and its functions import it in your Python code like this:
  - **import numpy as np**
- **Snippet:**

```
import numpy as np
```

## Why Numpy?

- In Python we have lists that serve the purpose of arrays, but they are slow to process.
- NumPy aims to provide an array object that is up to 50x faster than traditional Python lists.
- The array object in NumPy is called ndarray, it provides a lot of supporting functions that make working with ndarray very easy.
- The two most important features of Numpy are "Arrays" and "Random".
- As stated above arrays are faster and comes with many functions.

## Arrays of Numpy:

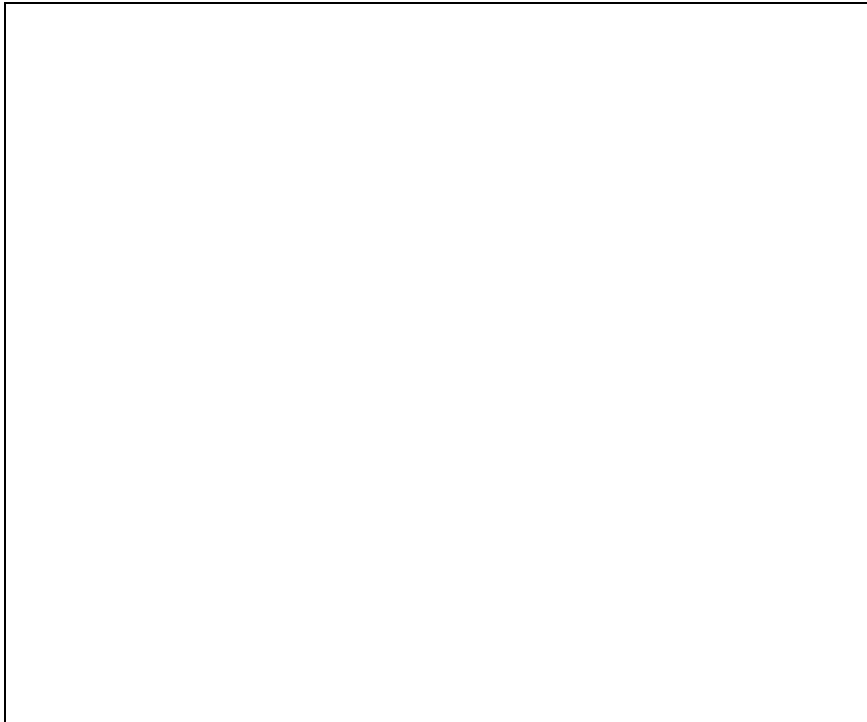
- One can convert any python object into numpy array using **numpy.array()** (**np.array()**) method.
- The indexes in NumPy arrays start with 0, meaning that the first element has index 0, and the second has index 1 etc.
- Slicing in Python means taking elements from one given index to another given index. We pass slice instead of index like this: [start:end:step], where step is optional.
- NumPy has different data types, and refer to data types with one character, like **i** for integers, **u** for unsigned integers, **b** for boolean, **f** for float, etc.
- The "**dtype**" will return the data type of an array and also can be used as an optional argument to specify the data type while creating an array.
- NumPy arrays have an attribute called **shape** that returns a tuple with each index having the number of corresponding elements.
- "**reshape**" allows one to change the shape of an array by adding or removing dimensions.
- While reshaping, It is allowed to have one "unknown" dimension. Meaning that you do not have to specify an exact number for one of the dimensions in the reshape method. Pass -1 as the value, and NumPy will calculate this number for you.

## Snippet:

```
A0=np.array(3)
A1 = np.array([1, 2, 3, 4, 5])
print(A1)
print(type(A1))
print(A1 [1])
print(A1 [1:3])
print(A1.dtype)
A2 = np.array([1.1, 2.1, 3.1],dtype= "i")
print(A2)
A3 = np.array([[1,2,3,4],[9,4,2,7]])
```

```
print(A0.shape)
print(A1.shape)
print(A2.shape)
print(A3.shape)
A4=A3.reshape(8,)
print(A4)
A5=A4.reshape(2,2,2)
print(A5)
print(A5.reshape(-1))
```

**Output:**



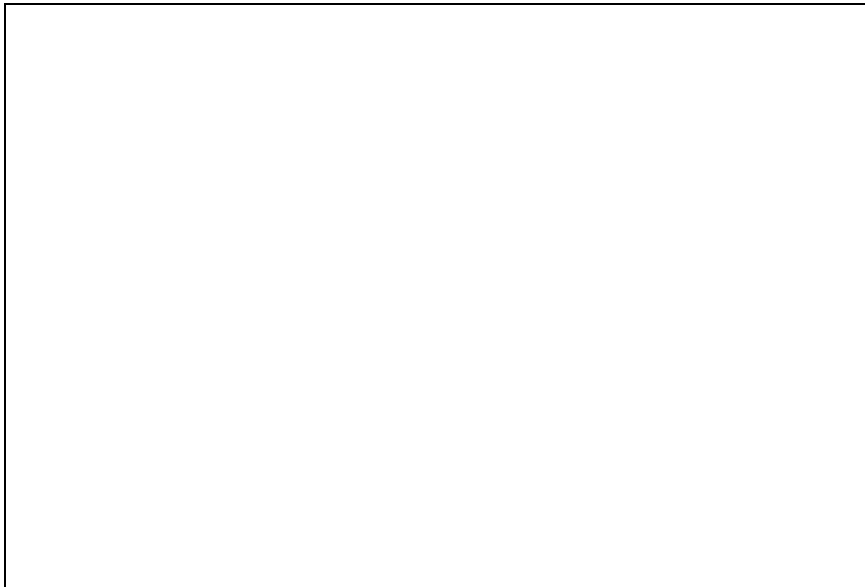
### **Random of Numpy:**

Computers work on programs, and programs are definitive set of instructions. So it means there must be some algorithm to generate a random number as well. If there is a program to generate random number it can be predicted, thus it is not truly random. Random numbers generated through a generation algorithm are called pseudo random.

- NumPy offers the **random** module to work with random numbers.
- The random module's **rand()** method returns a random float between 0 and 1.
- The random module's **randint(m,n)** method returns a random integer from m to n-1.
- The NumPy Random module provides two methods for permutate: **shuffle()** and **permutation()**. **shuffle()** rearranges the original array, whereas **permutation()** gives a new shuffled array as output.
- The random module's **choice()** method returns an element of given data structure.

**Snippet:**

```
print(np.random.rand())
print(np.random.rand())
print(np.random.randint(3))
print(np.random.randint(3))
print(np.random.randint(3,8))
A6=np.array([2,4,5,9,3,54,23,12,56,98])
np.random.shuffle(A6)
print(A6)
A7=np.random.permutation(A6)
print(A6)
print(A7)
print(np.random.choice(A6))
print(np.random.choice(A6))
```

**Output:****Pandas:** (Install Pandas if needed)

Pandas is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language. In particular, it offers data structures and operations for manipulating numerical tables and time series. It has functions for analyzing, cleaning, exploring, and manipulating data. The name "Pandas" has a reference to both "Panel Data", and "Python Data Analysis" and was created by Wes McKinney in 2008.

- To access Pandas and its functions, import it in your Python code like this:
  - **import pandas as pd**

**Snippet:**

```
import pandas as pd
```

## Why Pandas?

- Pandas allows users to store real world data in their true form as a data structure called data frame.
- It allows importing data from files with different formats into python environment with ease.
- Pandas allows us to analyze big data and make conclusions based on statistical theories.
- Pandas can clean messy data sets, and make them readable and relevant.
- Relevant data is very important in data science.

## DataFrames of Pandas:

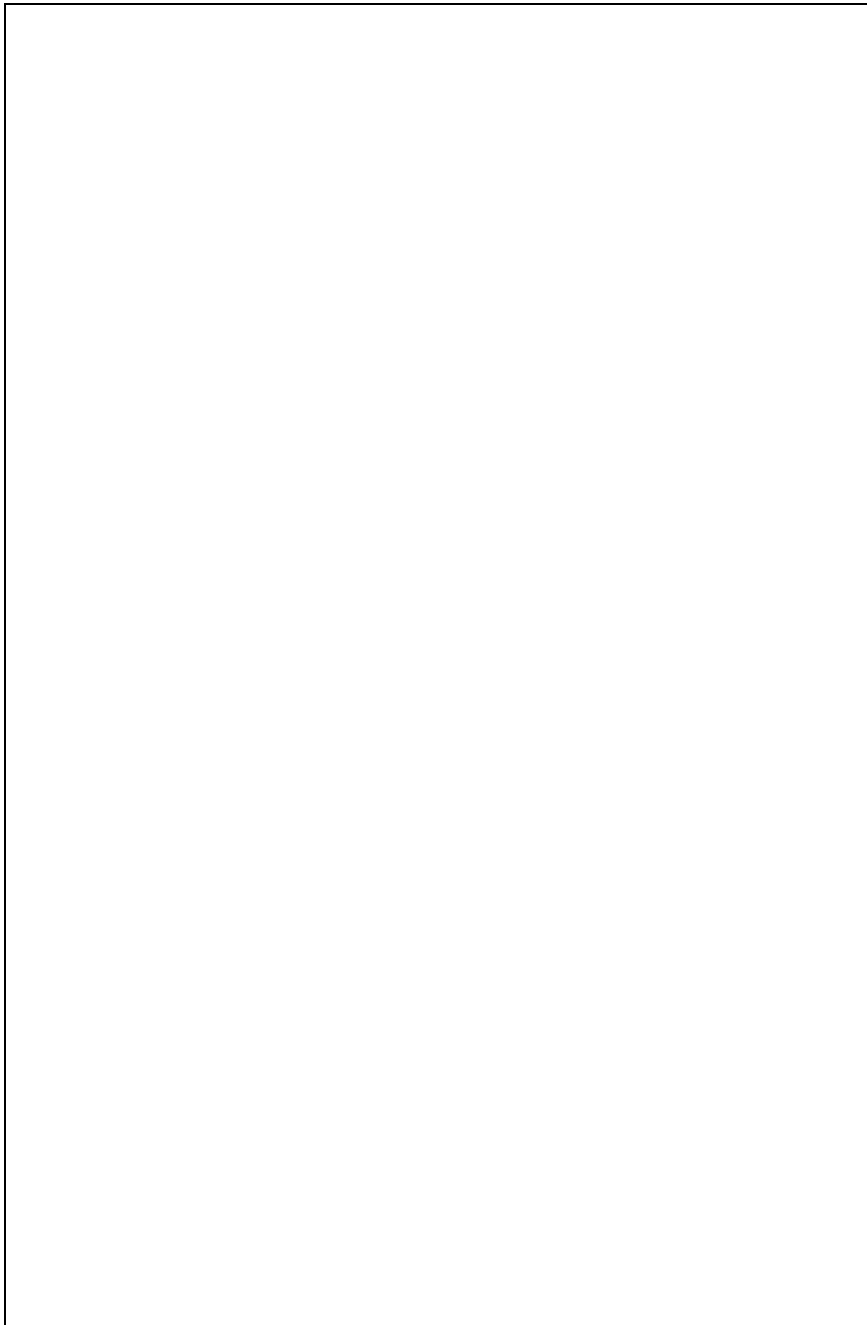
- A Pandas DataFrame is a 2 dimensional data structure, like a 2 dimensional array, or a table with rows and columns.
- A DataFrame can be created using **DataFrame()** method of pandas.
- The **head(n)** and **tail(n)** methods of pandas will print **the first n or last n** records of a DataFrame. The **n** is optional, the default value of n is **5**.
- Pandas use the **loc** attribute to return one or more specified row(s) using their names.
- Pandas use the **iloc** attribute to return one or more specified row(s) using their index.

## Snippet:

```
data = {
    'cars': ["BMW", "Volvo", "Ford", "Benz", "Maruthi", "Tata",
             "Hyundai", "Honda"],
    'passings': [3, 5, 2, 1, 17, 13, 10, 8]
}

df = pd.DataFrame(data)

print(df)
print("#####")
print(df.head())
print("#####")
print(df.tail(3))
print("#####")
print(df.loc[0])
print("#####")
print(df.loc[[1,4]])
print("#####")
print(df.iloc[[1,2]])
```

**Output:**

- The "columns" of dataframe returns the list of columns present in a dataframe.
- Once can use column name(s) with [] ([[ ]]) on a dataframe to retrieve one or more columns from it.

**Snippet:**

```
print(df.columns)
print("#####")
print(df["cars"])
print("#####")
print(df[["cars","passings"]])
```

**Output:****Importing Data:**

- Pandas support importing data from different kinds of files like .csv, .xls, .xlsx, etc.
- The read data will be normally stored in a dataframe.
- Pandas, read\_csv reads data from a CSV file.

Syntax: `pd.read_csv(filepath_or_buffer, sep=',', header='infer', index_col=None, usecols=None, engine=None, skiprows=None, nrows=None)`

- Pandas, read\_excel read data from an .xls or .xlsx file.

Syntax: `pd.read_excel(io, sheet_name=0, header=0, names=None, index_col=None, skiprows=None, nrows=None, ... ..)`

Note: Before executing the above commands, get some sample datasets from the internet in both CSV and XLS/XLSX formats.

**Snippet:**

```
iris_csv=pd.read_csv("iris.csv")
print(iris_csv.head())
```

**Output:**

**Snippet:**

```
iris_excel=pd.read_excel("iris.xlsx")
print(iris_excel.tail(3))
```

**Output:**

**SciPy:** (Install SciPy if needed)

SciPy is a free and open-source Python library used for scientific computing and technical computing. SciPy contains modules for optimization, linear algebra, integration, interpolation, special functions, FFT, signal and image processing, ODE solvers and other tasks common in science and engineering. It uses NumPy underneath.

- As SciPy is more focused on scientific implementations, it provides many built-in scientific constants.
- To access scipy constants, import it in your Python code like this:
  - **from scipy import constants**
- Optimizers are a set of procedures defined in SciPy that either finds the minimum value of a function or the root of an equation.

**Snippet:**

```
from scipy import constants
print(dir(constants))
print(constants.pi)
print(constants.e)
print(constants.u)
```

**Output:**

- SciPy is capable of finding roots for polynomials and linear equations.
- SciPy is capable of finding minima for polynomials.

**Snippet:**

```
from scipy.optimize import root
from scipy.optimize import minimize
def eqn(x):
    return x**4+2*x**3 + 3*x**2 -2*x + 1
r = root(eqn, 0)
print(r.x)
mymin = minimize(eqn, 0, method='BFGS')
print(mymin)
```

**Output:****Matplotlib:** (Install Matplotlib if needed)

Matplotlib is a low level graph plotting library in Python that serves as a visualization utility. Most of the Matplotlib utilities lies under the pyplot submodule, and are usually imported under the plt alias.



- To access Matplotlib.pyplot and its functions, import it in your Python code like this:
  - **import matplotlib.pyplot as plt**

**Snippet:**

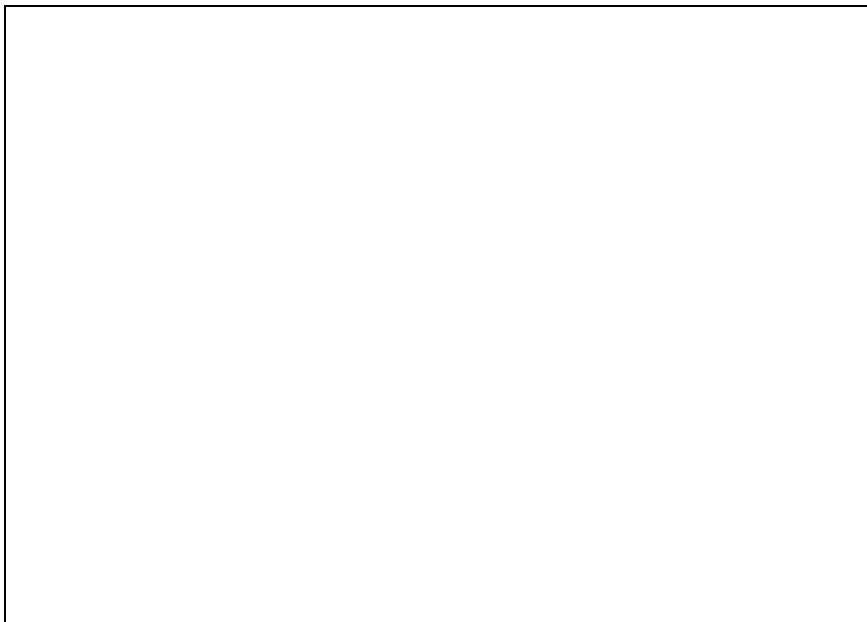
```
import matplotlib.pyplot as plt
```

- The plot() function is used to draw points (markers) in a diagram. By default, the plot() function draws a line from point to point.
- The scatter() function plots one dot for each observation.
- With Pyplot, you can use the bar() function to draw bar graphs.
- A histogram is a graph showing frequency distributions. It is a graph showing the number of observations within each given interval. The hist() function will use an array of numbers to create a histogram.
- With Pyplot, you can use the pie() function to draw pie charts.

**Snippet:**

```
X=range(-5,5)  
Y=[eqn(x) for x in X]  
plt.plot(X,Y)  
plt.show()
```

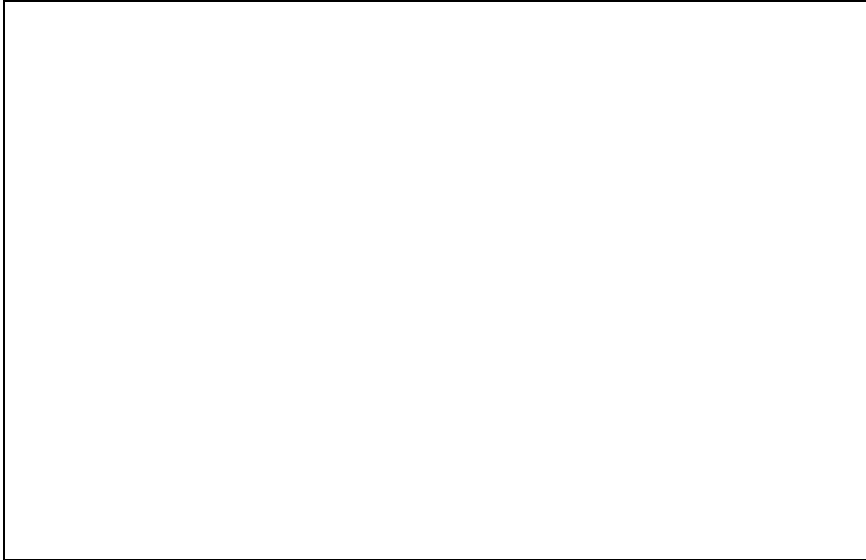
**Output:**



**Snippet:**

```
plt.plot(X,Y,marker='*')  
plt.show()
```

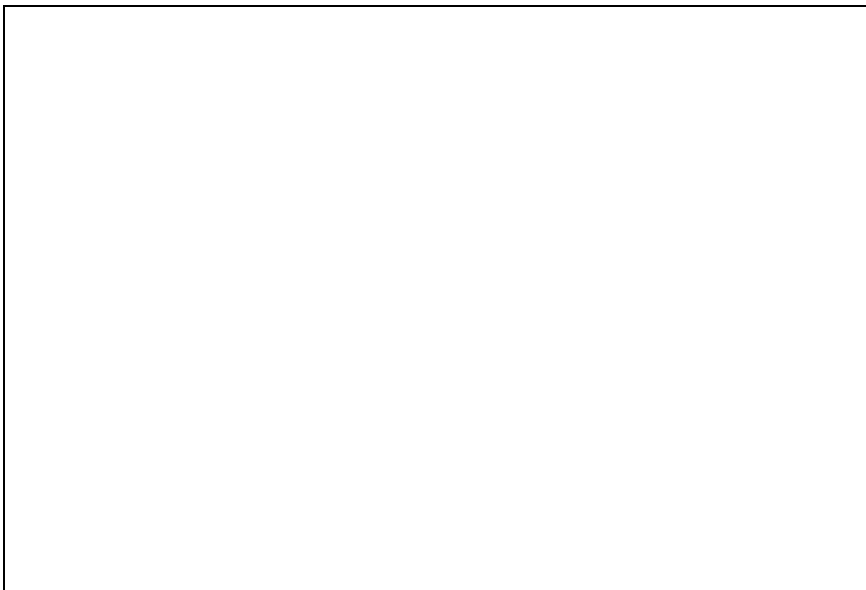
**Output:**



**Snippet:**

```
plt.plot(X,Y,linestyle=":")  
plt.show()
```

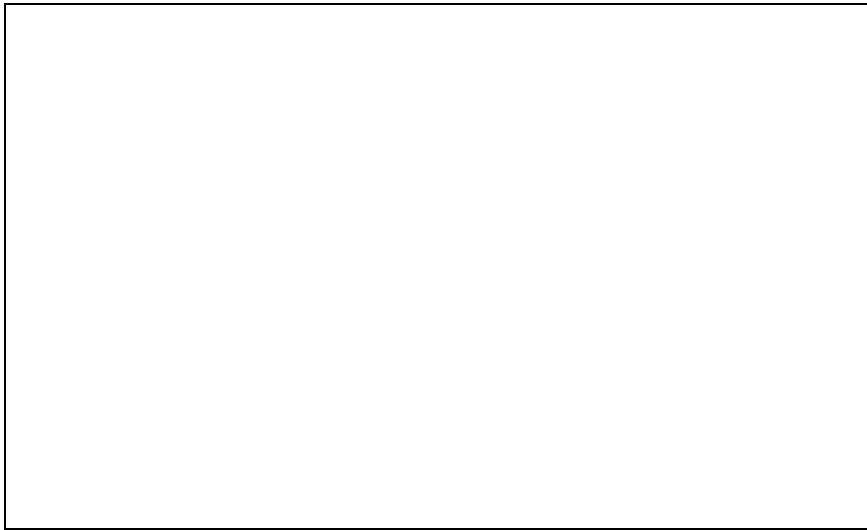
**Output:**



**Snippet:**

```
plt.plot(X,Y,linestyle=":")  
plt.xlabel("X")  
plt.ylabel("Y=f(X)")  
plt.title("Sample")  
plt.show()
```

**Output:**



**Snippet:**

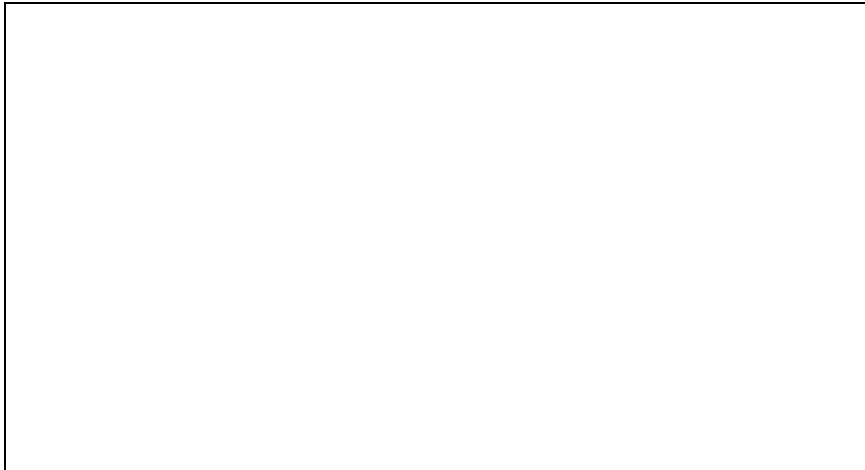
```
plt.subplot(2,2,1)
plt.plot(X,Y,linestyle=":")
plt.subplot(2,2,2)
plt.plot(X,Y,"o")
plt.subplot(2,2,3)
plt.plot(X,Y,linestyle="--")
plt.subplot(2,2,4)
plt.plot(X,Y)
plt.show()
```

**Output:**



**Snippet:**

```
plt.scatter(iris_csv["SepalLengthCm"][1:51],iris_csv["SepalWidthCm"][1:51])
plt.scatter(iris_csv["SepalLengthCm"][51:101],iris_csv["SepalWidthCm"][51:101],color="red")
plt.scatter(iris_csv["SepalLengthCm"][101:151],iris_csv["SepalWidthCm"][101:151],c="Green")
plt.show()
```

**Output:****Seaborn:** (Install Seaborn if needed)

Seaborn is a Python data visualization library based on matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphics. It provides beautiful default styles and color palettes to make statistical plots more attractive.

- To access seaborn and its functions, import it in your Python code like this:
  - **import seaborn as sns**

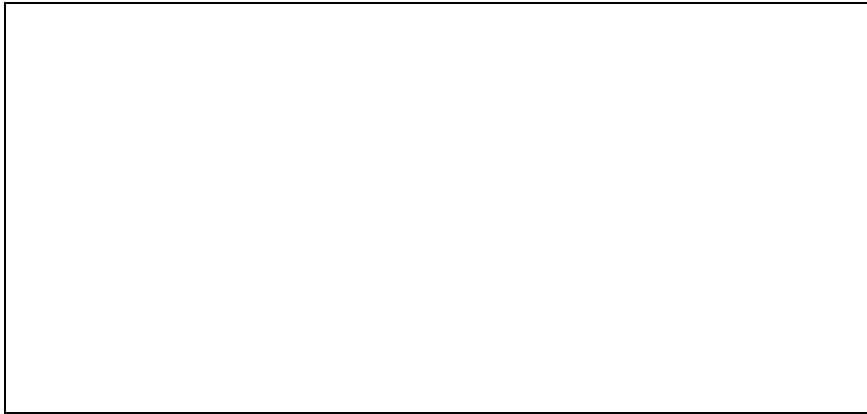
**Snippet:**

```
import seaborn as sns
```

- The `load_dataset()` load an example dataset from the online repository (requires internet). Use `get_dataset_names()` to see a list of available datasets.
- The `stripplot()` of seaborn is used for generating a strip plot. A strip plot is a single-axis scatter plot that is used to visualise the distribution of many individual one-dimensional values.
- **Snippet:**

```
print(sns.get_dataset_names())
iris=sns.load_dataset("iris")
print(iris.columns)
ax = sns.stripplot(x='species', y='sepal_length', data=iris)
plt.title('Graph')
plt.show()
```

- **Output:**



- The swarmplot() of seaborn is used for generating a swarmplot. The swarm plot looks similar to strip plot, the main difference is that in a swarm plot, the data points don't overlap and are adjusted along the categorical axis.
- **Snippet:**

```
print(sns.get_dataset_names())
iris=sns.load_dataset("iris")
print(iris.columns)
ax = sns.stripplot(x='species', y='sepal_length', data=iris)
plt.title('Graph')
plt.show()
```

- **Output:**



- The swarmplot() of seaborn is used for generating a swarmplot. The swarm plot looks similar to strip plot, the main difference is that in a swarm plot, the data points don't overlap and are adjusted along the categorical axis.
- **Snippet:**

```
print(sns.get_dataset_names())
iris=sns.load_dataset("iris")
print(iris.columns)
```

```
ax = sns.swarmplot(x='species', y='sepal_length', data=iris)
plt.title('Graph')
plt.show()
```

- **Output:**



- The swarmplot() of seaborn is used for generating a swarmplot. The swarm plot looks similar to strip plot, the main difference is that in a swarm plot, the data points don't overlap and are adjusted along the categorical axis.
- The color of the data points can be set using hue.
- **Snippet:**

```
ax = sns.scatterplot(x='sepal_length', y='sepal_width', data=iris, hue='species')
plt.title('SL vs SW')
plt.show()
```

- **Output:**



- Seaborn module comes with many other types of plots like barplots, histogram, boxplots, etc.

### **Sklearn (SciKit-Learn):** (Install sklearn if needed)

Simple and efficient tools for predictive data analysis. It is a free software machine learning library for the Python programming language. It features various classification, regression and clustering algorithms, including support-vector machines, random forests, gradient boosting, k-means, and DBSCAN, and is designed to interoperate with the Python numerical and scientific libraries NumPy and SciPy.

To access seaborn and its functions, import it in your Python code like this:

- **import sklearn**

#### **Snippet:**

```
import sklearn
```

- One can import any preprocessing or machine learning algorithm method from sklearn and train his/her own model by passing data and can do predictions.

#### **Snippet:**

```
from sklearn import linear_model
X=np.array(range(-50,50))
Y=np.array([3*x+1 for x in X])
X=X.reshape(100,-1)
regr=linear_model.LinearRegression()
regr.fit(X, Y)
print(regr.predict([[85]]))
print(3*85+1)
```

#### **Output:**

## Exp 2: Import, preprocess, and split the datasets using scikit-learn.

**Aim:** To import a dataset from a file, preprocess it, and then split it into training and testing datasets.

### Importing Libraries:

We have to first import all the necessary libraries as shown bellow.

#### Snippet:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
```

### Importing Data:

As learned in the last experiment, Pandas library comes with various methods to import data from different files and databases.

#### Snippet:

```
df=pd.read_excel("iris.xlsx")
or
df=pd.read_csv("iris_v2.csv")
```

### Data Preprocessing:

Data Preprocessing includes the steps we need to follow to transform or encode data so that the machine may easily parse it. The main agenda for a model to be accurate and precise in predictions is that the algorithm should be able to interpret the data's features efficiently.

But, most real-world datasets for machine learning are highly susceptible to being missing, inconsistent, and noisy due to their heterogeneous origin. Applying data analysis algorithms to this noisy data would not give quality results as they would fail to identify patterns effectively. Data Processing is, therefore, essential to improve the overall data quality. Duplicate or missing values may give an incorrect view of the overall statistics of data. Outliers and inconsistent data points often tend to disturb the model's overall learning, leading to false predictions. Quality decisions must be based on quality data. Data Preprocessing is important to get this quality data, without which it would just be a Garbage In, Garbage Out scenario.

Data Preprocessing mainly involves four stages:

- **Data Cleaning:** Done as part of data preprocessing to clean the data by filling in missing values, smoothing the noisy data, resolving the inconsistency, and removing outliers.
- **Data Transformation:** Once data clearing has been done, we need to consolidate the quality data into alternate forms by changing the value, structure, or format of data



using the below-mentioned Data Transformation strategies. The two important steps in transformation are generalization and normalization. Also the data has to be integrated to merge the data present in multiple sources into a larger data store like a data warehouse.

- Data Reduction: The dataset size in a data warehouse can be too large to be handled by data analysis and mining algorithms. Also, data with unnecessary features will tend to create more specific models than generalized models. In this stage, we mainly concentrate on dimensionality reduction.

### **Cleaning:**

- Removing Duplicates
- Handling Missing Values
  - Removing Rows
  - Removing Columns
  - Imputing
- Handling Outliers
- Removing Unnecessary columns.

### **Snippet:**

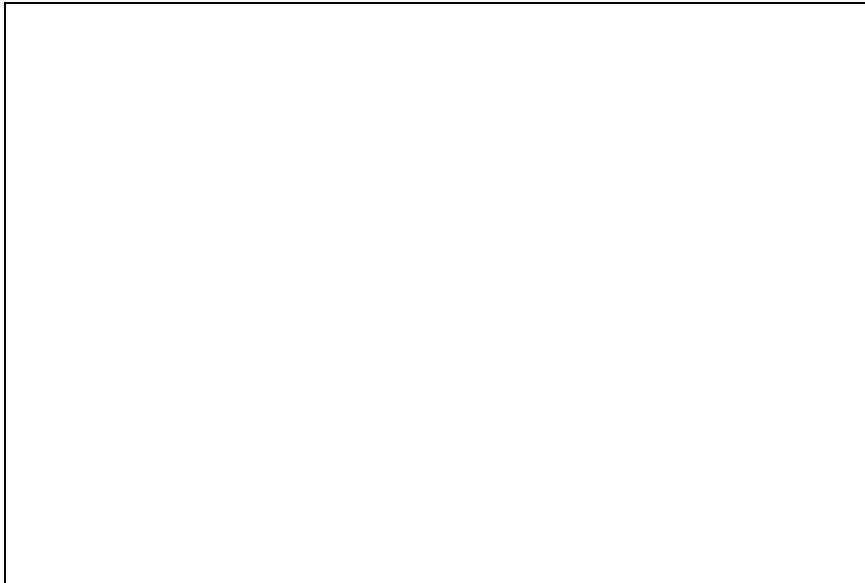
```
df1=df.copy()
print(df1.shape)
print(df1.columns)
df1=df1.drop(columns=["ID"])
print(df1.columns)
df1.drop_duplicates(inplace=True)
print(df1.shape)
print(df1.isna().sum())          # print(df1.isna().sum())
df2=df1.dropna()
print(df2.shape)
df3=df1
df3["SL"]=df3["SL"].fillna(df["SL"].mean())
df3["PL"]=df3["PL"].fillna(df["PL"].median())
print(df3.isna().sum())
print(df3.shape)
```

### **Output:**



**Snippet:**

```
plt.subplot(2,1,1)
sns.boxplot(df3["SW"])
df4=df3.loc[df3.SW<=4]
plt.subplot(2,1,2)
sns.boxplot(df4["SW"])
print(df4.shape)
```

**Output:****Transforming:**

- Transforming categorical to numerical data as majority of the algorithms cant handle categorical data.
- Transforming contonous data to discrete.
- Scaling the Data.
- Separating Independent and dependent variables. (X & Y)
- Handling Categorical Labels:
  - Label Encoding.
  - One-Hot Encoding.

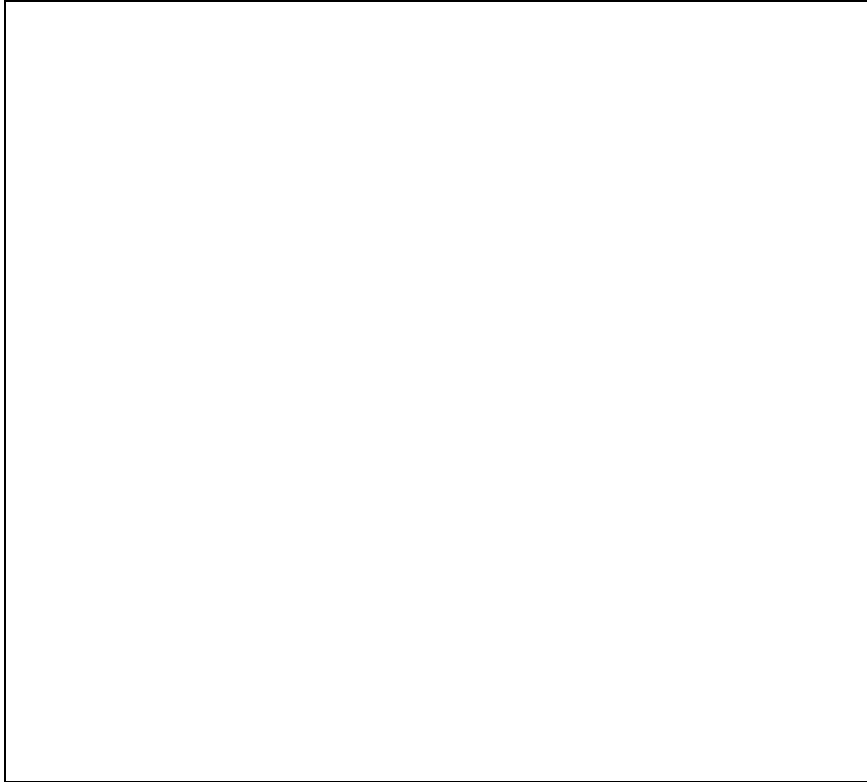
**Snippet:**

```
df5=df4.copy()
print(set(df5.Species))
df5.Species.replace({"Setosa":1,"Versicolor":2,"Virginica":3}
                    ,inplace=True)

print(set(df5.Species))
LE=preprocessing.LabelEncoder()
df6=df4.copy()
df6["Species"]=LE.fit_transform(df6["Species"])
print(set(df6.Species))
X,Y= df4.iloc[:, 0:-1],df4[["Species"]]
```

```
print(X)
print(Y)
OHE = preprocessing.OneHotEncoder()
Y_transformed=OHE.fit_transform(Y)
print(Y_transformed.toarray())
```

**Output:**



**Splitting Data into Training and Testing Data:**

**Snippet:**

```
X_train, X_test, Y_train, Y_test = train_test_split( X, Y,
test_size=0.33)
print(X_train.shape,X_test.shape)
print(Y_train.shape,Y_test.shape)
print(Y_train.head())
```

**Output:**

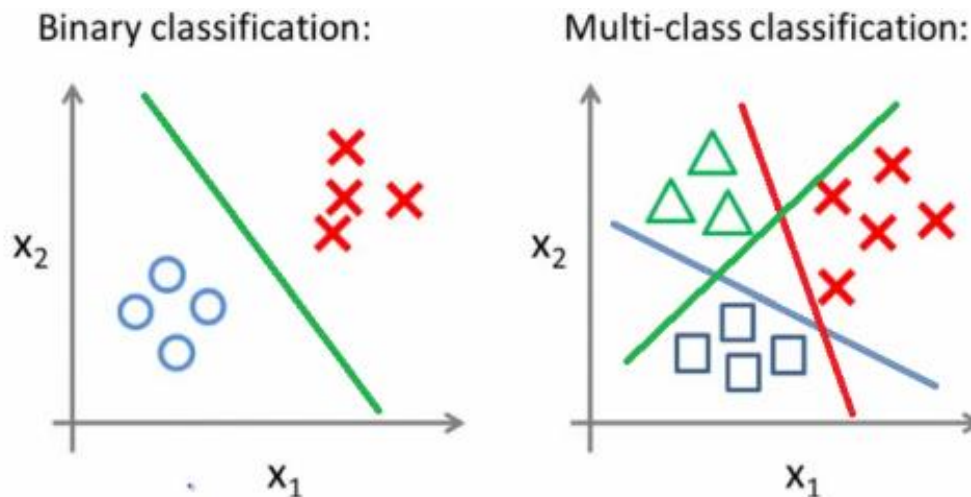


### Exp 3: Construct a classification model using the Bayes classifier using Python Programming.

**Aim:** To model a classification problem using the Bayes classifier.

#### Classification:

Is supervised learning, where the task is learning a function that maps an input to an output (Labels) based on example input-output pairs. In classification, the labels/targets are discrete in nature. A classification task based on number of labels broadly classified as binary classification and multi-class classification.



Also, based on the nature of the output of the classification model built, the algorithms are divided into probabilistic and non-probabilistic classifiers.

- Non-Probabilistic:
  - Formally, a non-probabilistic classifier, i.e., an "ordinary" classifier is some rule, or function, that assigns to a sample  $x_{new}$  a class label  $t_{new}$ .

$$t_{new} = f(x_{new})$$

- Probabilistic:
  - Probabilistic classifiers generalize this notion of classifiers: instead of functions, they are conditional distributions  $P(t_{new} = c | x_{new}, X, T)$ .
  - Meaning that for a given  $x_{new}$ , they assign probabilities to all  $t \in T$ .

#### The Bayes Classifier:

Naive Bayes methods are a set of supervised learning algorithms based on applying Bayes' theorem with the "naive" assumption of conditional independence between every pair of features given the value of the class variable. Bayes' theorem states the following relationship, given class variable  $y$  and dependent feature vector  $x_1$  through  $x_n$  :

$$P(y|x_1, \dots, x_n) = \frac{P(y)P(x_1, \dots, x_n|y)}{P(x_1, \dots, x_n)}$$

Using the naive conditional independence assumption that

$$P(x_i|y, x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n) = P(x_i|y)$$

for all  $i$ , this relationship is simplified to

$$P(y|x_1, \dots, x_n) = \frac{P(y) \prod_{i=1}^n P(x_i|y)}{P(x_1, \dots, x_n)}$$

Since  $P(x_1, \dots, x_n)$  is constant given the input, we can use the following classification rule:

$$P(y|x_1, \dots, x_n) \propto P(y) \prod_{i=1}^n P(x_i|y)$$

$$\hat{y} = \arg \max_y P(y) \prod_{i=1}^n P(x_i|y)$$

and we can use Maximum A Posteriori (MAP) estimation to estimate  $P(y)$  and  $P(x_i|y)$ ; the former is then the relative frequency of class  $y$  in the training set. The different naive Bayes classifiers differ mainly by the assumptions they make regarding the distribution of  $P(x_i|y)$ .

Sci-kit learn comes with following naïve bayes classifiers:

- Gaussian Naive Bayes: `GaussianNB()`
- Multinomial Naive Bayes: `MultinomialNB()`
- Complement Naive Bayes: `ComplementNB()`
- Bernoulli Naive Bayes: `BernoulliNB()`
- Categorical Naive Bayes: `CategoricalNB()`

Following are methods callable on a NB classifier model:

<code>fit(X, y[, sample_weight])</code>	Fit Gaussian Naive Bayes according to X, y.
<code>get_params([deep])</code>	Get parameters for this estimator.
<code>partial_fit(X, y[, classes, sample_weight])</code>	Incremental fit on a batch of samples.
<code>predict(X)</code>	Perform classification on an array of test vectors X.
<code>predict_joint_log_proba(X)</code>	Return joint log probability estimates for the test vector X.
<code>predict_log_proba(X)</code>	Return log-probability estimates for the test vector X.
<code>predict_proba(X)</code>	Return probability estimates for the test vector X.
<code>score(X, y[, sample_weight])</code>	Return the mean accuracy on the given test data and labels.
<code>set_params(**params)</code>	Set the parameters of this estimator.

Datasets: Iris, .

Program:

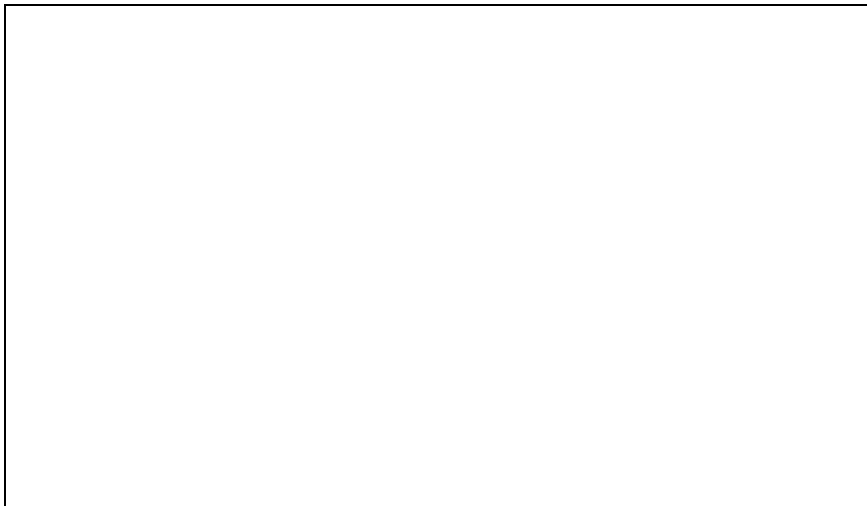
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn import metrics

df=pd.read_csv("_____") # Dataset Name
X,Y= df.iloc[:, 0:-1],df["Species"]
X_train, X_test, Y_train, Y_test = train_test_split( X, Y,
test_size=0.30)
gnb_model = GaussianNB()
gnb_model.fit(X_train, Y_train)
Y_pred_class=gnb_model.predict(X_test)
Y_pred_prob=gnb_model.predict_proba(X_test)
print((Y_pred_class==Y_test).sum()/Y_test.shape[0])
print(gnb_model.score(X_test,Y_test))
print(metrics.accuracy_score(Y_test,Y_pred_class))
print(metrics.confusion_matrix(Y_test,Y_pred_class))
```

Output (Iris):



Output (\_\_\_\_\_):



## Exp 4: Implement a Logistic Regression algorithm for binary classification using Python Programming.

**Aim:** To model a binary classification problem using the logistci regression.

### The Logistic Regression:

The logistic regression is implemented in LogisticRegression. Despite its name, it is implemented as a linear model for classification rather than regression in terms of the scikit-learn/ML nomenclature. The logistic regression is also known in the literature as logit regression, maximum-entropy classification (MaxEnt) or the log-linear classifier. In this model, the probabilities describing the possible outcomes of a single trial are modeled using a logistic function.

For notational ease, we assume that the target  $y_i$  takes values in the set  $\{0,1\}$  for data point  $i$ . Once fitted, the predict\_proba method of LogisticRegression predicts the probability of the positive class  $P(y_i = 1|X_i)$  as

$$P(y_i = 1|X_i) = \hat{P}(X_i) = \frac{1}{1 + \exp(-X_i w - w_0)}$$

Syntax: `sklearn.linear_model.LogisticRegression(penalty='l2', fit_intercept=True, solver='lbfgs', max_iter=100, multi_class='auto', ... ..)`

Parameters:

- `penalty`{‘l1’, ‘l2’, ‘elasticnet’, None}, default=‘l2’
- `fit_intercept`bool, default=True : Specifies if a constant (a.k.a. bias or intercept) should be added to the decision function.
- `solver`{‘lbfgs’, ‘liblinear’, ‘newton-cg’, ‘newton-cholesky’, ‘sag’, ‘saga’}, default=‘lbfgs’: Algorithm to use in the optimization problem.
- `multi_class`{‘auto’, ‘ovr’, ‘multinomial’}, default=‘auto’

The following are methods callable on an LR classifier model:

<code>decision_function(X)</code>	Predict confidence scores for samples.
<code>densify()</code>	Convert coefficient matrix to dense array format.
<code>fit(X, y[, sample_weight])</code>	Fit the model according to the given training data.
<code>get_params([deep])</code>	Get parameters for this estimator.
<code>predict(X)</code>	Predict class labels for samples in X.
<code>predict_log_proba(X)</code>	Predict logarithm of probability estimates.
<code>predict_proba(X)</code>	Probability estimates.
<code>score(X, y[, sample_weight])</code>	Return the mean accuracy on the given test data and labels.
<code>set_params(**params)</code>	Set the parameters of this estimator.
<code>sparsify()</code>	Convert coefficient matrix to sparse format.

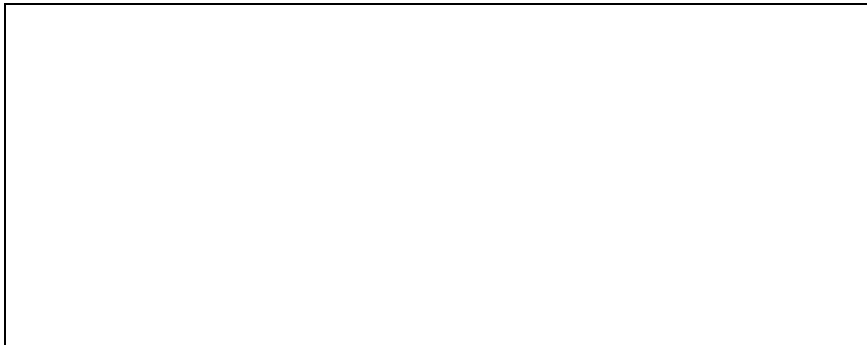
Datasets: vote, .

Program:

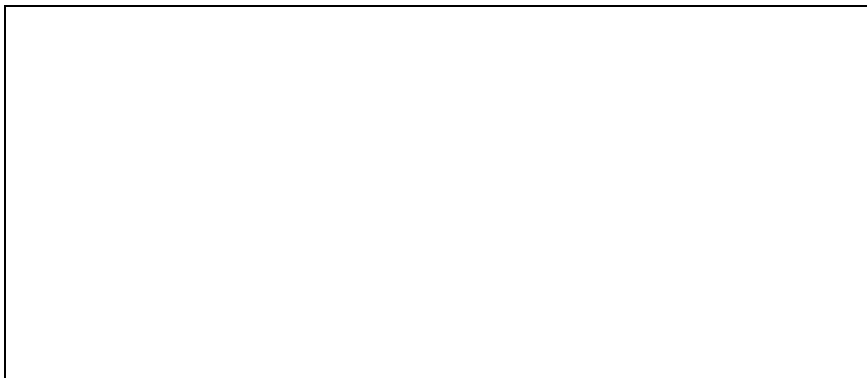
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn import metrics
from sklearn import preprocessing

df=pd.read_excel("vote.xlsx")
print(df.columns)
X,Y= df.iloc[:, 0:-1],df["Class"]
enc = preprocessing.OrdinalEncoder()
X=enc.fit_transform(X)
X_train, X_test, Y_train, Y_test = train_test_split( X, Y,
test_size=0.30)
LR_model = LogisticRegression()
LR_model.fit(X_train, Y_train)
Y_pred_class=LR_model.predict(X_test)
Y_pred_prob=LR_model.predict_proba(X_test)
print((Y_pred_class==Y_test).sum()/Y_test.shape[0])
print(LR_model.score(X_test,Y_test))
print(metrics.accuracy_score(Y_test,Y_pred_class))
print(metrics.confusion_matrix(Y_test,Y_pred_class))
print(Y_pred_prob[10],Y_pred_class[0])
```

Output (Vote):



Output (\_\_\_\_\_):





### Exp 5: Implement the KNN algorithm for classification and demonstrate the process of finding out the optimal "K" value using Python Programming.

**Aim:** To model a classification problem using KNN algorithm and to demonstrate the tuning process of K.

#### K Nearest Neighbour Classifier:

Neighbors-based classification is a type of instance-based learning or non-generalizing learning: it does not attempt to construct a general internal model but simply stores training data instances. Classification is computed from a simple majority vote of the nearest neighbors of each point: a query point is assigned the data class which has the most representatives within the nearest neighbors of the point.

The K-neighbors classification in `KNeighborsClassifier` is the most commonly used technique. The optimal choice of the value K is highly data-dependent: in general, a larger K suppresses the effects of noise but makes the classification boundaries less distinct.

Syntax: `sklearn.neighbors.KNeighborsClassifier(n_neighbors=5, weights='uniform', algorithm='auto', p=2, ... ..)`

Parameters:

- `n_neighbors`int, default=5
- `weights`{‘uniform’, ‘distance’}, callable or None, default=‘uniform’
- `algorithm`{‘auto’, ‘ball\_tree’, ‘kd\_tree’, ‘brute’}, default=‘auto’: Algorithm used to compute the nearest neighbors.
- `p`int, default=2: Power parameter for the Minkowski metric.

The following are methods callable on a KNN classifier model:

<code>fit(X, y)</code>	Fit the k-nearest neighbors classifier from the training dataset.
<code>get_params([deep])</code>	Get parameters for this estimator.
<code>kneighbors([X, n_neighbors, return_distance])</code>	Find the K-neighbors of a point.
<code>kneighbors_graph([X, n_neighbors, mode])</code>	Compute the (weighted) graph of k-Neighbors for points in X.
<code>predict(X)</code>	Predict the class labels for the provided data.
<code>predict_proba(X)</code>	Return probability estimates for the test data X.
<code>score(X, y[, sample_weight])</code>	Return the mean accuracy on the given test data and labels.
<code>set_params(**params)</code>	Set the parameters of this estimator.

Datasets: wine, .

Program:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn import metrics

df=pd.read_csv("wine.csv")
print(df.columns)
print(df.shape)

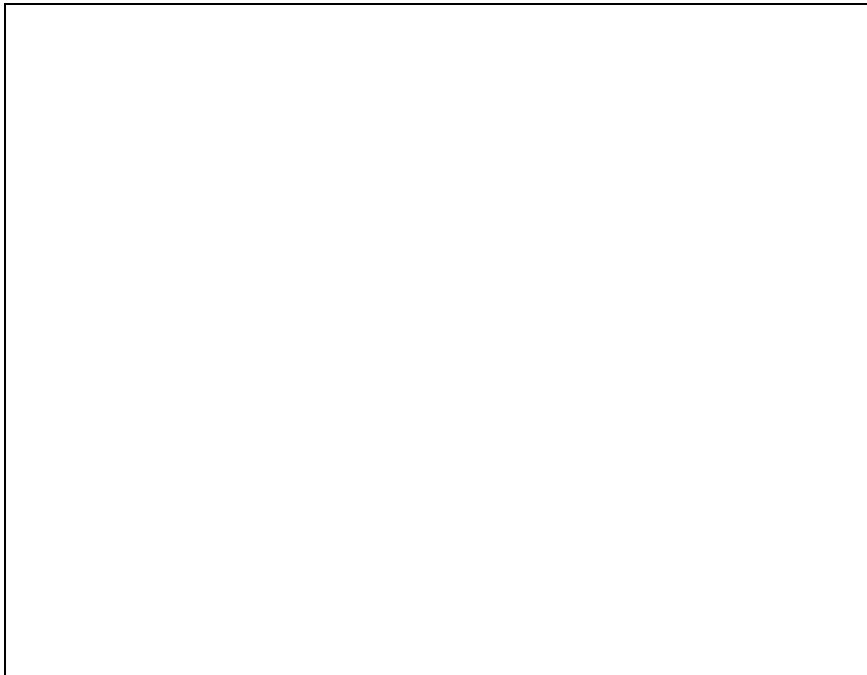
X,Y= df.iloc[:, 1:],df.iloc[:,0]
X_train, X_test, Y_train, Y_test = train_test_split( X, Y, test_size=0.20)

acc=[]
for k in range(1,16):
    temp_model = KNeighborsClassifier(n_neighbors=k)
    temp_model.fit(X_train, Y_train)
    acc.append(temp_model.score(X_test, Y_test))

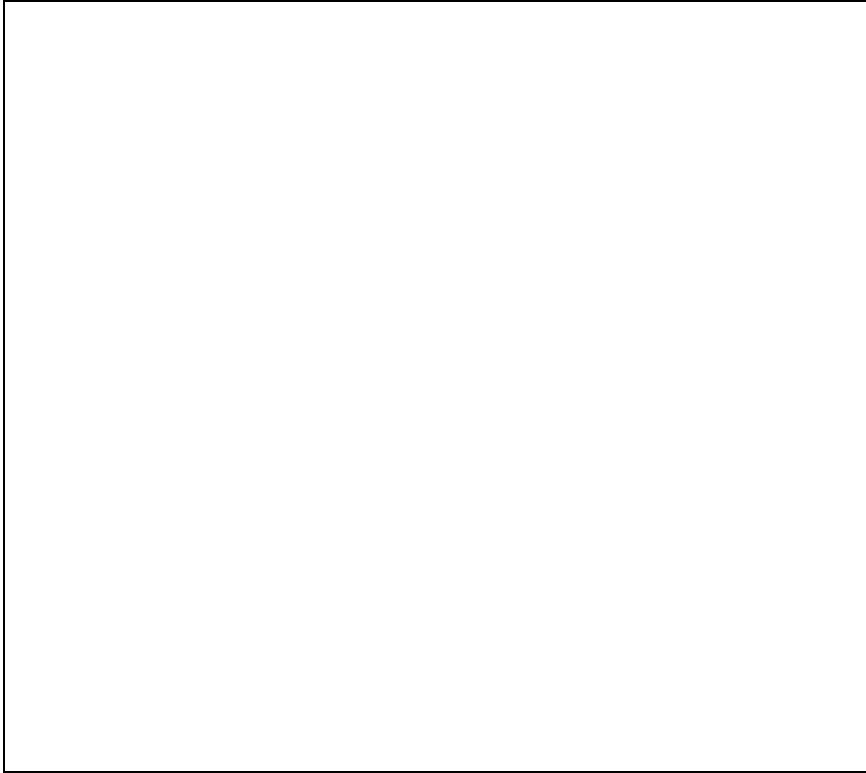
plt.plot(range(1,16),acc)
KNN_model = KNeighborsClassifier(n_neighbors=<K for Max Acc>)
KNN_model.fit(X_train,Y_train)
Y_pred=KNN_model.predict(X_test)

print(metrics.accuracy_score(Y_test,Y_pred))
print(metrics.confusion_matrix(Y_test,Y_pred))
```

Output (Wine):



Output (\_\_\_\_\_):



## Exp 6: Construct an SVM classifier using python programming.

**Aim:** To model a classification problem using the support vector machines (SVM) algorithm.

### Support Vector Machines (SVM):

SVM is a linear classifier that was initially implemented for binary classification that has widely been used in a wide range of ML applications. The standard SVM uses a linear decision boundary, i.e., a hyperplane, given by  $w^T x_{new} + b$ , to classify a new data object.

The advantages of support vector machines are:

- Effective in high-dimensional spaces.
- Still effective in cases where the number of dimensions is greater than the number of samples.
- Uses a subset of training points in the decision function (called support vectors), so it is also memory efficient.
- Versatile: different Kernel functions can be specified for the decision function. Common kernels are provided, but it is also possible to specify custom kernels.

The disadvantage of support vector machines include:

If the number of features is much greater than the number of samples, avoiding over-fitting in choosing Kernel functions and regularization term is crucial.

Syntax: `sklearn.svm.SVC(*, C=1.0, kernel='rbf', degree=3, gamma='scale', coef0=0.0, decision_function_shape='ovr',.....)`

Parameters:

- C: Regularization parameter. The strength of the regularization is inversely proportional to C. It Must be strictly positive.
- kernel: {'linear', 'poly', 'rbf', 'sigmoid', 'precomputed'} or callable: Specifies the kernel type to be used in the algorithm.
- degree:int, default=3: Degree of the polynomial kernel function ('poly'). Must be non-negative.
- gamma: {'scale', 'auto'} or float, default='scale': Kernel coefficient for 'rbf', 'poly' and 'sigmoid'.
- coef0:float, default=0.0: Independent term in kernel function. It is only significant in 'poly' and 'sigmoid'.

The following are methods callable on a SVM classifier model:

<code>decision_function(X)</code>	Evaluate the decision function for the samples in X.
<code>fit(X, y[, sample_weight])</code>	Fit the SVM model according to the given training data.
<code>get_params([deep])</code>	Get parameters for this estimator.
<code>predict(X)</code>	Perform classification on samples in X.
<code>predict_log_proba(X)</code>	Compute log probabilities of possible outcomes for samples in X.
<code>predict_proba(X)</code>	Compute probabilities of possible outcomes for samples in X.
<code>score(X, y[, sample_weight])</code>	Return the mean accuracy on the given test data and labels.
<code>set_params(**params)</code>	Set the parameters of this estimator.

Datasets: vote, .

Program:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
from sklearn import svm
from sklearn import metrics
from sklearn import preprocessing

df=pd.read_excel("vote.xlsx")
print(df.columns)
X,Y= df.iloc[:, 0:-1],df["Class"]
enc = preprocessing.OrdinalEncoder()
X=enc.fit_transform(X)
X_train, X_test, Y_train, Y_test = train_test_split( X, Y, test_size=0.30)
SVC_model = svm.SVC()
SVC_model.fit(X_train, Y_train)
Y_pred=SVC_model.predict(X_test)
print(metrics.accuracy_score(Y_test,Y_pred))
print(metrics.confusion_matrix(Y_test,Y_pred))
```

Output (Vote):



Output ( )::



## **Exp 7: Demonstrate the process of the Decision Tree construction for classification problems using python programming.**

**Aim:** To construct a decision tree for the given dataset using Python.

### **Decision Tree:**

Decision Trees (DTs) are a non-parametric supervised learning method used for classification and regression. The goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features. A tree can be seen as a piecewise constant approximation.

Some advantages of decision trees are:

- Simple to understand and to interpret. Trees can be visualized.
- Requires little data preparation.
- The cost of using the tree (i.e., predicting data) is logarithmic in the number of data points used to train the tree.
- Able to handle both numerical and categorical data. However, the scikit-learn implementation does not support categorical variables for now.
- Able to handle multi-output problems.
- Uses a white box model. If a given situation is observable in a model, the explanation for the condition is easily explained by boolean logic.

The disadvantages of decision trees include:

- Decision-tree learners can create over-complex trees that do not generalize the data well.
- Decision trees can be unstable because small variations in the data might result in a completely different tree being generated.
- Decision tree learners create biased trees if some classes dominate. It is therefore recommended to balance the dataset prior to fitting with the decision tree.

Syntax: `sklearn.tree.DecisionTreeClassifier(*, criterion='gini', splitter='best', max_depth=None, min_samples_split=2, min_samples_leaf=1,... ..)`

Parameters:

- `criterion`: {"gini", "entropy", "log\_loss"}, default="gini".
- `Splitter`: {"best", "random"}, default="best": The strategy used to choose the split at each node. Supported strategies are "best" to choose the best split and "random" to choose the best random split.
- `max_depth`: int, default=None: The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than `min_samples_split` samples.
- `min_samples_split`: int or float, default=2.
- `min_samples_leaf`: int or float, default=1: The minimum number of samples required to be at a leaf node.

The following are methods callable on a DT classifier model:

<code>apply(X[, check_input])</code>	Return the index of the leaf that each sample is predicted as.
<code>cost_complexity_pruning_path(X, y[, ...])</code>	Compute the pruning path during Minimal Cost-Complexity Pruning.
<code>decision_path(X[, check_input])</code>	Return the decision path in the tree.
<code>fit(X, y[, sample_weight, check_input])</code>	Build a decision tree classifier from the training set (X, y).
<code>get_depth()</code>	Return the depth of the decision tree.
<code>get_n_leaves()</code>	Return the number of leaves of the decision tree.
<code>get_params([deep])</code>	Get parameters for this estimator.
<code>predict(X[, check_input])</code>	Predict class or regression value for X.
<code>predict_log_proba(X)</code>	Predict class log-probabilities of the input samples X.
<code>predict_proba(X[, check_input])</code>	Predict class probabilities of the input samples X.
<code>score(X, y[, sample_weight])</code>	Return the mean accuracy on the given test data and labels.
<code>set_params(**params)</code>	Set the parameters of this estimator.

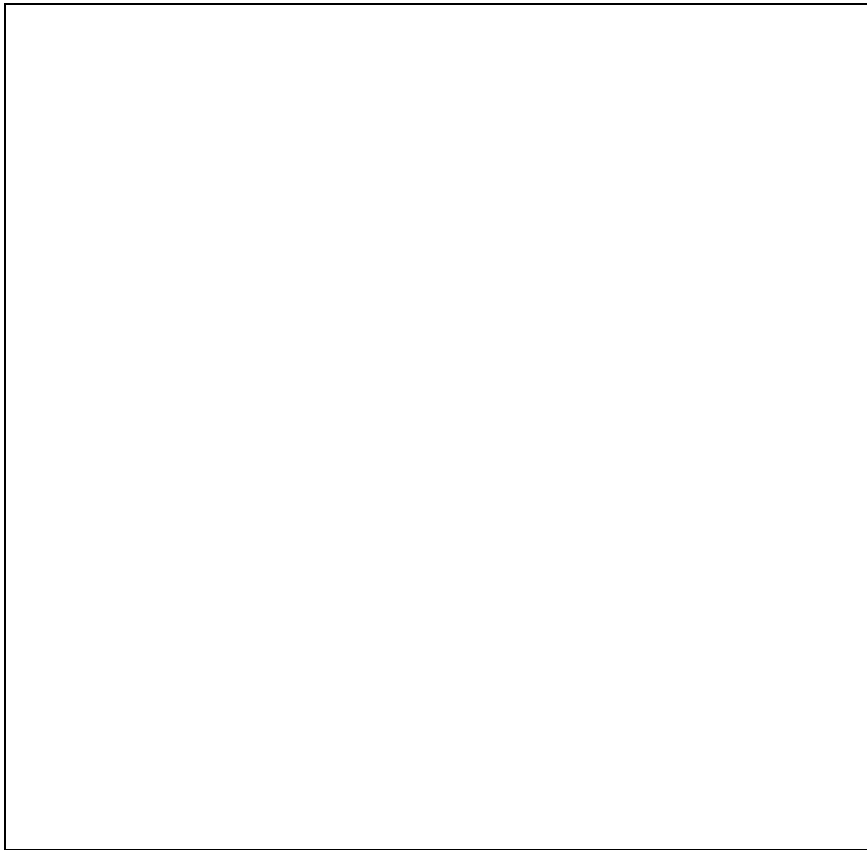
Datasets: baloon, .

Program:

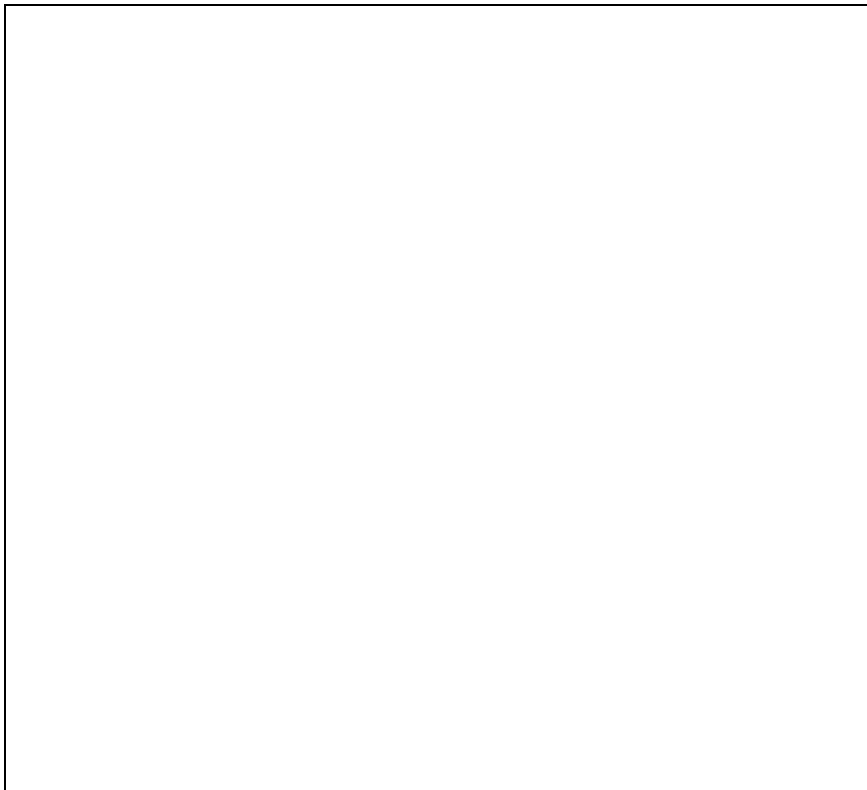
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn import metrics
from sklearn import preprocessing
from sklearn import tree

df=pd.read_csv("baloon.csv")
print(df.columns)
X,Y= df.iloc[:, 0:-1],df["inflated"]
enc = preprocessing.OrdinalEncoder()
X=enc.fit_transform(X)
X_train, X_test, Y_train, Y_test = train_test_split( X, Y, test_size=0.10)
DT_model = DecisionTreeClassifier()
DT_model.fit(X_train, Y_train)
Y_pred=DT_model.predict(X_test)
print(metrics.accuracy_score(Y_test,Y_pred))
print(metrics.confusion_matrix(Y_test,Y_pred))
tree.plot_tree(DT_model)
```

Output (Balloon):



Output (\_\_\_\_\_):





## Exp 8: Implement an Ensemble Learner using Random Forest Algorithm using python programming.

**Aim:** To construct an ensemble learner using the random forest algorithm for the given dataset using Python.

### Ensemble Learning:

In ensemble algorithms, bagging methods form a class of algorithms which build several instances of a black-box estimator on random subsets of the original training set and then aggregate their individual predictions to form a final prediction. These methods are used as a way to reduce the variance of a base estimator (e.g., a decision tree), by introducing randomization into its construction procedure and then making an ensemble out of it.

### Random Forest:

In random forests, each tree in the ensemble is built from a sample drawn with replacement (i.e., a bootstrap sample) from the training set. Furthermore, when splitting each node during the construction of a tree, the best split is found either from all input features or a random subset of size `max_features`.

The purpose of these two sources of randomness is to decrease the variance of the forest estimator. Indeed, individual decision trees typically exhibit high variance and tend to overfit. The injected randomness in forests yield decision trees with somewhat decoupled prediction errors. By taking an average of those predictions, some errors can cancel out. Random forests achieve a reduced variance by combining diverse trees, sometimes at the cost of a slight increase in bias. In practice the variance reduction is often significant hence yielding an overall better model.

Syntax: `sklearn.ensemble.RandomForestClassifier(n_estimators=100, bootstrap=True, oob_score=False, n_jobs=None, ... ..)`

Parameters:

- `n_estimators`: int, default=100: The number of trees in the forest.
- `Bootstrap`: bool, default=True: whether bootstrap samples are used when building trees. If False, the whole dataset is used to build each tree.
- `oob_score`: bool, default=False: whether to use out-of-bag samples to estimate the generalization score. Only available if `bootstrap=True`.
- `n_jobs`: int, default=None: The number of jobs to run in parallel.

The following are methods callable on a `RandomForestClassifier`:

<code>apply(X)</code>	Apply trees in the forest to X, return leaf indices.
<code>decision_path(X)</code>	Return the decision path in the forest.
<code>fit(X, y[, sample_weight])</code>	Build a forest of trees from the training set (X, y).
<code>get_params([deep])</code>	Get parameters for this estimator.
<code>predict(X)</code>	Predict class for X.
<code>predict_log_proba(X)</code>	Predict class log-probabilities for X.

<code>predict_proba(X)</code>	Predict class probabilities for X.
<code>score(X, y[, sample_weight])</code>	Return the mean accuracy on the given test data and labels.
<code>set_params(**params)</code>	Set the parameters of this estimator.

Datasets: iris, .

Program:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn import metrics
from sklearn import preprocessing

df=pd.read_csv("iris.csv")
print(df.columns)
X,Y= df.iloc[:, 0:-1],df["Species"]
X_train, X_test, Y_train, Y_test = train_test_split( X, Y, test_size=0.10)
RF_model = RandomForestClassifier(n_estimators=4)
RF_model.fit(X_train, Y_train)
Y_pred=RF_model.predict(X_test)
print(metrics.accuracy_score(Y_test,Y_pred))
print(metrics.confusion_matrix(Y_test,Y_pred))
```

Output (iris):

Output (\_\_\_\_\_):

## Exp 9: Implement an Ensemble Learner using Adaboost Algorithm using Python programming.

**Aim:** To construct an ensemble learner using the Adaboost algorithm for the given dataset using Python.

### Ensemble Learning:

In ensemble algorithms, bagging methods form a class of algorithms which build several instances of a black-box estimator on random subsets of the original training set and then aggregate their individual predictions to form a final prediction. These methods are used as a way to reduce the variance of a base estimator (e.g., a decision tree), by introducing randomization into its construction procedure and then making an ensemble out of it.

### Adaboost:

The core principle of AdaBoost is to fit a sequence of weak learners (i.e., models that are only slightly better than random guessing, such as small decision trees) on repeatedly modified versions of the data. The predictions from all of them are then combined through a weighted majority vote (or sum) to produce the final prediction. The data modifications at each so-called boosting iteration consist of applying weights  $w_1, w_2, \dots, w_N$  to each of the training samples. Initially, those weights are all set to  $w_i = 1/N$ , so that the first step simply trains a weak learner on the original data. For each successive iteration, the sample weights are individually modified and the learning algorithm is reapplied to the reweighted data. At a given step, those training examples that were incorrectly predicted by the boosted model induced at the previous step have their weights increased, whereas the weights are decreased for those that were predicted correctly. As iterations proceed, examples that are difficult to predict receive ever-increasing influence. Each subsequent weak learner is thereby forced to concentrate on the examples that are missed by the previous ones in the sequence

Syntax: `class sklearn.ensemble.AdaBoostClassifier(base_estimator=None, *, n_estimators=50, learning_rate=1.0, algorithm='SAMME.R',... ..)`

Parameters:

- **base\_estimator:** object, default=None: The base estimator from which the boosted ensemble is built. If None, then the base estimator is `DecisionTreeClassifier` initialized with `max_depth=1`.
- **n\_estimators:** int, default=50: The number of estimators.
- **learning\_rate:** float, default=1.0: Weight applied to each classifier at each boosting iteration. A higher learning rate increases the contribution of each classifier.
- **algorithm:** {'SAMME', 'SAMME.R'}, default='SAMME.R' If 'SAMME.R' then use the SAMME.R real boosting algorithm.

The following are methods callable on a `RandomForestClassifier`:

<code>decision_function(X)</code>	Compute the decision function of X.
<code>fit(X, y[, sample_weight])</code>	Build a boosted classifier/regressor from the training set (X, y).
<code>get_params([deep])</code>	Get parameters for this estimator.

predict(X)	Predict classes for X.
predict_log_proba(X)	Predict class log-probabilities for X.
predict_proba(X)	Predict class probabilities for X.
score(X, y[, sample_weight])	Return the mean accuracy on the given test data and labels.
set_params(**params)	Set the parameters of this estimator.
staged_decision_function(X)	Compute decision function of X for each boosting iteration.
staged_predict(X)	Return staged predictions for X.
staged_predict_proba(X)	Predict class probabilities for X.
staged_score(X, y[, sample_weight])	Return staged scores for X, y.

Datasets: iris, .

Program:

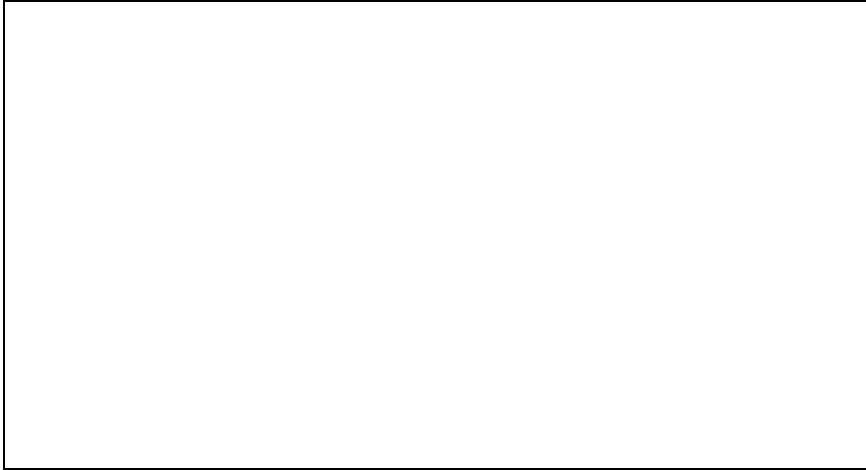
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
from sklearn.ensemble import AdaBoostClassifier
from sklearn import metrics
from sklearn import preprocessing
from sklearn.naive_bayes import GaussianNB

df=pd.read_csv("iris.csv")
print(df.columns)
X,Y= df.iloc[:, 0:-1],df["Species"]
X_train, X_test, Y_train, Y_test = train_test_split( X, Y, test_size=0.10)
est=GaussianNB()
AB_model = AdaBoostClassifier(base_estimator=est,n_estimators=5)
AB_model.fit(X_train, Y_train)
AB_pred=DT_model.predict(X_test)
print(metrics.accuracy_score(Y_test,Y_pred))
print(metrics.confusion_matrix(Y_test,Y_pred))
```

Output (iris):



Output (\_\_\_\_\_):

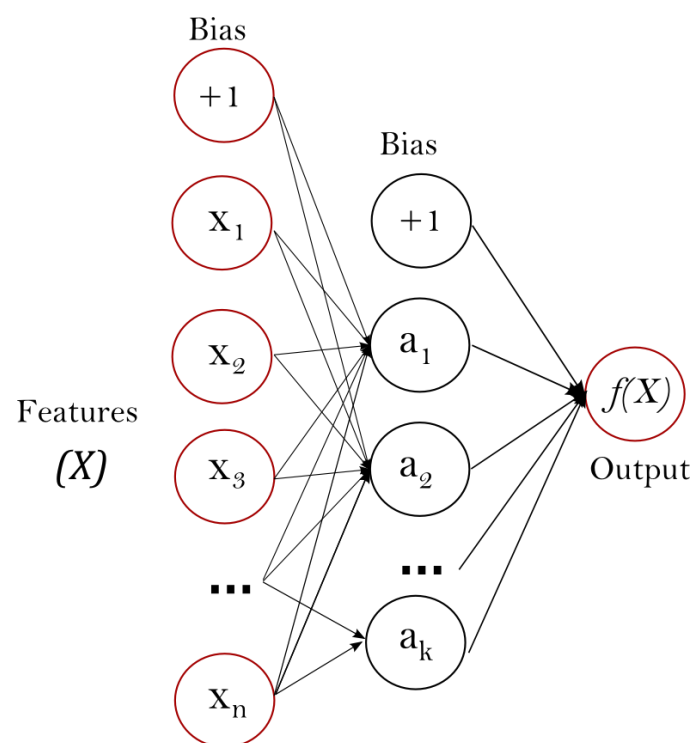


**Exp 10: Demonstrate the working of Multi-layer perceptron with MLPClassifier() using Python programming.**

**Aim:** To construct a Multi-Layer Perceptron for the given dataset using Python.

**Multi-Layer Perceptron:**

Multi-layer Perceptron (MLP) is a supervised learning algorithm that learns a function  $f(.): R^m \rightarrow R^o$  by training on a dataset, where  $m$  is the number of dimensions for input and  $o$  is the number of dimensions for output. Given a set of features  $X = x_1, x_2, \dots, x_m$  and a target  $y$ , it can learn a non-linear function approximator for either classification or regression. It is different from logistic regression, in that between the input and the output layer, there can be one or more non-linear layers, called hidden layers. Following figure shows a one hidden layer MLP with scalar output.



The advantages of Multi-layer Perceptron are:

- Capability to learn non-linear models.
- Capability to learn models in real-time (on-line learning) using `partial_fit`.

The disadvantages of Multi-layer Perceptron (MLP) include:

- MLP with hidden layers have a non-convex loss function where there exists more than one local minimum. Therefore different random weight initializations can lead to different validation accuracy.
- MLP requires tuning a number of hyperparameters such as the number of hidden neurons, layers, and iterations.
- MLP is sensitive to feature scaling.

Syntax: `class sklearn.neural_network.MLPClassifier(hidden_layer_sizes=(100,), activation='relu', solver='adam', batch_size='auto', learning_rate='constant', learning_rate_init=0.001, max_iter=200, tol=0.0001,... ...)`

Parameters:

- `hidden_layer_sizes`: array-like of shape(n\_layers - 2,), default=(100,): The ith element represents the number of neurons in the ith hidden layer.
- `activation`: {'identity', 'logistic', 'tanh', 'relu'}, default='relu': Activation function for the hidden layer.
- `solver`: {'lbfgs', 'sgd', 'adam'}, default='adam': The solver for weight optimization.
- `batch_size`: int, default='auto': Size of minibatches for stochastic optimizers. If the solver is 'lbfgs', the classifier will not use minibatch. When set to "auto", `batch_size=min(200, n_samples)`
- `learning_rate`: {'constant', 'invscaling', 'adaptive'}, default='constant': Learning rate schedule for weight updates.
- `learning_rate_init`: float, default=0.001, The initial learning rate used. It controls the step-size in updating the weights. Only used when solver='sgd' or 'adam'.
- `max_iter`: int, default=200, Maximum number of iterations.
- `tol`: float, default=1e-4, Tolerance for the optimization. When the loss or score is not improving by at least tol for `n_iter_no_change` consecutive iterations

The following are methods callable on a `RandomForestClassifier`:

<code>fit(X, y)</code>	Fit the model to data matrix X and target(s) y.
<code>get_params([deep])</code>	Get parameters for this estimator.
<code>partial_fit(X, y[, classes])</code>	Update the model with a single iteration over the given data.
<code>predict(X)</code>	Predict using the multi-layer perceptron classifier.
<code>predict_log_proba(X)</code>	Return the log of probability estimates.
<code>predict_proba(X)</code>	Probability estimates.
<code>score(X, y[, sample_weight])</code>	Return the mean accuracy on the given test data and labels.
<code>set_params(**params)</code>	Set the parameters of this estimator.

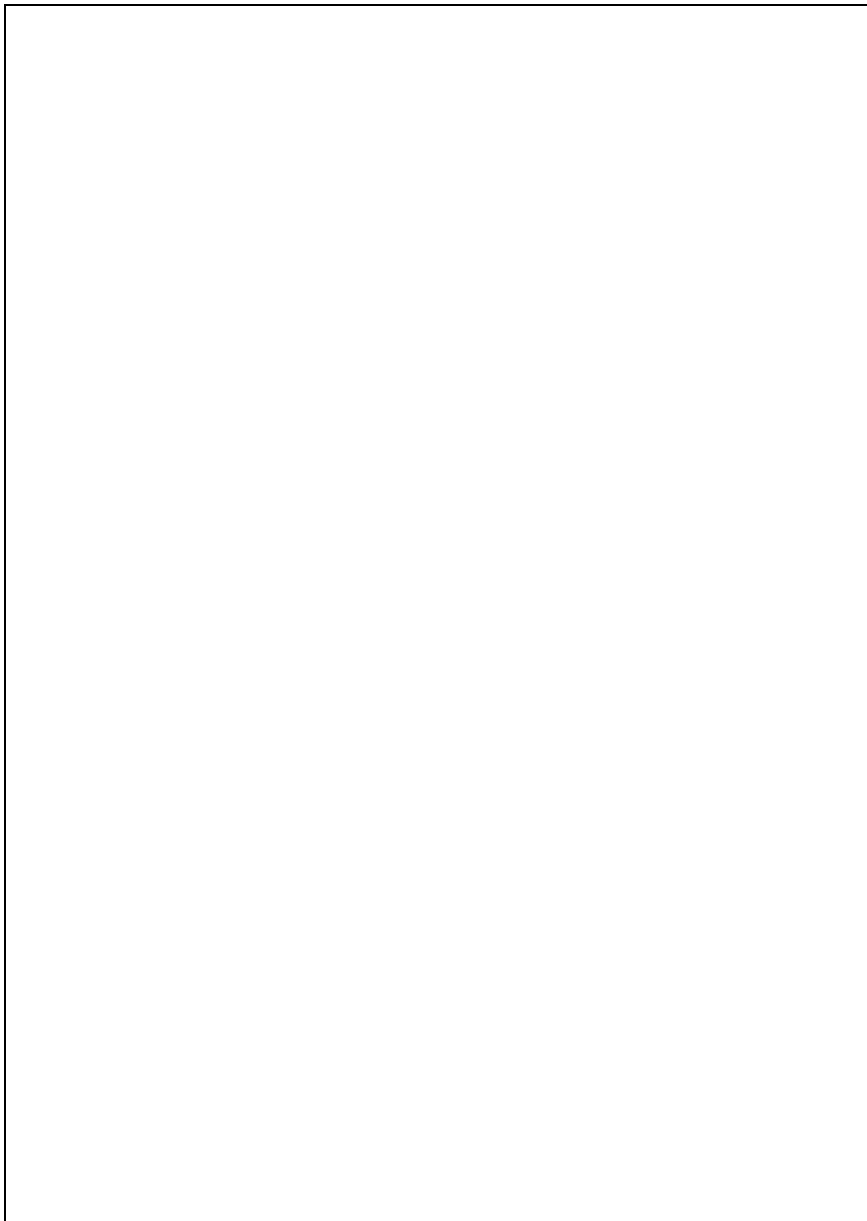
Datasets: vote.

Program:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier
from sklearn import metrics
from sklearn import preprocessing
```

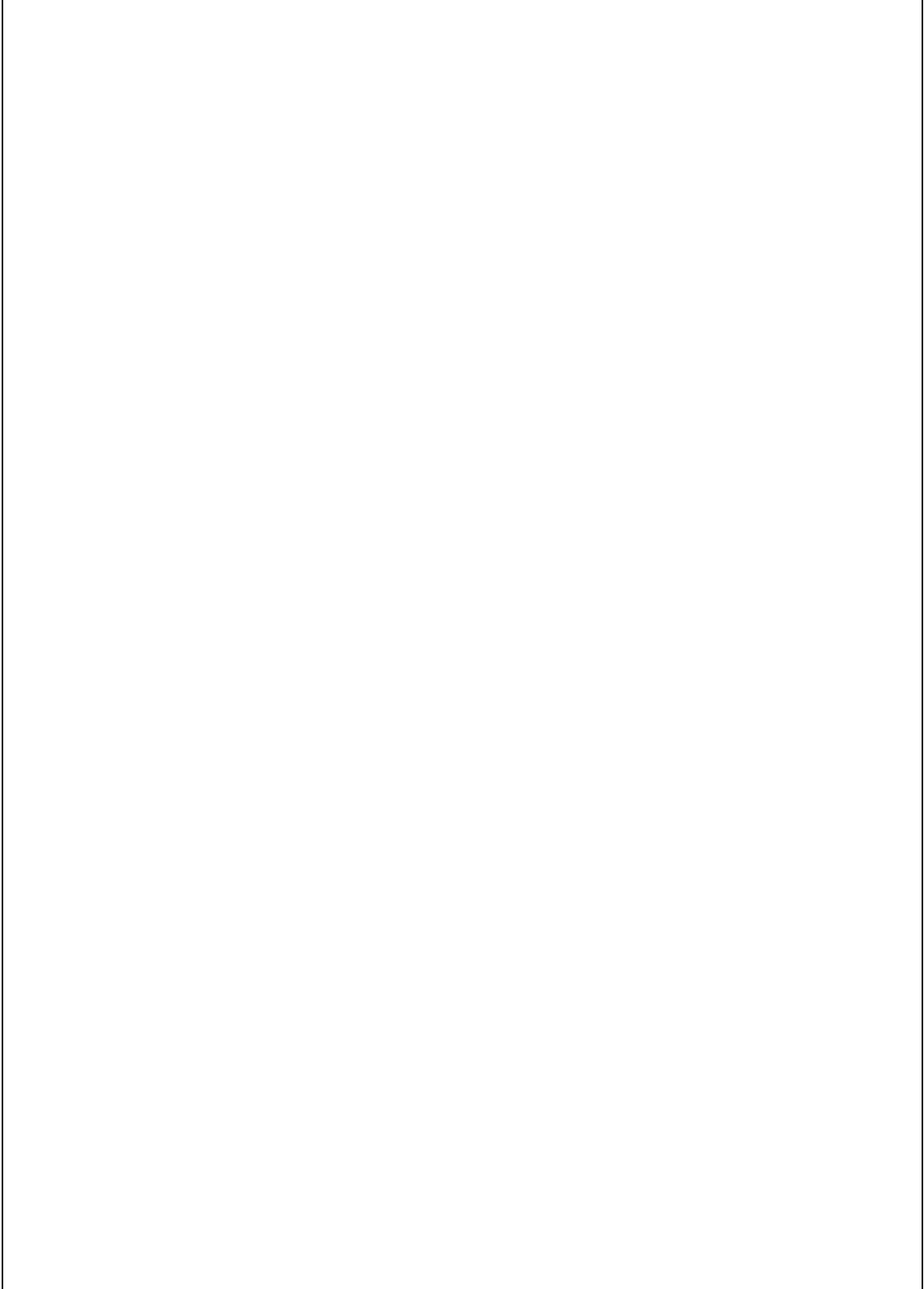
```
df=pd.read_excel("vote.xlsx")
print(df.columns)
X,Y= df.iloc[:, 0:-1],df["Class"]
enc = preprocessing.OrdinalEncoder()
X=enc.fit_transform(X)
X_train, X_test, Y_train, Y_test = train_test_split( X, Y, test_size=0.30)
MLP_model = MLPClassifier(hidden_layer_sizes=(4,))
MLP_model.fit(X_train, Y_train)
Y_pred=MLP_model.predict(X_test)
print(metrics.accuracy_score(Y_test,Y_pred))
print(metrics.confusion_matrix(Y_test,Y_pred))
print(MLP_model.coefs_)
```

Output (vote):





MLP Network (vote):



## Exp 11: Demonstrate the K-Means algorithm for the given data set using Python programming.

**Aim:** To cluster the given data into groups using K-Means Clustering Algorithm.

### Clustering:

Cluster analysis or clustering is the task of grouping a set of objects in such a way that objects in the same group are more similar to each other than to those in other groups.

### K-Means:

The KMeans algorithm clusters data by trying to separate samples in  $n$  groups of equal variance, minimizing a criterion known as the inertia or within-cluster sum-of-squares (see below). This algorithm requires the number of clusters to be specified. It scales well to large numbers of samples and has been used across a large range of application areas in many different fields.

The k-means algorithm divides a set of  $N$  samples  $X$  into  $K$  disjoint clusters  $C$ , each described by the mean  $\mu_j$  of the samples in the cluster. The means are commonly called the cluster “centroids”; note that they are not, in general, points from  $X$ , although they live in the same space.

Syntax: `class sklearn.cluster.KMeans(n_clusters=8, init='k-means++', n_init='auto', max_iter=300, tol=0.0001,... ...)`

Parameters:

- `n_clusters`: int, default=8, The number of clusters to form as well as the number of centroids to generate.
- `init`: {'k-means++', 'random'}, callable or array-like of shape (n\_clusters, n\_features), default='k-means++'.
- `n_init`: 'auto' or int, default=10, Number of times the k-means algorithm is run with different centroid seeds.
- `max_iter`: int, default=300, Maximum number of iterations of the k-means algorithm for a single run.
- `tol`: float, default=1e-4, Relative tolerance with regards to Frobenius norm of the difference in the cluster centers of two consecutive iterations to declare convergence.

The following are methods callable on a `RandomForestClassifier`:

<code>fit(X[, y, sample_weight])</code>	Compute k-means clustering.
<code>fit_predict(X[, y, sample_weight])</code>	Compute cluster centers and predict cluster index for each sample.
<code>fit_transform(X[, y, sample_weight])</code>	Compute clustering and transform X to cluster-distance space.
<code>get_feature_names_out([input_features])</code>	Get output feature names for transformation.
<code>get_params([deep])</code>	Get parameters for this estimator.

<code>predict(X[, sample_weight])</code>	Predict the closest cluster each sample in X belongs to.
<code>score(X[, y, sample_weight])</code>	Opposite of the value of X on the K-means objective.
<code>set_output(*[, transform])</code>	Set output container.
<code>set_params(**params)</code>	Set the parameters of this estimator.
<code>transform(X)</code>	Transform X to a cluster-distance space.

Datasets: iris.

Program:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
from sklearn.cluster import KMeans

df=pd.read_csv("iris.csv")
print(df.head())
X= df.iloc[:, 0:-1]
X_train, X_test = train_test_split( X, test_size=0.30)
KM_model = KMeans(n_clusters=3)
KM_model.fit(X_train)
Clusters=KM_model.predict(X_test)
print(Clusters)
```

Output (iris):

