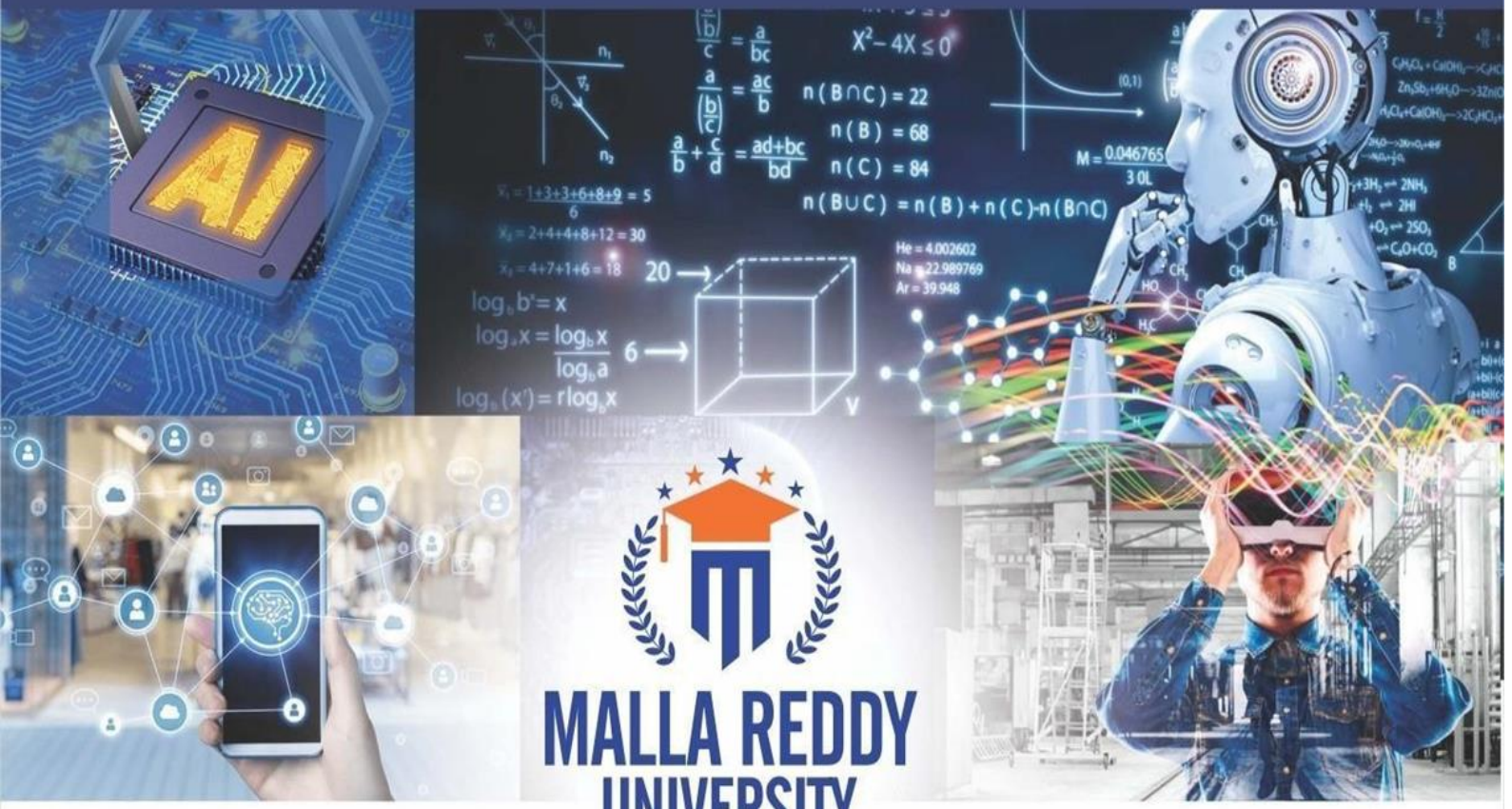
KNOWLEDGE EMPOWERMENT WORLD



MALLA REDDY UNIVERSITY

(Telangana State Private Universities Act No.13 of 2020 and G.O.Ms.No.14, Higher Education (UE) Department)

MACHINE LEARNING

Learning Manual

B.Tech: II Year II Semester

(CSE-AI&ML)

(2024-2025)

School of Engineering

www.mallareddyuniversity.ac.in

Learning Manual

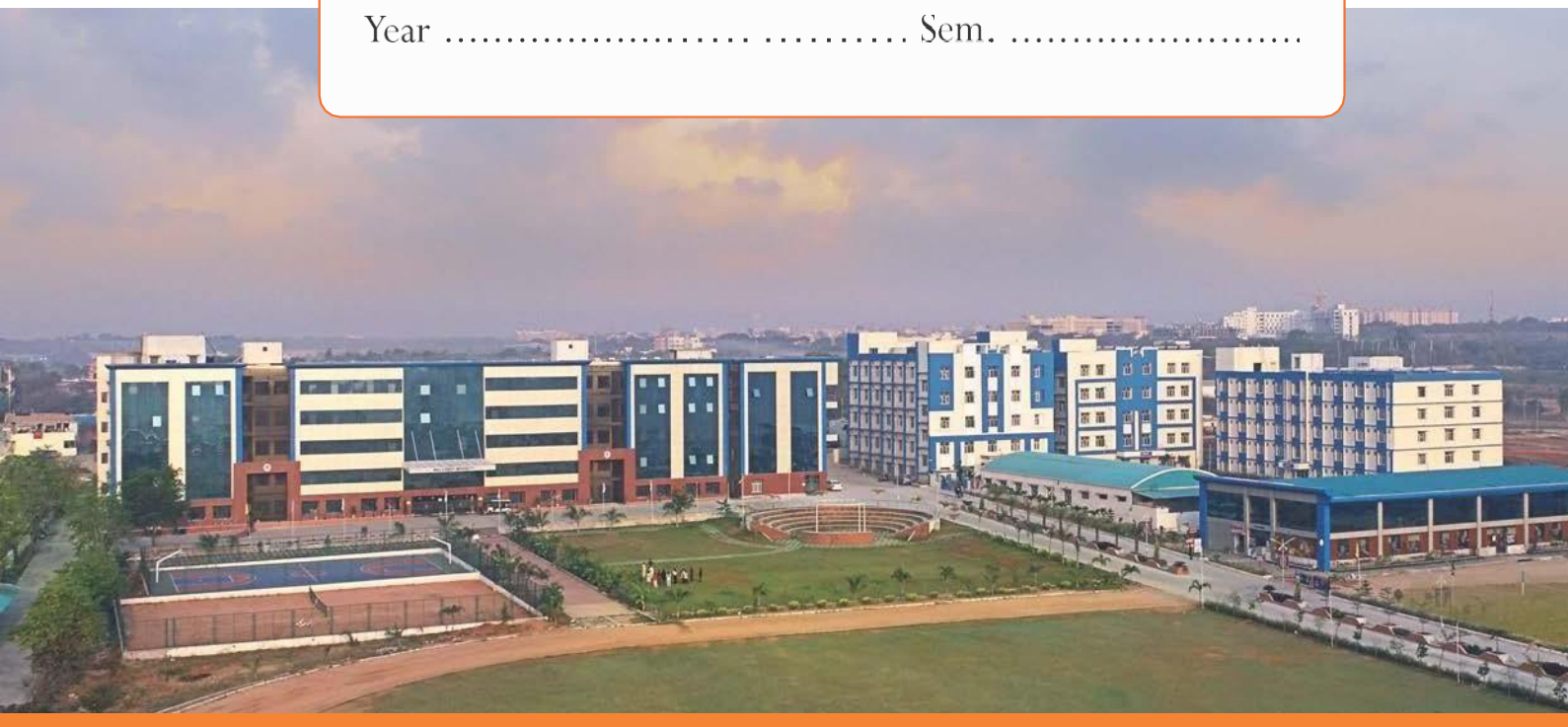
MALLA REDDY UNIVERSITY

(Telangana State Private Universities Act No.13 of 2020 and
G.O.Ms.No.14, Higher Education (UE) Department)

Name .....

Roll No. ..... Branch .....

Year ..... Sem. .....



MALLA REDDY UNIVERSITY
(Telangana State Private Universities Act No.13 of 2020 and
G.O.Ms.No.14, Higher Education (UE) Department)

Maisammaguda, Kompally,
Medchal - Malkajgiri District
Hyderabad - 500100, Telangana State.
mruh@mallareddyuniversity.ac.in
www.mallareddyuniversity.ac.in

Certificate

School of Engineering

Certified that this is the bonafide record of practical work done by

Mr./Ms..... Roll. No..... of

B.Tech ..... year .... Semester for Academic year 20..... - 20..... in

..... Laboratory.

Faculty Incharge

HOD

Dean-AIML

:

External

GENERAL LABORATORY INSTRUCTIONS

1. Students are advised to come to the laboratory at least 5 minutes before (to the starting time), those who come after 5 minutes will not be allowed into the lab.

2. Plan your task properly much before to the commencement, come prepared to the lab with the synopsis / program / experiment details.

3. Student should enter into the laboratory with:
   Laboratory observation notes with all the details (Problem statement, Aim, Algorithm, Procedure, Program, Expected Output, etc.,) filled in for the lab session.

4. Laboratory Record updated up to the last session experiments and other utensils (if any) needed in the lab.

5. Proper Dress code and Identity card.

6. Sign in the laboratory login register, write the TIME-IN, and occupy the computer system allotted to you by the faculty.

7. Execute your task in the laboratory, and record the results / output in the lab observation note book, and get certified by the concerned faculty.

8. All the students should be polite and cooperative with the laboratory staff, must maintain the discipline and decency in the laboratory.

9. Computer labs are established with sophisticated and high end branded systems, which should be utilized properly.

10. Students / Faculty must keep their mobile phones in SWITCHED OFF mode during the lab sessions. Misuse of the equipment, misbehaviors with the staff and systems etc., will attract severe punishment.

11. Students must take the permission of the faculty in case of any urgency to go out; if anybody found loitering outside the lab / class without permission during working hours will be treated seriously and punished appropriately.

12. Students should LOG OFF/ SHUT DOWN the computer system before he/she leaves the lab after completing the task (experiment) in all aspects. He/she must ensure the system / seat is kept properly.

Course Objectives:

- To understand the use of datasets in implementing the machine learning algorithms.
- Implement the machine learning concepts and algorithms in python.
- Evaluate the performance of various machine learning algorithms.

Course Outcomes:

At the end of this course, the students will be able to

- Understand the mathematical and statistical perspectives of machine learning algorithms through python programming.
- Design and evaluate the unsupervised models through python in built functions.
- Evaluate the machine learning models pre-processed through various feature engineering algorithms by python programming.
- Design and apply various reinforcement algorithms to solve real time complex problems.

AI & ML DEPARTMENT (II YEAR II SEMESTER)

MACHINE LEARNING LABORATORY (MR23-1CS0203)

INDEX

1.Data Preprocessing

Machine Learning algorithms are completely dependent on data because it is the most crucial aspect that makes model training possible. On the other hand, if we won't be able to make sense out of that data, before feeding it to ML algorithms, a machine will be useless. In simple words, we always need to feed right data i.e. the data in correct scale, format and containing meaningful features, for the problem we want machine to solve.

This makes data preparation the most important step in ML process. Data preparation may be defined as the procedure that makes our dataset more appropriate for ML process.

Why Data Preprocessing?

After selecting the raw data for ML training, the most important task is data pre-processing. In broad sense, data preprocessing will convert the selected data into a form we can work with or can feed to ML algorithms. We always need to preprocess our data so that it can be as per the expectation of machine learning algorithm.

Data Pre-processing Techniques:

We have the following data preprocessing techniques that can be applied on data set to produce data for ML algorithms −

Scaling

Most probably our dataset comprises of the attributes with varying scale, but we cannot provide such data to ML algorithm hence it requires rescaling. Data rescaling makes sure that attributes are at same scale. Generally, attributes are rescaled into the range of 0 and 1. ML algorithms like gradient descent and k-Nearest Neighbors requires scaled data. We can rescale the data with the help of MinMaxScaler class of scikit-learn Python library.

In this example we will rescale the data of Pima Indians Diabetes dataset which we used

```
from pandas import read_csv
from numpy import set_printoptions
from sklearn import preprocessing
path = r'C:\pima-indians-diabetes.csv'
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = read_csv(path, names=names)
array = dataframe.values
```

earlier. First, the CSV data will be loaded (as done in the previous chapters) and then with the help of MinMaxScaler class, it will be rescaled in the range of 0 and 1.

The first few lines of the following script are same as we have written in previous chapters while loading CSV data.

Now, we can use MinMaxScaler class to rescale the data in the range of 0 and 1.

```
data_scaler =
preprocessing.MinMaxScaler(feature_range=(0,1))
data_rescaled = data_scaler.fit_transform(array)
```

We can also summarize the data for output as per our choice. Here, we are setting the precision to 1 and showing the first 10 rows in the output.

```
set_printoptions(precision=1)
print ("\nScaled data:\n", data_rescaled[0:10])
```

Output

```
Scaled data:
[
[0.4 0.7 0.6 0.4 0. 0.5 0.2 0.5 1. ]
[0.1 0.4 0.5 0.3 0. 0.4 0.1 0.2 0. ]
[0.5 0.9 0.5 0. 0. 0.3 0.3 0.2 1. ]
[0.1 0.4 0.5 0.2 0.1 0.4 0. 0. 0. ]
[0. 0.7 0.3 0.4 0.2 0.6 0.9 0.2 1. ]
[0.3 0.6 0.6 0. 0. 0.4 0.1 0.2 0. ]
[0.2 0.4 0.4 0.3 0.1 0.5 0.1 0.1 1. ]
[0.6 0.6 0. 0. 0. 0.5 0. 0.1 0. ]
[0.1 1. 0.6 0.5 0.6 0.5 0. 0.5 1. ]
[0.5 0.6 0.8 0. 0. 0. 0.1 0.6 1. ]
]
```

From the above output, all the data got rescaled into the range of 0 and 1.

Normalization

Another useful data preprocessing technique is Normalization. This is used to rescale each row of data to have a length of 1. It is mainly useful in Sparse dataset where we have lots of zeros. We can rescale the data with the help of Normalizer class of scikit-learn Python library.

Types of Normalization

In machine learning, there are two types of normalization preprocessing techniques as follows:

L1 Normalization

It may be defined as the normalization technique that modifies the dataset values in a way that in each row the sum of the absolute values will always be up to 1. It is also called Least Absolute Deviations.

In the below example, we use L1 Normalize technique to normalize the data of Pima Indians Diabetes dataset which we used earlier. First, the CSV data will be loaded and then with the help of Normalizer class it will be normalized.

The first few lines of following script are same as we have written in previous chapters while loading CSV data.

```
from pandas import read_csv
from numpy import set_printoptions
from sklearn.preprocessing import Normalizer
```

```
path = r'C:\pima-indians-diabetes.csv'
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = read_csv (path, names=names)
array = dataframe.values
```

Now, we can use Normalizer class with L1 to normalize the data.

```
Data_normalizer =
Normalizer(norm='l1').fit(array) Data_normalized
= Data_normalizer.transform(array)
```

We can also summarize the data for output as per our choice. Here, we are setting the precision to 2 and showing the first 3 rows in the output.

```
set_printoptions(precision=2)
print ("\nNormalized data:\n", Data_normalized [0:3])
```

Output

```
Normalized data:
[
[0.02 0.43 0.21 0.1 0. 0.1 0. 0.14 0. ]
[0. 0.36 0.28 0.12 0. 0.11 0. 0.13 0. ]
[0.03 0.59 0.21 0. 0. 0.07 0. 0.1 0. ]
]
```

L2 Normalization

It may be defined as the normalization technique that modifies the dataset values in a way that in each row the sum of the squares will always be up to 1. It is also called least squares.

In the below example, we use L2 Normalization technique to normalize the data of Pima Indians Diabetes dataset which we used earlier. First, the CSV data will be loaded (as done in previous chapters) and then with the help of Normalizer class it will be normalized.

The first few lines of following script are same as we have written in previous chapters while loading CSV data.

```
from pandas import read_csv
from numpy import set_printoptions
from sklearn.preprocessing import Normalizer
path = r'C:\pima-indians-diabetes.csv'
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = read_csv (path, names=names)
array = dataframe.values
```

Now, we can use Normalizer class with L1 to normalize the data.

```
Data_normalizer =
Normalizer(norm='l2').fit(array) Data_normalized
= Data_normalizer.transform(array)
```

We can also summarize the data for output as per our choice. Here, we are setting the precision to 2 and showing the first 3 rows in the output.

```
set_printoptions(precision=2)
print ("\nNormalized data:\n", Data_normalized [0:3])
```

Output

Normalized data:
[
[0.03 0.83 0.4 0.2 0. 0.19 0. 0.28 0.01]
[0.01 0.72 0.56 0.24 0. 0.22 0. 0.26 0.]
[0.04 0.92 0.32 0. 0. 0.12 0. 0.16 0.01]
]

Binarization

As the name suggests, this is the technique with the help of which we can make our data binary. We can use a binary threshold for making our data binary. The values above that threshold value will be converted to 1 and below that threshold will be converted to 0. For example, if we choose threshold value = 0.5, then the dataset value above it will become 1 and below this will become 0. That is why we can call it **binarizing** the data or **thresholding** the data. This technique is useful when we have probabilities in our dataset and want to convert them into crisp values.

We can binarize the data with the help of Binarizer class of scikit-learn Python library.

In the below example, we will rescale the data of Pima Indians Diabetes dataset which we used earlier. First, the CSV data will be loaded and then with the help of Binarizer class it will be converted into binary values i.e. 0 and 1 depending upon the threshold value. We are taking 0.5 as threshold value.

The first few lines of following script are same as we have written in previous chapters while loading CSV data.

```
from pandas import read_csv
from sklearn.preprocessing import Binarizer
path = r'C:\pima-indians-diabetes.csv'
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = read_csv(path, names=names)
array = dataframe.values
```

Now, we can use Binarize class to convert the data into binary values.

```
binarizer =
Binarizer(threshold=0.5).fit(array)
Data_binarized =
binarizer.transform(array)
```

Here, we are showing the first 5 rows in the output.

```
print ("\nBinary data:\n", Data_binarized [0:5])
```

Output

```
Binary data:
[
[1. 1. 1. 1. 0. 1. 1. 1. 1.]
[1. 1. 1. 1. 0. 1. 0. 1. 0.]
[1. 1. 1. 0. 0. 1. 1. 1. 1.]
[1. 1. 1. 1. 1. 1. 0. 1. 0.]
[0. 1. 1. 1. 1. 1. 1. 1. 1.]
]
```

Standardization

Another useful data preprocessing technique which is basically used to transform the data attributes with a Gaussian distribution. It differs the mean and SD (Standard Deviation) to a standard Gaussian distribution with a mean of 0 and a SD of 1. This technique is useful in ML algorithms like linear regression, logistic regression that assumes a Gaussian distribution in input dataset and produce better results with rescaled data. We can standardize the data (mean
= 0 and SD =1) with the help of StandardScaler class of scikit-learn Python library.

In the below example, we will rescale the data of Pima Indians Diabetes dataset which we used earlier. First, the CSV data will be loaded and then with the help of StandardScaler class it will be converted into Gaussian Distribution with mean = 0 and SD = 1.

The first few lines of following script are same as we have written in previous chapters while loading CSV data.

```
from sklearn.preprocessing import StandardScaler
from pandas import read_csv
from numpy import set_printoptions
path = r'C:\pima-indians-diabetes.csv'
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = read_csv(path, names=names)
array = dataframe.values
```

Now, we can use StandardScaler class to rescale the data.

```
data_scaler = StandardScaler().fit(array)
data_rescaled =
data_scaler.transform(array)
```

We can also summarize the data for output as per our choice. Here, we are setting the precision to 2 and showing the first 5 rows in the output.

```
set_printoptions(precision=2)
print ("\nRescaled data:\n", data_rescaled [0:5])
```

Output

Rescaled data:
[
[0.64 0.85 0.15 0.91 -0.69 0.2 0.47 1.43 1.37]
[-0.84 -1.12 -0.16 0.53 -0.69 -0.68 -0.37 -0.19 -0.73]
[1.23 1.94 -0.26 -1.29 -0.69 -1.1 0.6 -0.11 1.37]
[-0.84 -1. -0.16 0.15 0.12 -0.49 -0.92 -1.04 -0.73]
[-1.14 0.5 -1.5 0.91 0.77 1.41 5.48 -0.02 1.37]
]

Data Labeling

We discussed the importance of good fata for ML algorithms as well as some techniques to pre-process the data before sending it to ML algorithms. One more aspect in this regard is data labeling. It is also very important to send the data to ML algorithms having proper labeling. For example, in case of classification problems, lot of labels in the form of words, numbers etc. are there on the data.

What is Label Encoding?

Most of the sklearn functions expect that the data with number labels rather than word labels. Hence, we need to convert such labels into number labels. This process is called label encoding. We can perform label encoding of data with the help of LabelEncoder() function of scikit-learn Python library.

In the following example, Python script will perform the label encoding.

First, import the required Python libraries as follows −

import numpy as np
from sklearn import preprocessing

Now, we need to provide the input labels as follows −

input_labels =

['red','black','red','green','black','yellow','white'] The next

line of code will create the label encoder and train it.

encoder =
preprocessing.LabelEncoder()
encoder.fit(input_labels)

The next lines of script will check the performance by encoding the random ordered list −

```
test_labels = ['green','red','black']
encoded_values = encoder.transform(test_labels)
print("\nLabels =", test_labels)
print("Encoded values =", list(encoded_values))
encoded_values = [3,0,4,1]
decoded_list = encoder.inverse_transform(encoded_values)
```

We can get the list of encoded values with the help of following python script −

```
print("\nEncoded values =", encoded_values)
print("\nDecoded labels =", list(decoded_list))
```

Output

```
Labels = ['green', 'red',
'black'] Encoded values =
[1, 2, 0]
Encoded values = [3, 0, 4, 1]
Decoded labels = ['white', 'black', 'yellow', 'green']
```

Exercise:1.1: Implement a program to clean a dataset of missing values.

```
import pandas as pd
from sklearn.impute import SimpleImputer
        def generate_example_dataset():
            # Create an example dataset with missing values
            data = {
                'PassengerId': [1, 2, 3, 4, 5],
                'Name': ['John', 'Jane', 'Bob', 'Alice', 'Charlie'],
                'Age': [22, None, 25, None, 30],
                'Fare': [7.25, 71.28, None, 8.05, 10.5],
                'Survived': [0, 1, 1, 0, 1]
            }
            return pd.DataFrame(data) def
        clean_dataset(df):
            # Display the original dataset
            print("Original Dataset:")
            print(df)
            # Drop non-numeric columns
            numeric_df = df.select_dtypes(include='number')

            # Handling missing values using SimpleImputer (mean strategy) imputer
            = SimpleImputer(strategy='mean')
            df_cleaned=pd.DataFrame(imputer.fit_transform(numeric_df),
        columns=numeric_df.columns) #
    Display the cleaned dataset
            print("\nCleaned Dataset:")
```

```
    print(df_cleaned)

if__name__== "_main_":
# Generate an example dataset with missing values
    example_dataset = generate_example_dataset() #
Call the clean_dataset function

    clean_dataset(example_dataset)
```

Output:-

```
Original Dataset:
     PassengerId     Name   Age   Fare Survived
  0            1     John 22.0    7.25         0
  1            2     Jane   NaN 71.28          1
  2            3      Bob 25.0     NaN         1
  3            4    Alice   NaN   8.05         0
  4            5 Charlie 30.0 10.50            1

Cleaned Dataset:
     PassengerId        Age   Fare Survived
  0          1.0 22.000000    7.25       0.0
  1          2.0 25.666667 71.28         1.0
  2          3.0 25.000000 24.27         1.0
  3          4.0 25.666667    8.05       0.0
     4 5.0 30.000000 10.50     1.0
```

2.Implementation of Linear Regression Model

Introduction to Linear Regression

Linear regression may be defined as the statistical model that analyzes the linear relationship between a dependent variable with given set of independent variables. Linear relationship between variables means that when the value of one or more independent variables will change (increase or decrease), the value of dependent variable will also change accordingly (increase or decrease).

Mathematically the relationship can be represented with the help of following equation

− Y = mX + b

Here, Y is the dependent variable we are trying to predict

X is the dependent variable we are using to make predictions.

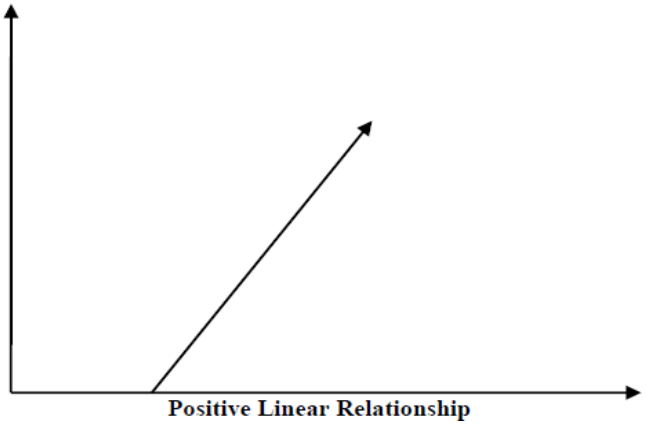
m is the slop of the regression line which represents the effect X has on Y

b is a constant, known as the Y-intercept. If X = 0,Y would be equal to b.

Furthermore, the linear relationship can be positive or negative in nature as explained below −

Positive Linear Relationship

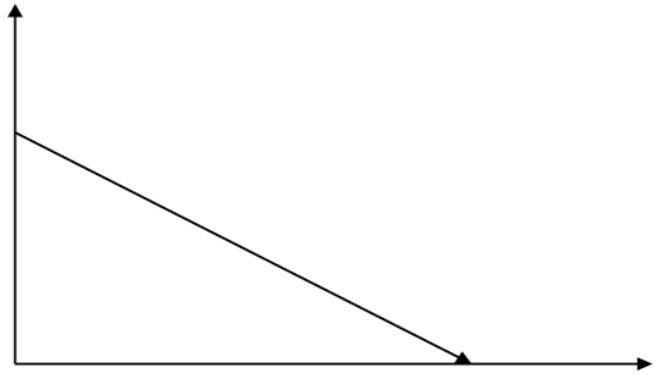
A linear relationship will be called positive if both independent and dependent variable increases. It can be understood with the help of following graph −



Positive Linear Relationship

Negative Linear relationship

A linear relationship will be called positive if independent increases and dependent variable decreases. It can be understood with the help of following graph −



Negative Linear Relationship

Types of Linear Regression

Linear regression is of the following two types −

- Simple Linear Regression
- Multiple Linear

Regression Simple Linear
Regression (SLR)

It is the most basic version of linear regression which predicts a response using a single feature. The assumption in SLR is that the two variables are linearly related.

Making Predictions with Linear Regression

Given the representation is a linear equation, making predictions is as simple as solving the equation for a specific set of inputs.

Let’s make this concrete with an example. Imagine we are predicting weight (y) from height (x). Our linear regression model representation for this problem would be:

y = B0 + B1

* x1 or

weight =B0 +B1 * height

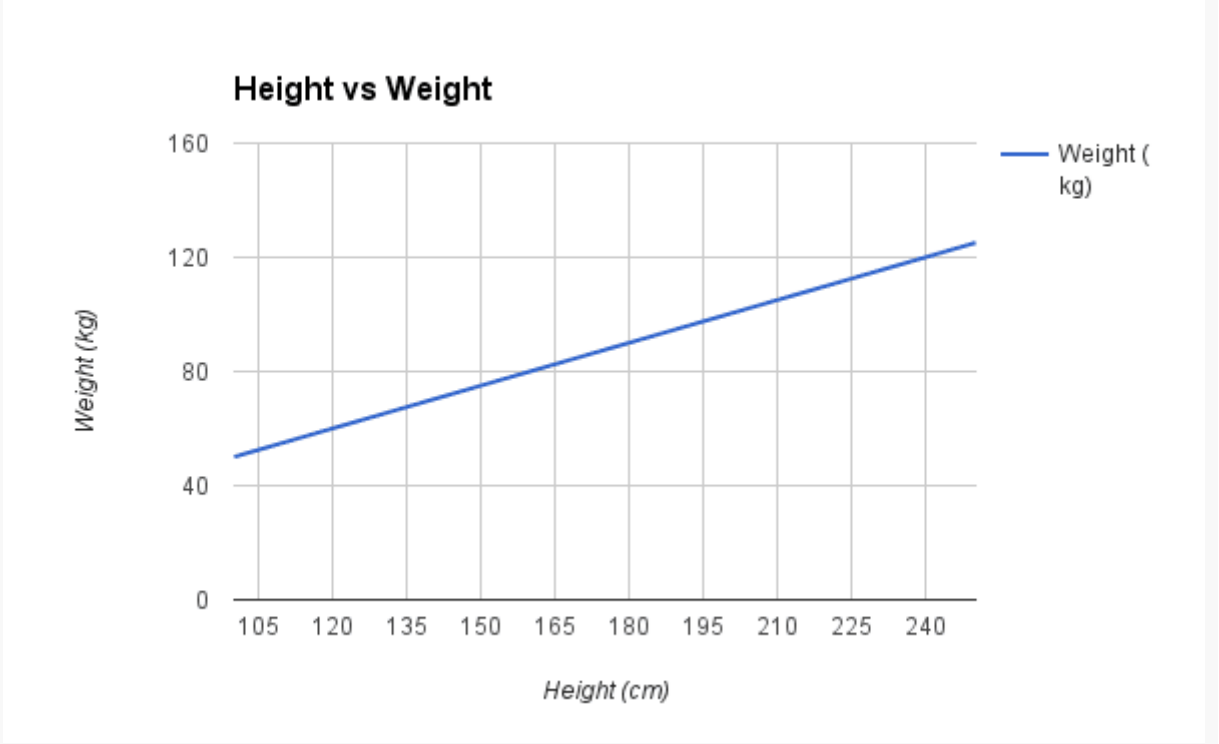
Where B0 is the bias coefficient and B1 is the coefficient for the height column. We use a learning technique to find a good set of coefficient values. Once found, we can plug in different height values to predict the weight.

For example, lets use B0 = 0.1 and B1 = 0.5. Let's plug them in and calculate the weight (in kilograms) for a person with the height of 182 centimeters.

weight = 0.1 + 0.5 * 182

weight = 91.1

You can see that the above equation could be plotted as a line in two-dimensions. The B0 is our starting point regardless of what height we have. We can run through a bunch of heights from 100 to 250 centimeters and plug them to the equation and get weight values, creating our line.



In the following Python implementation example, we are using diabetes dataset from scikit-learn.

First, we will start with importing necessary packages as follows −

```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
from sklearn import datasets, linear_model
from sklearn.metrics import mean_squared_error, r2_score
```

Next, we will load the diabetes dataset and create its

object − diabetes = datasets.load_diabetes()

As we are implementing SLR, we will be using only one feature as follows −

X = diabetes.data[:, np.newaxis, 2]
Next, we need to split the data into training and testing sets as follows −

X_train = X[:-30]

X_test = X[-30:]
Next, we need to split the target into training and testing sets as follows

− y_train = diabetes.target[:-30]

y_test = diabetes.target[-30:]

Now, to train the model we need to create linear regression object as follows −

regr = linear_model.LinearRegression()

Next, train the model using the training sets as follows −

regr.fit(X_train, y_train)

Next, make predictions using the testing set as

follows − y_pred = regr.predict(X_test)

Next, we will be printing some coefficient like MSE, Variance score etc. as follows −

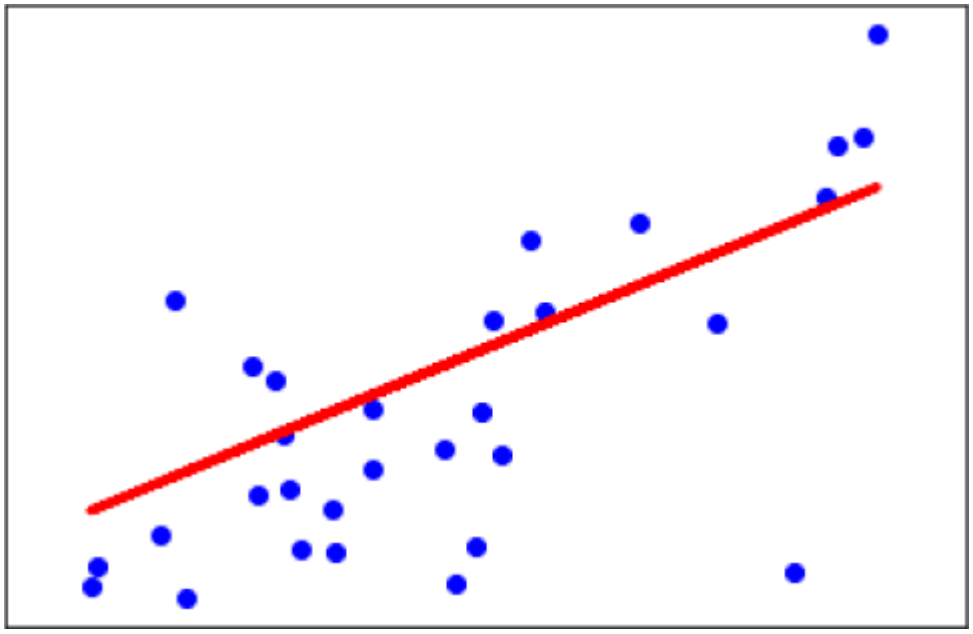
print('Coefficients: \n', regr.coef_)
print("Mean squared error: %.2f" % mean_squared_error(y_test, y_pred))
print('Variance score: %.2f' % r2_score(y_test, y_pred))

Now, plot the outputs as follows −

plt.scatter(X_test, y_test, color='blue')
plt.plot(X_test, y_pred, color='red',
linewidth=3) plt.xticks(())
plt.ytick
s(())
plt.show
()

Output

Coefficients
:
[941.430973
33]
Mean squared error:
3035.06 Variance score:
0.41



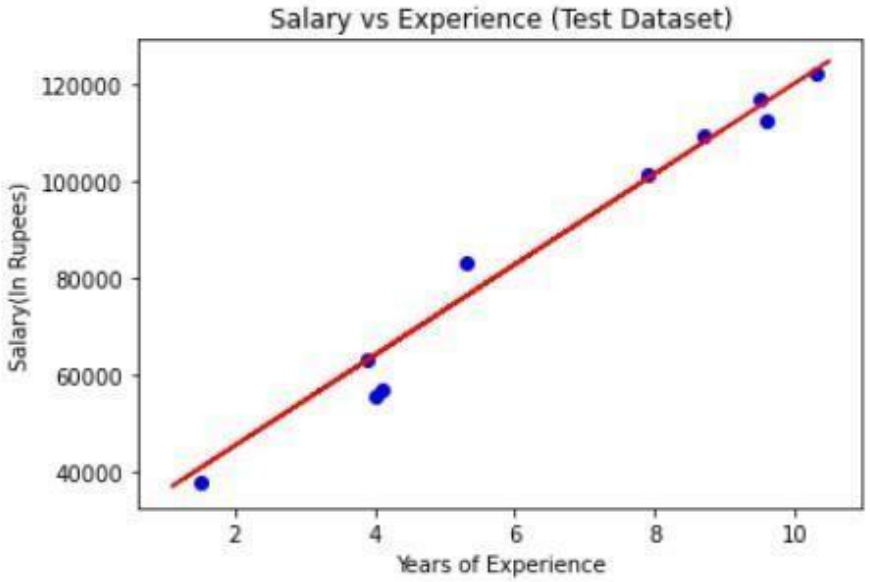
Exercise:2.1: Implement a program to fit a linear regression model to a dataset.

```
import numpy as nm
import matplotlib.pyplot as mtp
import pandas as pd
data_set= pd.read_csv('Salary_Data.csv') x=
data_set.iloc[:, :-1].values
y= data_set.iloc[:, 1].values
# Splitting the dataset into training and test set. from
sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test= train_test_split(x, y, test_size= 1/3, random_state=0) #Fitting
the Simple Linear Regression model to the training dataset
from sklearn.linear_model import LinearRegression
regressor= LinearRegression()
regressor.fit(x_train, y_train) #Prediction of Test and Training set result y_pred=
regressor.predict(x_test)


x_pred= regressor.predict(x_train) mtp.scatter(x_train,
```

```
y_train, color="green") mtp.plot(x_train, x_pred,

color="red") mtp.title("Salary vs Experience (Training

Dataset)") mtp.xlabel("Years of Experience")

mtp.ylabel("Salary(In Rupees)")

mtp.show()

    #visualizing the Test set results mtp.scatter(x_test, y_test, color="blue")
mtp.plot(x_train, x_pred, color="red")

mtp.title("Salary vs Experience (Test Dataset)")

mtp.xlabel("Years of Experience")

mtp.ylabel("Salary(In Rupees)")

mtp.show()
```

Output:


Salary vs Experience (Training Dataset)
Salary(In Rupees)
Years of Experience
120000
100000
80000
60000
40000
2
4
6
8
10


Salary vs Experience (Test Dataset)
Salary(In Rupees)
Years of Experience
120000
100000
80000
60000
40000
2
4
6
8
10

Exercise:2.2: Implement a program to calculate the coefficient of determination for a linear regression model.

First, we will start with importing necessary packages as follows −

```
%matplotlib inline
import matplotlib.pyplot as
plt import numpy as np
from sklearn import datasets, linear_model
from sklearn.metrics import mean_squared_error, r2_score
```

Next, we will load the diabetes dataset and create its object −

diabetes = datasets.load_diabetes()

As we are implementing SLR, we will be using only one feature as follows −

X = diabetes.data[:, np.newaxis, 2]
Next, we need to split the data into training and testing sets as follows −

X_train = X[:-30]

X_test = X[-30:]
Next, we need to split the target into training and testing sets as follows −

y_train = diabetes.target[:-30]

y_test = diabetes.target[-30:]

Now, to train the model we need to create linear regression object as follows

− regr = linear_model.LinearRegression()

Next, train the model using the training sets as follows −

regr.fit(X_train, y_train)

Next, make predictions using the testing set as follows

− y_pred = regr.predict(X_test)

Next, we will be printing some coefficient like MSE, Variance score etc. as follows −

print('Coefficients: \n', regr.coef_)
print("Mean squared error: %.2f" % mean_squared_error(y_test, y_pred))
print('Variance score: %.2f' % r2_score(y_test, y_pred))

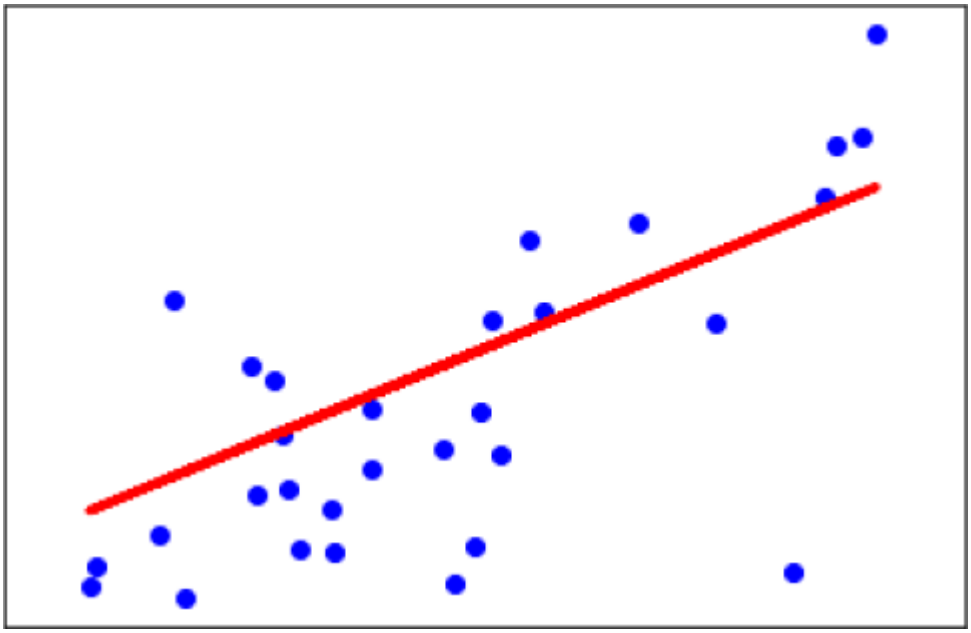
Now, plot the outputs as follows −

plt.scatter(X_test, y_test, color='blue')
plt.plot(X_test, y_pred, color='red', linewidth=3)
plt.xticks(())
plt.yticks(())

```
plt.show()
```

Output:

```
Coefficients:
[941.43097333]
Mean squared error: 3035.06
Variance score: 0.41
```



3. Implementation of Multiple Linear Regression

In the following example, we will perform multiple linear regression for a fictitious economy, where the index_price is the dependent variable, and the 2 independent/input variables are:

- interest_rate
- unemployment_rate

Please note that you will have to validate that several assumptions are met before you apply linear regression models. Most notably, you have to make sure that a linear relationship exists between the dependent variable and the independent variable/s (more on that under the *checking for linearity* section).

Let's now jump into the dataset that we'll be using. The data will be captured using Pandas DataFrame:

```
import pandas as pd

data = {'year': [2017,2017,2017,2017,2017,2017,2017,2017,2017,2017,2017,2016,2016,2016,2016,2016,2016,2016,2016,2016,2016,2016,2016,2016],

    'month': [12,11,10,9,8,7,6,5,4,3,2,1,12,11,10,9,8,7,6,5,4,3,2,1],

    'interest_rate': [2.75,2.5,2.5,2.5,2.5,2.5,2.5,2.25,2.25,2.25,2,2,2,1.75,1.75,1.75,1.75,1.75,1.75,1.75,1.75,1.75,1.75,1.75],

    'unemployment_rate': [5.3,5.3,5.3,5.3,5.4,5.6,5.5,5.5,5.5,5.6,5.7,5.9,6,5.9,5.8,6.1,6.2,6.1,6.1,6.1,5.9,6.2,6.2,6.1],

    'index_price': [1464,1394,1357,1293,1256,1254,1234,1195,1159,1167,1130,1075,1047,965,943,958,971,949,884,866,876,822,704,719]

    }
df = pd.DataFrame(data) print(df)
```

Checking for Linearity

Before you execute a linear regression model, it is advisable to validate that certain assumptions are met.

As noted earlier, you may want to check that a linear relationship exists between the dependent variable and the independent variable/s.

In our example, you may want to check that a linear relationship exists between the:

- index_price (dependent variable) and interest_rate (independent variable)
- index_price (dependent variable) and unemployment_rate (independent variable)

To perform a quick linearity check, you can use scatter diagrams (utilizing the *matplotlib* library). For example, you can use the code below in order to plot the relationship between the index_price and the interest_rate:

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

  data={'year':
[2017,2017,2017,2017,2017,2017,2017,2017,2017,2017,2017,2016,2016,2016,2016,20
16,2016,2016,2016,2016,2016,2016,2016],
'month': [12,11,10,9,8,7,6,5,4,3,2,1,12,11,10,9,8,7,6,5,4,3,2,1],
'interest_rate':
[2.75,2.5,2.5,2.5,2.5,2.5,2.5,2.25,2.25,2.25,2,2,2,1.75,1.75,1.75,1.75,1.75,1.75,1.75,1.75,1.75,
1.75,1.75],
'unemployment_rate':
[5.3,5.3,5.3,5.3,5.4,5.6,5.5,5.5,5.5,5.5,5.6,5.7,5.9,6,5.9,5.8,6.1,6.2,6.1,6.1,6.1,5.9,6.2,6.2,6.1],
'index_price':
[1464,1394,1357,1293,1256,1254,1234,1195,1159,1167,1130,1075,1047,965,943,958,971,94
9,884,866,876,822,704,719]
}
df = pd.DataFrame(data)


plt.scatter(df['interest_rate'], df['index_price'], color='red') plt.title('Index Price Vs Interest Rate',
fontsize=14) plt.xlabel('Interest Rate', fontsize=14)
plt.ylabel('Index Price', fontsize=14) plt.grid(True)
plt.show()
```

You'll notice that indeed a linear relationship exists between the index_price and the interest_rate. Specifically, when interest rates go up, the index price also goes up.

And for the second case, you can use this code in order to plot the relationship between the index_price and the unemployment_rate:

```
df = pd.DataFrame(data)

plt.scatter(df['unemployment_rate'], df['index_price'], color='green') plt.title('Index Price Vs
Unemployment Rate', fontsize=14) plt.xlabel('Unemployment Rate', fontsize=14)
plt.ylabel('Index Price', fontsize=14) plt.grid(True)
plt.show()
```

You'll notice that a linear relationship also exists between the index_price and the unemployment_rate – when the unemployment rates go up, the index price goes down (here we still have a linear relationship, but with a negative slope).

Next, we are going to perform the actual multiple linear regression in Python.

Performing the Multiple Linear Regression

Once you added the data into Python, you may use either sklearn or statsmodels to get the regression results.

Either method would work, but let's review both methods for illustration purposes.

```
import pandas as pd
from sklearn import linear_model
import statsmodels.api as sm
data = {'year':
[2017,2017,2017,2017,2017,2017,2017,2017,2017,2017,2017,2016,2016,2016,2016,20
16,2016,2016,2016,2016,2016,2016,2016],
'month': [12,11,10,9,8,7,6,5,4,3,2,1,12,11,10,9,8,7,6,5,4,3,2,1],
'interest_rate':
[2.75,2.5,2.5,2.5,2.5,2.5,2.5,2.25,2.25,2.25,2,2,2,1.75,1.75,1.75,1.75,1.75,1.75,1.75,1.75,1.75,
1.75,1.75],
'unemployment_rate':
[5.3,5.3,5.3,5.3,5.4,5.6,5.5,5.5,5.5,5.5,5.6,5.7,5.9,6,5.9,5.8,6.1,6.2,6.1,6.1,6.1,5.9,6.2,6.2,6.1],
'index_price':
[1464,1394,1357,1293,1256,1254,1234,1195,1159,1167,1130,1075,1047,965,943,958,971,94
9,884,866,876,822,704,719]
}
df = pd.DataFrame(data)
x = df[['interest_rate','unemployment_rate']]
y = df['index_price']

# with sklearn

regr = linear_model.LinearRegression()
regr.fit(x, y)
print('Intercept: \n', regr.intercept_)
print('Coefficients: \n', regr.coef_) # with statsmodels
x = sm.add_constant(x) # adding a constant model = sm.OLS(y, x).fit()
predictions = model.predict(x)
print_model = model.summary()
print(print_model)
```

Once you run the code in Python, you'll observe two parts:

(1) The first part shows the output generated by sklearn:

```
Intercept: 1798.4039776258564
Coefficients:
[ 345.54008701 -250.14657137]
```

This output includes the intercept and coefficients. You can use this information to build the multiple

linear regression equation as follows:

index_price = (intercept) + (interest_rate coef)*X_1 + (unemployment_rate coef)*X_2 And

once you plug the numbers:

index_price = (1798.4040) + (345.5401)*X_1 + (-250.1466)*X_2

(2) The second part displays a comprehensive table with statistical info generated by *statsmodels*.

This information can provide you additional insights about the model used (such as the fit of the model, standard errors, etc):

OLS Regression Results

Dep. Variable:	index_price	R-squared:	0.898
Model:	OLS	Adj. R-squared:	0.888
Method:	Least Squares	F-statistic:	92.07
Date:	Sat, 30 Jul 2022	Prob (F-statistic):	4.04e-11
Time:	13:47:01	Log-Likelihood:	-134.61
No. Observations	24	AIC:	275.2
Df Residuals:	21	BIC:	278.8
Df Model:	2		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	1798.4040	899.248	2.000	0.059	-71.685	
interest_rate	345.5401	111.367	3.103	0.005	113.940	577.140
unemployment_rate	-250.1466	117.950	-2.121	0.046	-495.437	

Omnibus:	2.691	Durbin-Watson:	0.530
Prob(Omnibus):	0.260	Jarque-Bera (JB):	1.551
Skew:	-0.612	Prob(JB):	0.461
Kurtosis:	3.226	Cond. No.	394.

Exercise:3.1: Use the given Adertisements.CSV file and apply the multiple linear regression.

Importing necessary libraries

```
import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split

from sklearn.linear_model import LinearRegression

from sklearn.metrics import mean_squared_error, r2_score


# Load the dataset

file_path = 'Advertisements.csv'  # Update with the correct file path

data = pd.read_csv(file_path)


# Display the first few rows of the dataset

print("Dataset Preview:")

print(data.head())


# Check for missing values

print("\nChecking for missing values:")

print(data.isnull().sum())


# Define independent variables (features) and dependent variable (target)

X = data[['TV', 'Radio', 'Newspaper']]  # Replace with appropriate column
names from your file

y = data['Sales']  # Replace with the appropriate target column name


# Split the dataset into training and testing sets

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)


# Create and train the linear regression model
```

```
model = LinearRegression()

model.fit(X_train, y_train)


# Model coefficients and intercept

print("\nModel Coefficients:")

print("Intercept:", model.intercept_)

print("Coefficients:", model.coef_)


# Make predictions

y_pred = model.predict(X_test)


# Evaluate the model

mse = mean_squared_error(y_test, y_pred)

r2 = r2_score(y_test, y_pred)

print("\nModel Evaluation:")

print("Mean Squared Error (MSE):", mse)

print("R-squared (R2):", r2)


# Visualize the predicted vs actual values

plt.scatter(y_test, y_pred, color='blue', alpha=0.5)

plt.xlabel("Actual Sales")

plt.ylabel("Predicted Sales")

plt.title("Actual vs Predicted Sales")

plt.grid()

plt.show()
```

II YEAR II SEMESTER MACHINE LEARNING

4. Implementation of Logistic Regression

Logistic regression is a supervised learning classification algorithm used to predict the probability of a target variable. The nature of target or dependent variable is dichotomous, which means there would be only two possible classes.

In simple words, the dependent variable is binary in nature having data coded as either 1 (stands for success/yes) or 0 (stands for failure/no).

Mathematically, a logistic regression model predicts P(Y=1) as a function of X. It is one of the simplest ML algorithms that can be used for various classification problems such as spam detection, Diabetes prediction, cancer detection etc.

Types of Logistic Regression

Generally, logistic regression means binary logistic regression having binary target variables, but there can be two more categories of target variables that can be predicted by it. Based on those number of categories, Logistic regression can be divided into following types −

Binary or Binomial

In such a kind of classification, a dependent variable will have only two possible types either 1 and 0. For example, these variables may represent success or failure, yes or no, win or loss etc.

Multinomial

In such a kind of classification, dependent variable can have 3 or more possible unordered types or the types having no quantitative significance. For example, these variables may represent “Type A” or “Type B” or “Type C”.

Ordinal

In such a kind of classification, dependent variable can have 3 or more possible ordered types or the types having a quantitative significance. For example, these variables may represent “poor” or “good”, “very good”, “Excellent” and each category can have the scores like 0,1,2,3.

It is a special case of linear regression where the target variable is categorical in nature. It uses a log of odds as the dependent variable. Logistic Regression predicts the probability of occurrence of a binary event utilizing a logit function.
Linear Regression Equation:

$$y = \beta 0 + \beta 1X1 + \beta 2X2 + \ldots\ldots + \beta nXn$$

Where, y is dependent variable and x1, x2 ... and Xn are explanatory variables.
Sigmoid Function:

$$p = 1/1 + e^{-y}$$

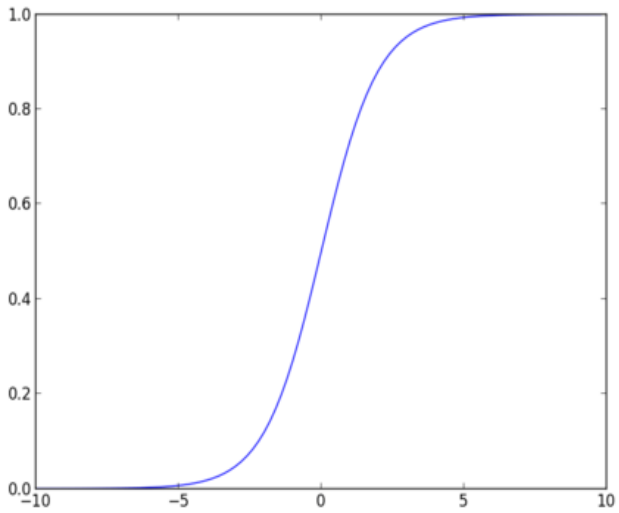
Apply Sigmoid function on linear regression:

$$p = 1/1 + e^{-(\beta 0+\beta 1X1+\beta 2X2.....\beta nXn)}$$

Sigmoid Function

The sigmoid function, also called logistic function gives an 'S' shaped curve that can take any real-valued number and map it into a value between 0 and 1. If the curve goes to positive infinity, y predicted will become 1, and if the curve goes to negative infinity, y predicted will become 0. If the output of the sigmoid function is more than 0.5, we can classify the outcome as 1 or YES, and if it is less than 0.5, we can classify it as 0 or NO. For example: If the output is 0.75, we can say in terms of probability as: There is a 75 percent chance that patient will suffer from cancer.

$$f(x) = \frac{1}{1 + e^{-(x)}}$$



Model building in Scikit-learn

Let's build the diabetes prediction model.
Here, you are going to predict diabetes using Logistic Regression Classifier.
Let's first load the required Pima Indian Diabetes dataset using the pandas' read CSV function. You can download data from the following link: https://www.kaggle.com/uciml/pima-indians- diabetes-database

Loading

Data #import
pandas import
pandas as pd
col_names = ['pregnant', 'glucose', 'bp', 'skin', 'insulin', 'bmi', 'pedigree', 'age',
'label'] # load dataset

pima = pd.read_csv("pima-indians-diabetes.csv", header=None, names=col_names) pima.head()

	pregnant	glucose	bp	skin	insulin	bmi	pedigree	age	label
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

Selecting Feature

Here, you need to divide the given columns into two types of variables dependent(or target variable) and independent variable(or feature variables).

#split dataset in features and target variable
feature_cols = ['pregnant', 'insulin', 'bmi', 'age','glucose','bp','pedigree']
X = pima[feature_cols] #
Features y = pima.label #
Target variable

Splitting Data

To understand model performance, dividing the dataset into a training set and a test set is a good strategy.
Let's split dataset by using function train_test_split(). You need to pass 3 parameters features, target, and test_set size. Additionally, you can use random_state to select records randomly.

# split X and y into training and testing sets
from sklearn.cross_validation import train_test_split
X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.25,random_state=0)

Here, the Dataset is broken into two parts in a ratio of 75:25. It means 75% data will be used for model training and 25% for model testing.

Model Development and Prediction

First, import the Logistic Regression module and create a Logistic Regression classifier object using LogisticRegression() function.
Then, fit your model on the train set using fit() and perform prediction on the test set using predict().

```
# import the class
from sklearn.linear_model import
LogisticRegression # instantiate the model (using
the default parameters) logreg =
LogisticRegression()
# fit the model with
data
logreg.fit(X_train,y_tr
ain)

y_pred=logreg.predict(X_test)
```

Model Evaluation using Confusion Matrix

A confusion matrix is a table that is used to evaluate the performance of a classification model. You can also visualize the performance of an algorithm. The fundamental of a confusion matrix is the number of correct and incorrect predictions are summed up class-wise.

```
# import the metrics
class from sklearn
import metrics
cnf_matrix = metrics.confusion_matrix(y_test,
y_pred) cnf_matrix

array([[119, 11],
       [ 26, 36]])
```

Confusion Matrix Evaluation Metrics

Let's evaluate the model using model evaluation metrics such as accuracy, precision, and recall.

```
print("Accuracy:",metrics.accuracy_score(y_test,
y_pred))
print("Precision:",metrics.precision_score(y_test,
y_pred))     print("Recall:",metrics.recall_score(y_test,
y_pred))
```

Accuracy: 0.8072916666666666
Precision: 0.7659574468085106
Recall: 0.5806451612903226

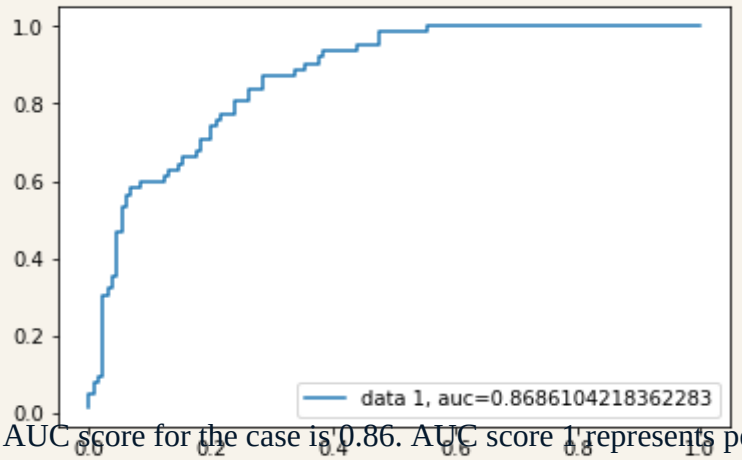
Well, you got a classification rate of 80%, considered as good accuracy.
Precision: Precision is about being precise, i.e., how accurate your model is. In other words, you can say, when a model makes a prediction, how often it is correct. In your prediction case, when your Logistic Regression model predicted patients are going to suffer from diabetes, that patients have 76% of the time. Recall: If there are patients who have diabetes in the test set and your Logistic Regression model can identify it 58% of the time.

ROC Curve

Receiver Operating Characteristic (ROC) curve is a plot of the true positive rate against the false

positive rate. It shows the tradeoff between sensitivity and specificity.

```
y_pred_proba =
logreg.predict_proba(X_test)[::,1] fpr, tpr, _ =
metrics.roc_curve(y_test, y_pred_proba) auc =
metrics.roc_auc_score(y_test, y_pred_proba)
plt.plot(fpr,tpr,label="data 1, auc="+str(auc))
plt.legend(loc=4)
plt.show()
```



AUC score for the case is 0.86. AUC score 1 represents perfect classifier, and 0.5 represents a worthless classifier.

Exercise:4.1: Implement a program to fit a logistic regression model to a dataset.

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, confusion_matrix x
= np.arange(10).reshape(-1, 1)
y = np.array([0, 0, 0, 0, 1, 1, 1, 1, 1, 1])
model = LogisticRegression(solver='liblinear', random_state=0)
model.fit(x, y)
model = LogisticRegression(solver='liblinear', random_state=0).fit(x, y)

model.classes_
model.intercept_
model.coef_
model.predict_proba(x)
model.predict(x)
model.score(x, y)
confusion_matrix(y, model.predict(x))
print(classification_report(y, model.predict(x)))
#Improve the Model
```

#You can improve your model by setting different parameters. For example, let's work with #the regularization strength C equal to 10.0, instead of the default value of 1.0:

```
model = LogisticRegression(solver='liblinear', C=10.0, random_state=0)
model.fit(x, y)
model.classes_
model.intercept_
model.coef_
model.predict_proba(x)
model.predict(x)
model.score(x, y)
confusion_matrix(y, model.predict(x)) print(classification_report(y, model.predict(x)))
```

Output:

	precision	recall	f1-score	support
0	1.00	1.00	1.00	4
1	1.00	1.00	1.00	6
accuracy			1.00	10
macro avg	1.00	1.00	1.00	10
weighted avg	1.00	1.00	1.00	10

Exercise:4.2: Implement a program to calculate the odds ratio for a logistic regression model.

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split from
sklearn.linear_model import LogisticRegression from
sklearn.metrics import accuracy_score
# Load or create your dataset
# for simplicity, let's create a sample dataset data
= {
    'Age': [25, 30, 35, 40, 45, 50, 55, 60, 65, 70],
    'Smoker': [0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
    'Outcome': [0, 0, 0, 1, 1, 1, 1, 1, 1, 1]
}
df = pd.DataFrame(data)
# Split the dataset into features (X) and target variable (y) X =
df[['Age', 'Smoker']]
y = df['Outcome']
# 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) #
Train a logistic regression model
model = LogisticRegression() model.fit(X_train,
y_train)
# Make predictions on the test set
y_pred = model.predict(X_test)
```

 # Calculate the accuracy of the model accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy:.2f}')

```
# Calculate odds ratio for each feature
odds_ratio = np.exp(model.coef_)
print(f'Odds Ratio: {odds_ratio}')
```

5. Implementation of Multi-Output Classification

Multi-output classification is a type of machine learning that predicts multiple outputs simultaneously. In multi-output classification, the model will give two or more outputs after making any prediction. In other types of classifications, the model usually predicts only a single output.

An example of a multi-output classification model is a model that predicts the type and color of fruit simultaneously. The type of fruit can be, orange, mango and pineapple. The color can be, red, green, yellow, and orange. The multi-output classification solves this problem and gives two prediction results.

In this tutorial, we will build a multi-output text classification model using the Netflix dataset. The model will classify the input text as either TV Show or Movie. This will be the first output. The model will also classify the rating as: TV-MA, TV-14, TV-PG, R, PG-13 and TV-Y. The rating will be the second output. We will use Scikit-Learn Multi Output Classifier algorithm to build this model.

Code:

```
# Load EDA

Pkgs import

pandas as pd

import numpy

as np # Load

Dataset

df =

pd.read_csv("netflix_titles_dataset.csv")

df.head()

# Class 1/Target 1/Output 1

Distribution

df['type'].value_counts()

# Class 2/Target 2/Output 2

Distribution

df['rating'].value_counts()

# Text Cleaning

import neattext.functions as

nfx df['title'] =

df['title'].nfx.lower()

df['title'] = df['title'].apply(nfx.remove_stopwords)
```

```
df.head()
```

```
from sklearn.linear_model import LogisticRegression
from sklearn.feature_extraction.text import
CountVectorizer from sklearn.model_selection import
train_test_split
from sklearn.metrics import accuracy_score
from sklearn.multioutput import
MultiOutputClassifier # Features & Labels
Xfeatures = df['title']
ylabels =
df[['type','rating']] #
Split Data
x_train,x_test,y_train,y_test =
train_test_split(Xfeatures,ylabels,test_size=0.3,random_state
=7) from sklearn.pipeline import Pipeline
pipe_lr = Pipeline(steps=[('cv',CountVectorizer()),
                ('lr_multi',MultiOutputClassifier(LogisticRegression()))])
# Fit on Dataset
pipe_lr.fit(x_train,y_tra
in) # Accuracy Score
pipe_lr.score(x_test,y_t
est) # Sample
Prediction
print(x_test.iloc[0])
print("Actual
Prediction:",y_test.iloc[0]) pred1 =
x_test.iloc[0]
pred1
pipe_lr.predict([pred
1]) # Prediction
Prob
print(pipe_lr.classes
_)
pipe_lr.predict_proba([pred1])
```

Output:

[array(['Movie', 'TV Show'], dtype=object), array(['PG-13', 'R', 'TV-14', 'TV-MA', 'TV-PG', 'TV-Y'], dtype=object)]

```
[array([[0.74445483, 0.25554517]]),
 array([[0.12310188, 0.07038494, 0.21476461, 0.46916205, 0.10270243,
         0.01988409]])]
```

Exercise:5.1: Generate a multi-output data with a make_multilabel_classification function. The target dataset contains 10 features (x), 2 classes (y), and 5000 samples, and implement the above algorithm

```
# Importing necessary libraries
import numpy as np
import pandas as pd
from sklearn.datasets import make_multilabel_classification
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

# Generate the dataset
X, Y = make_multilabel_classification(n_samples=5000, n_features=10, n_classes=2, random_state=42)

# Convert Y into a continuous format for regression (sum of labels as continuous targets)
# For demonstration, you can replace this logic as per your regression objective
Y_continuous = Y.sum(axis=1)

# Convert the dataset into a DataFrame for better visualization
data = pd.DataFrame(X, columns=[f"Feature_{i+1}" for i in range(X.shape[1])])
data['Target'] = Y_continuous

# Display the first few rows of the dataset
print("Dataset Preview:")
print(data.head())

# Define independent variables (features) and dependent variable (target)
X_train, X_test, y_train, y_test = train_test_split(X, Y_continuous, test_size=0.3, random_state=42)

# Create and train the multiple linear regression model
model = LinearRegression()
model.fit(X_train, y_train)

# Model coefficients and intercept
print("\nModel Coefficients:")
print("Intercept:", model.intercept_)
print("Coefficients:", model.coef_)

# Make predictions
y_pred = model.predict(X_test)

# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
print("\nModel Evaluation:")
print("Mean Squared Error (MSE):", mse)
print("R-squared (R2):", r2)

# Visualize the predicted vs actual values
import matplotlib.pyplot as plt
```

```
plt.scatter(y_test, y_pred, color='blue', alpha=0.5)
plt.xlabel("Actual Target Values")
plt.ylabel("Predicted Target Values")
plt.title("Actual vs Predicted Target Values")
plt.grid()
plt.show()
```

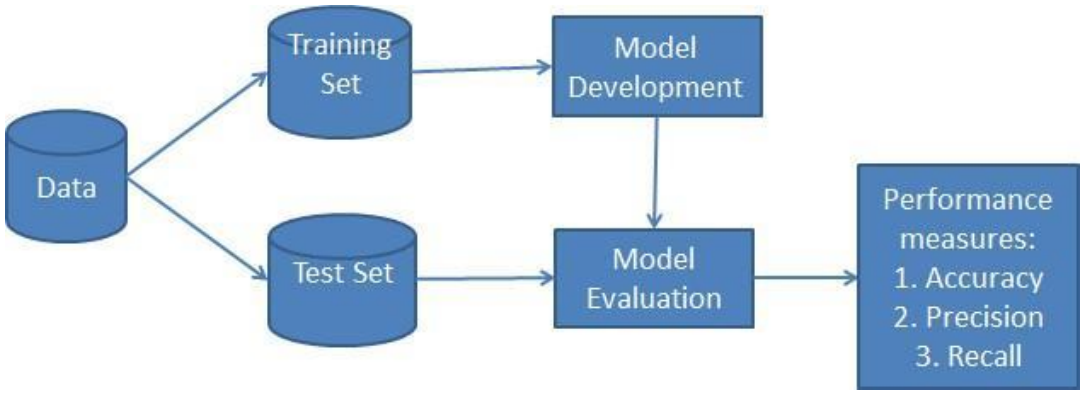
6. Implementation of Naïve Bayes classification algorithm

Naive Bayes is the most straightforward and fast classification algorithm, which is suitable for a large chunk of data. Naive Bayes classifier is successfully used in various applications such as spam filtering, text classification, sentiment analysis, and recommender systems. It uses Bayes theorem of probability for prediction of unknown class.

Classification Workflow

Whenever you perform classification, the first step is to understand the problem and identify potential features and label. Features are those characteristics or attributes which affect the results of the label. For example, in the case of a loan distribution, bank managers identify the customer's occupation, income, age, location, previous loan history, transaction history, and credit score. These characteristics are known as features that help the model classify customers.

The classification has two phases, a learning phase and the evaluation phase. In the learning phase, the classifier trains its model on a given dataset, and in the evaluation phase, it tests the classifier's performance. Performance is evaluated on the basis of various parameters such as accuracy, error, precision, and recall.



What is Naive Bayes Classifier?

Naive Bayes is a statistical classification technique based on Bayes Theorem. It is one of the simplest supervised learning algorithms. Naive Bayes classifier is the fast, accurate and reliable algorithm. Naive Bayes classifiers have high accuracy and speed on large datasets.

Naive Bayes classifier assumes that the effect of a particular feature in a class is independent of other features. For example, a loan applicant is desirable or not depending on his/her income, previous loan and transaction history, age, and location. Even if these features are interdependent, these features are still considered independently. This assumption simplifies computation, and that's why it is considered as naive. This assumption is called class conditional independence.

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)}$$

- P(h): the probability of hypothesis h being true (regardless of the data). This is known as the prior probability of h.
- P(D): the probability of the data (regardless of the hypothesis). This is known as the prior probability.
- P(h|D): the probability of hypothesis h given the data D. This is known as posterior probability.
- P(D|h): the probability of data d given that the hypothesis h was true. This is known as posterior probability.

Naive Bayes Classifier with Synthetic Dataset

Generating the Dataset

```
from sklearn.datasets import
make_classification X, y =
make_classification(
    n_features=6,
    n_classes=3,
    n_samples=800,
    n_informative=2,
    random_state=1,
    n_clusters_per_class
    =1,
)
import matplotlib.pyplot as plt
plt.scatter(X[:, 0], X[:, 1], c=y,
marker="*");
```

Train Test Split

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test =
    train_test_split( X, y, test_size=0.33,
    random_state=125
)
```

Model Building and Training

```
from sklearn.naive_bayes import GaussianNB
```

 # Build a Gaussian Classifier

```
model = GaussianNB()
```

 # Model training

```
model.fit(X_train, y_train)
```

 # Predict Output

```
predicted = model.predict([X_test[6]])

print("Actual Value:", y_test[6])
print("Predicted Value:",
predicted[0])
```

Model Evaluation

```
from sklearn.metrics
    import (
    accuracy_score,
    confusion_matrix,
    ConfusionMatrixDispl
    ay, f1_score,
)

y_pred = model.predict(X_test)
accuray = accuracy_score(y_pred, y_test)
f1 = f1_score(y_pred, y_test, average="weighted")

print("Accuracy:",
accuray) print("F1
Score:", f1)
```

Visualize the Confusion matrix

```
labels = [0,1,2]
cm = confusion_matrix(y_test, y_pred, labels=labels)
```

```
disp = ConfusionMatrixDisplay(confusion_matrix=cm,
display_labels=labels) disp.plot();
```

Exercise:6.1: Apply Naive Bayes Classifier with Loan Dataset

```
# Importing necessary libraries
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
import seaborn as sns
import matplotlib.pyplot as plt

# Load the Loan Dataset
file_path = 'LoanDataset.csv'  # Replace with the actual file path
data = pd.read_csv(file_path)

# Display the first few rows of the dataset
print("Dataset Preview:")
print(data.head())

# Check for missing values
print("\nMissing Values:")
print(data.isnull().sum())

# Preprocess the dataset (fill missing values or drop them if necessary)
data = data.dropna()  # For simplicity, dropping missing values. Use imputation if needed.

# Encode categorical variables if present
# Assuming 'Gender', 'Married', and 'Loan_Status' are categorical
data = pd.get_dummies(data, columns=['Gender', 'Married'], drop_first=True)
data['Loan_Status'] = data['Loan_Status'].map({'Y': 1, 'N': 0})  # Encode target as binary

# Define features and target variable
X = data.drop('Loan_Status', axis=1)  # Features
y = data['Loan_Status']  # Target

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Create and train the Naive Bayes Classifier
model = GaussianNB()
model.fit(X_train, y_train)

# Make predictions
y_pred = model.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)

print("\nModel Evaluation:")
print(f"Accuracy: {accuracy:.2f}")
print("\nConfusion Matrix:")
```

```
print(conf_matrix)
print("\nClassification Report:")
print(classification_report(y_test, y_pred))

# Visualize the confusion matrix
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=['Not Approved', 'Approved'],
yticklabels=['Not Approved', 'Approved'])
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.title("Confusion Matrix")
plt.show()
```

7. Implementation of Fuzzy K-means clustering:

Fuzzy K-Means Clustering is a soft clustering algorithm that allows each data point to belong to multiple clusters with varying degrees of membership. Unlike traditional K-Means clustering, which assigns each data point to exactly one cluster, Fuzzy K-Means uses membership probabilities to represent how strongly a data point belongs to each cluster. This is particularly useful for datasets where cluster boundaries are not well-defined.

Clustering Workflow

The process of clustering involves understanding the dataset, identifying key features, and determining the number of clusters. Features are the attributes or characteristics of the data points that help group them into clusters. For example, in market segmentation, features might include customer age, income, spending habits, and geographical location. These features help in categorizing customers into distinct segments.

Clustering typically involves two main phases: initialization and optimization. During initialization, cluster centers are selected, either randomly or using a specific heuristic. In the optimization phase, the algorithm iteratively updates the cluster centers and membership probabilities to minimize a cost function. The performance of clustering can be evaluated using metrics like silhouette score, Davies-Bouldin index, and intra-cluster variance.

Fuzzy K-Means is an extension of the traditional K-Means algorithm that incorporates fuzzy logic. In this approach, each data point has a degree of membership in each cluster, quantified by membership values between 0 and 1. These values sum to 1 across all clusters for a given data point. The algorithm minimizes a cost function that considers both the distance of a data point from cluster centers and its membership degree.

The fuzziness parameter, denoted by , controls the level of cluster overlap. Higher values of result in softer clustering, while lower values make the algorithm behave more like traditional K-Means.

Key Terminology:

Cluster Centers: The central points of each cluster, representing the mean or centroid.

Membership Matrix: A matrix where each entry represents the degree of membership of a data point to a cluster.

Fuzziness Parameter (m): A parameter that controls the degree of fuzziness in the clustering process.

Fuzzy K-Means Clustering with Synthetic Dataset

#Generating the Dataset

```
from sklearn.datasets import make_blobs
X, _ = make_blobs(
   n_features=2, n_samples=800, centers=3, random_state=42
)
import matplotlib.pyplot as plt
plt.scatter(X[:, 0], X[:, 1], marker="*");
```

#Train-Test Split

```
from sklearn.model_selection import train_test_split
X_train, X_test = train_test_split(X, test_size=0.33, random_state=125)
```

Model Building and Training

To implement Fuzzy K-Means, we use the `skfuzzy` library:

```
import skfuzzy as fuzz

# Number of clusters
n_clusters = 3

# Perform Fuzzy K-Means clustering
cntr, u, u0, d, jm, p, fpc = fuzz.cluster.cmeans(
    X_train.T, c=n_clusters, m=2.0, error=0.005, maxiter=1000, init=None
)

#Predict Output
```

To classify new data points, calculate the membership degrees:

```
# Predict cluster membership for a test point
u_test, _, _, _, _, _ = fuzz.cluster.cmeans_predict(X_test.T, cntr, m=2.0, error=0.005, maxiter=1000)

# Find the most probable cluster for the 6th test point
predicted_cluster = u_test[:, 6].argmax()
print("Predicted Cluster:", predicted_cluster)
```

Model Evaluation

Evaluate the clustering performance using the fuzzy partition coefficient (FPC):

```
print("Fuzzy Partition Coefficient (FPC):", fpc)
```

#Visualize the Clusters

```
# Visualize the clusters and centroids
for j in range(n_clusters):
    plt.scatter(X_train[u[j] > 0.5, 0], X_train[u[j] > 0.5, 1], label=f"Cluster {j}")

plt.scatter(cntr[:, 0], cntr[:, 1], c='red', marker='X', s=100, label="Centroids")
plt.legend()
plt.show()
```

Exercise:7.1: Implement a program to cluster a dataset using Fuzzy K-means clustering.

```
# Import necessary libraries
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
import skfuzzy as fuzz

# Generate synthetic dataset
n_samples = 300
n_features = 2
```

```
n_clusters = 3
random_state = 42

# Creating a dataset
X, _ = make_blobs(n_samples=n_samples, centers=n_clusters, n_features=n_features,
random_state=random_state)

# Visualize the dataset
plt.scatter(X[:, 0], X[:, 1], c='blue', marker='o', edgecolor='k')
plt.title("Generated Data Points")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.grid()
plt.show()

# Fuzzy C-Means clustering
# Transpose the data for compatibility with skfuzzy
X_transposed = np.transpose(X)

# Set the number of clusters
cntr, u, u0, d, jm, p, fpc = fuzz.cluster.cmeans(
    X_transposed, c=n_clusters, m=2, error=0.005, maxiter=1000, init=None
)

# Assign clusters to each data point
cluster_membership = np.argmax(u, axis=0)

# Visualize clustered data
colors = ['red', 'green', 'blue']
for i in range(n_clusters):
    plt.scatter(X[cluster_membership == i, 0], X[cluster_membership == i, 1], color=colors[i],
label=f"Cluster {i+1}")

plt.title("Fuzzy K-Means Clustering")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.legend()
plt.grid()
plt.show()

# Membership values for a specific point (example: first point in X)
print("Membership values for the first data point:", u[:, 0])

# Fuzzy Partition Coefficient (FPC) - A measure of the quality of the clustering
print("\nFuzzy Partition Coefficient (FPC):", fpc)
```

8. Hierarchical clustering

Hierarchical Clustering

Hierarchical clustering is an unsupervised machine learning algorithm used to group data points into clusters based on their similarity. Unlike other clustering methods such as K-Means, hierarchical clustering does not require specifying the number of clusters beforehand. Instead, it creates a tree-like structure called a dendrogram that represents the hierarchy of clusters.

Types of Hierarchical Clustering

Agglomerative (Bottom-Up): Starts with each data point as an individual cluster and merges the closest clusters iteratively until only one cluster remains.

Divisive (Top-Down): Starts with all data points in a single cluster and recursively splits them into smaller clusters.

Hierarchical Clustering Workflow

The process of hierarchical clustering involves several key steps:

Compute the Distance Matrix:
Calculate the pairwise distance between all data points. Common distance metrics include Euclidean, Manhattan, and Cosine distance.

Initialize Clusters:
In agglomerative clustering, each data point starts as its own cluster.

Merge Clusters:
Find the two closest clusters and merge them. The proximity between clusters is calculated using linkage criteria such as:

Single Linkage: Distance between the closest points in two clusters.

Complete Linkage: Distance between the farthest points in two clusters.

Average Linkage: Average distance between all points in two clusters.

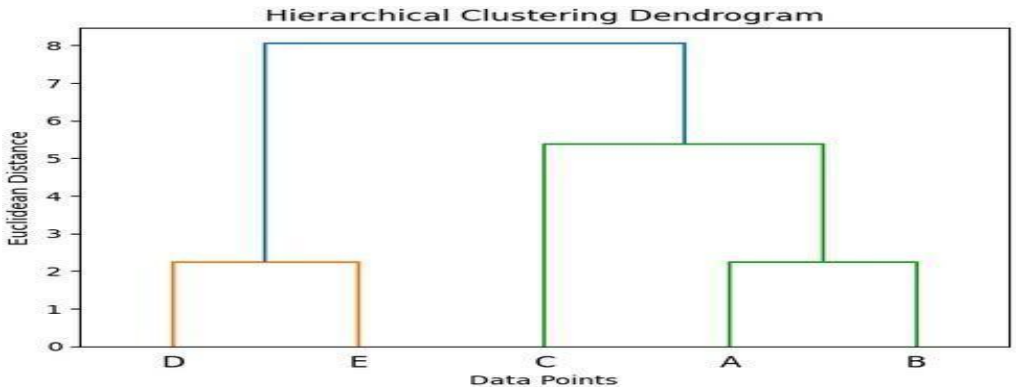
Centroid Linkage: Distance between the centroids of two clusters.

Repeat Until One Cluster Remains:
Continue merging clusters until all data points belong to a single cluster.

Dendrogram Construction:
Plot the dendrogram to visualize the hierarchy of clusters and determine the optimal number of clusters by cutting the dendrogram at an appropriate level.

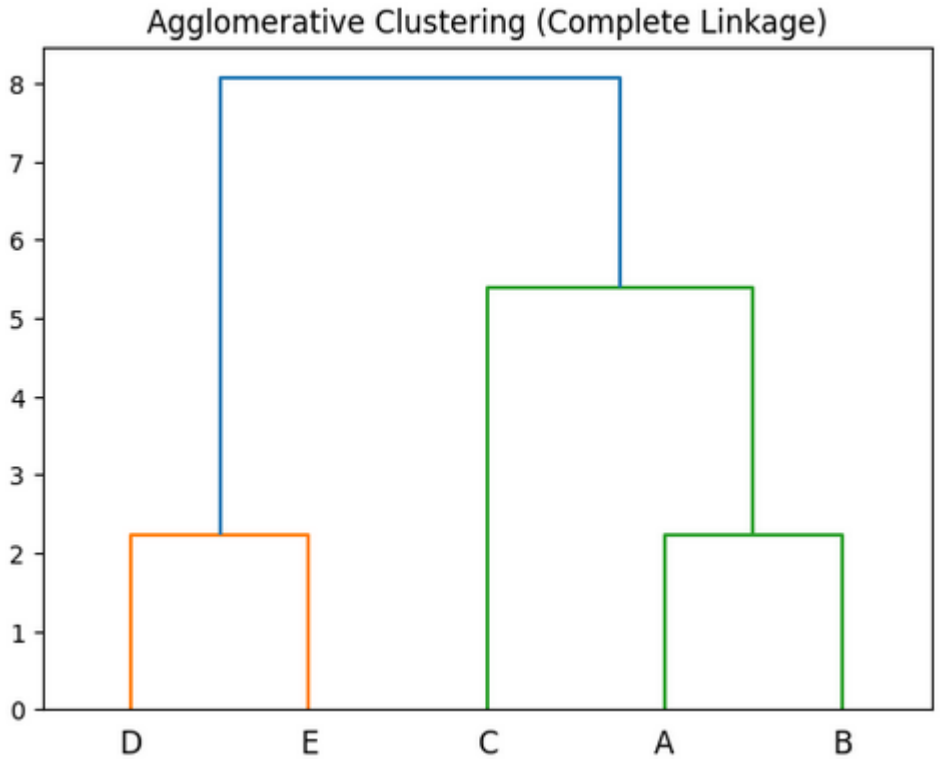
Exercise:8.1: Implement a program to perform hierarchical clustering on a dataset.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy.cluster.hierarchy import linkage, dendrogram #
Generate example data
np.random.seed(42)
X = np.array([[2, 5], [3, 3], [5, 8], [8, 5], [10, 6]])
# Perform hierarchical clustering using linkage function linkage_matrix
= linkage(X, method='complete', metric='euclidean') # Plot the
dendrogram
dendrogram(linkage_matrix, labels=['A', 'B', 'C', 'D', 'E'])
plt.title('Hierarchical Clustering Dendrogram')
plt.xlabel('Data Points')
plt.ylabel('Euclidean Distance')
plt.show()
```



Exercise:8.2: Implement a program to calculate the agglomerative clustering algorithm for a hierarchical clustering.

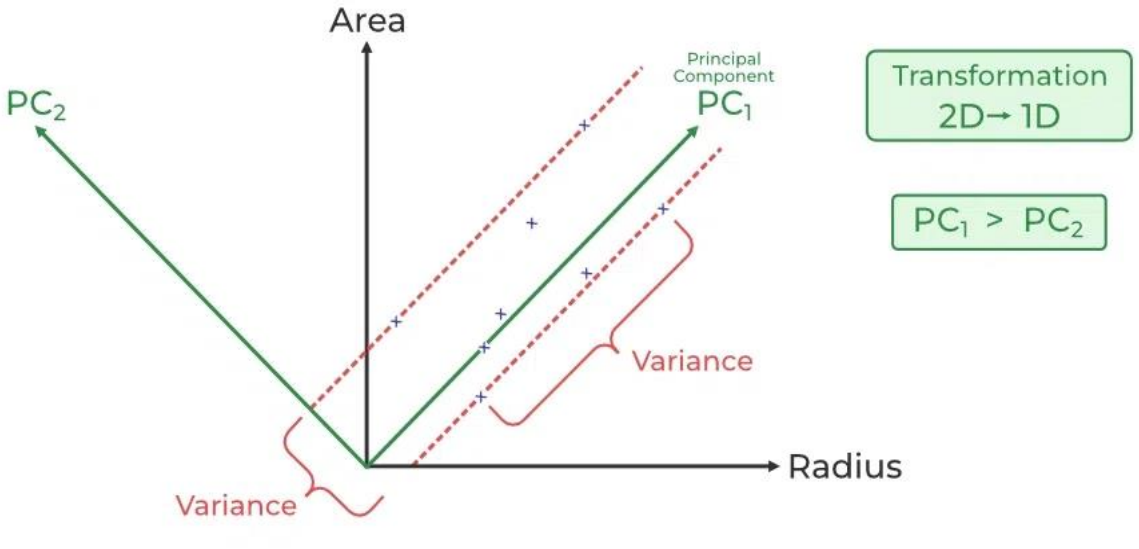
```
import numpy as np
from sklearn.cluster import AgglomerativeClustering from
scipy.cluster.hierarchy import dendrogram, linkage import
matplotlib.pyplot as plt

# Generate example data
np.random.seed(42)
X = np.array([[2, 5], [3, 3], [5, 8], [8, 5], [10, 6]])

# Agglomerative clustering with complete linkage
agg_cluster_complete = AgglomerativeClustering(n_clusters=None, linkage='complete',
distance_threshold=0)
agg_labels_complete = agg_cluster_complete.fit_predict(X)

# Plot dendrogram for complete linkage
linkage_matrix_complete = linkage(X, method='complete')
dendrogram(linkage_matrix_complete, labels=['A', 'B', 'C', 'D', 'E']) plt.title('Agglomerative Clustering
(Complete Linkage)') plt.show()
```



9. Principal Component Analysis (PCA)

Principal Component Analysis (PCA) is a dimensionality reduction technique used to transform a high-dimensional dataset into a lower-dimensional space while retaining as much variability (information) as possible. It is widely used in machine learning, data preprocessing, and exploratory data analysis to simplify datasets, reduce computation time, and eliminate multicollinearity.

How PCA Works



PCA transforms the original features into a new set of features called **principal components**. These components are linear combinations of the original features and are ordered by the amount of variance they capture:

1. **Variance Maximization**: PCA identifies directions (principal components) that maximize the variance in the dataset.
2. **Orthogonality**: The principal components are orthogonal (uncorrelated) to each other.
3. **Dimensionality Reduction**: By selecting a subset of the principal components, PCA reduces the dimensionality of the data while preserving most of the variance.

Mathematical Steps of PCA

1. **Standardize the Data**: Ensure that all features have a mean of 0 and a standard deviation of 1.
2. **Compute the Covariance Matrix**: The covariance matrix captures the relationships between features.
3. **Calculate Eigenvalues and Eigenvectors**: Decompose the covariance matrix to find its eigenvalues (representing variance) and eigenvectors (representing directions).
4. **Sort Eigenvalues**: Rank the eigenvalues in descending order and select the top eigenvalues to form the principal components.
5. **Project the Data**: Transform the data into the new space defined by the selected principal components, where is the matrix of selected eigenvectors.

Exercise:9.1: Implement a program to perform principal component analysis on a dataset.

```
import numpy as np
# Step 1: Data standardization def
standardize(X):
    return (X - np.mean(X, axis=0)) / np.std(X, axis=0)

# Step 2: Covariance matrix calculation def
compute_covariance_matrix(X):
    return np.cov(X.T)
```

```
# Step 3: Eigenvalue and eigenvector calculation def
find_eigenvectors_and_eigenvalues(X):
    cov_matrix = compute_covariance_matrix(X)
    eigenvalues, eigenvectors = np.linalg.eig(cov_matrix)
    return eigenvalues, eigenvectors

# Step 4: Principal component calculation def
project_data(X, eigenvectors, k):
    sorted_eigenvectors = eigenvectors[:, np.argsort(-eigenvectors)[:,:k]]
    return np.dot(X, sorted_eigenvectors)

# Step 5: Dimensionality reduction
  def get_variance_explained(eigenvalues, k): return
    sum(eigenvalues[:k]) / sum(eigenvalues)

# Example usage
X = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
X_std = standardize(X)
eigenvalues, eigenvectors = find_eigenvectors_and_eigenvalues(X_std)
projected_data = project_data(X_std, eigenvectors, 2) variance_explained
= get_variance_explained(eigenvalues, 2) print("Standardized data:")
print(X_std) print("Covariance
matrix:")
print(compute_covariance_matrix(X_std))
print("Eigenvalues:")
print(eigenvalues) print("Eigenvectors:")
print(eigenvectors)
print("Projected data:")
print(projected_data)
print("Variance explained:")
 print(variance_explained)
```

```
Standardized data:
[[-1.22474487 -1.22474487 -1.22474487]
 [ 0.          0.          0.        ]
 [ 1.22474487  1.22474487  1.22474487]]
Covariance matrix:
[[1.5 1.5 1.5]
 [1.5 1.5 1.5]
 [1.5 1.5 1.5]]
Eigenvalues:
[0.  4.5 0. ]
Eigenvectors:
[[-0.81649658  0.57735027  0.        ]
 [ 0.40824829  0.57735027 -0.70710678]
 [ 0.40824829  0.57735027  0.70710678]]
Projected data:
[[[-1.41421356  0.29289322]
  [-0.54818816 -0.34108138]
  [-2.28023897 -2.07313218]]

 [[ 0.          0.        ]
  [ 0.          0.        ]
  [ 0.          0.        ]]

 [[ 1.41421356 -0.29289322]
  [ 0.54818816  0.34108138]
  [ 2.28023897  2.07313218]]]
Variance explained:
1.0
```

Exercise:9.2: Implement a program to calculate the covariance matrix for a dataset.

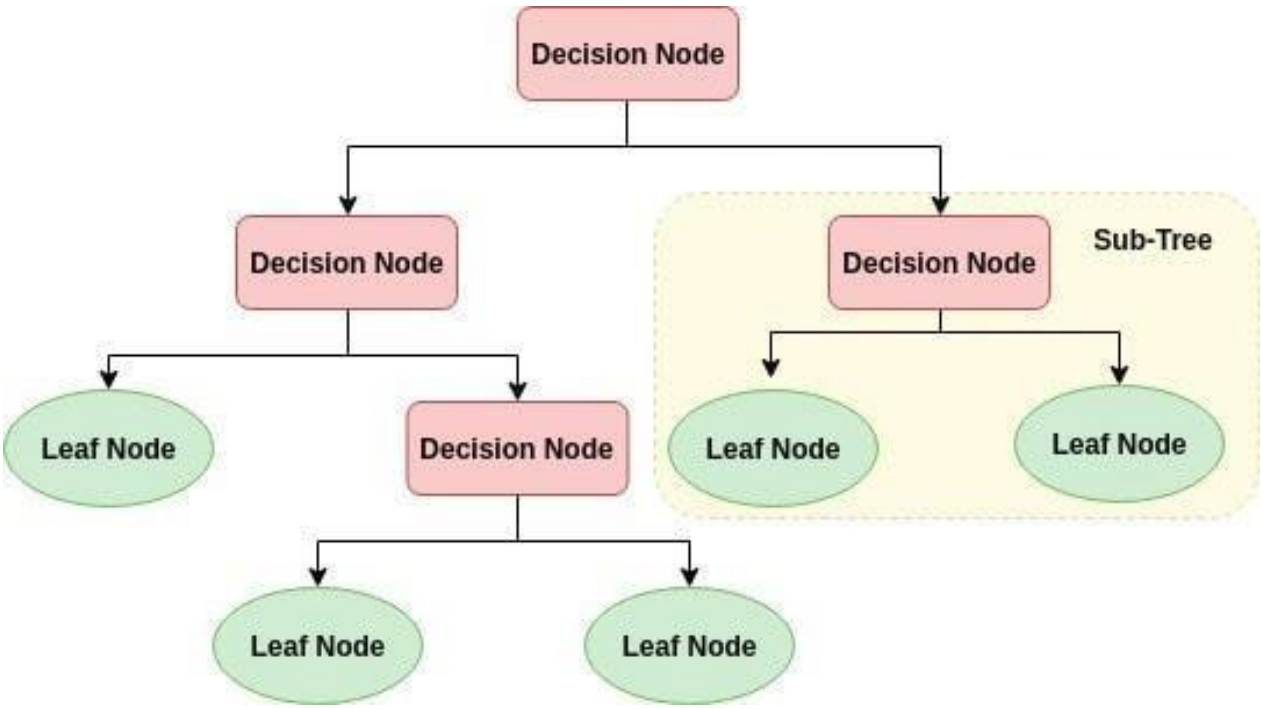
```
import numpy as np # Predefined dataset
dataset = np.array([
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9],
    [10, 11, 12]
])
# Calculate the covariance matrix covariance_matrix = np.cov(dataset, rowvar=False)

# Print the covariance matrix
print("Covariance Matrix:")
print(covariance_matrix)
```

Output:

```
Covariance Matrix: [[15.
15. 15.]
   [15. 15. 15.]
[15. 15. 15.]]
```

10. Implementation of Decision Tree Classification Algorithm

A decision tree is a flowchart-like tree structure where an internal node represents feature (or attribute), the branch represents a decision rule, and each leaf node represents the outcome. The topmost node in a decision tree is known as the root node. It learns to partition on the basis of the attribute value. It partitions the tree in recursively manner call recursive partitioning. This flowchart-like structure helps you in decision making. It's visualization like a flowchart diagram which easily mimics the human level thinking. That is why decision trees are easy to understand and interpret.



Decision Tree is a white box type of ML algorithm. It shares internal decision-making logic, which is not available in the black box type of algorithms such as Neural Network. Its training time is faster compared to the neural network algorithm. The time complexity of decision trees is a function of the number of records and number of attributes in the given data. The decision tree is a distribution-free or non-parametric method, which does not

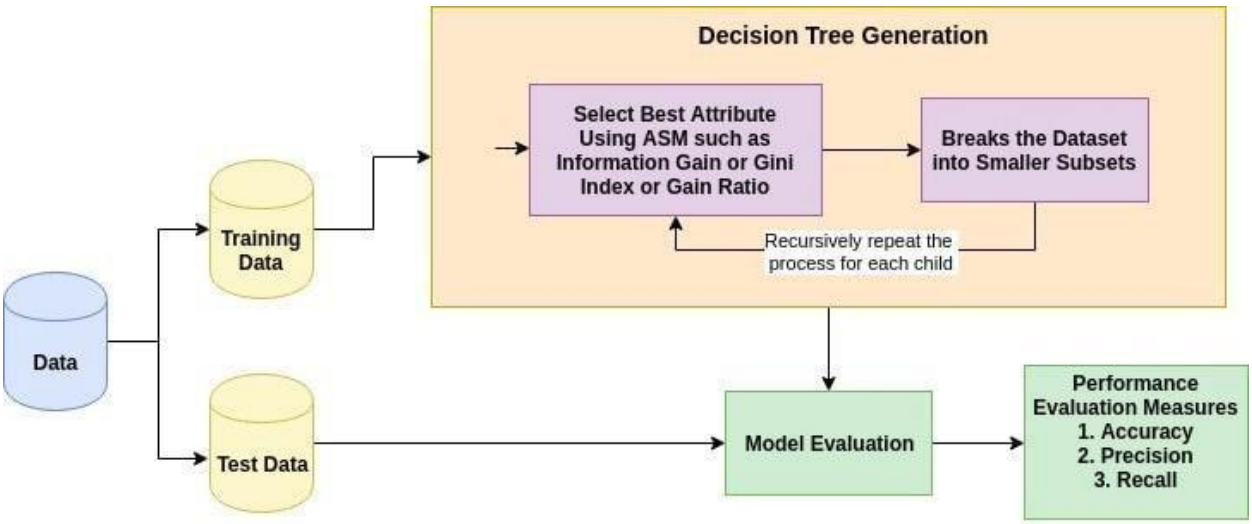
depend upon probability distribution assumptions. Decision trees can handle high dimensional data with good accuracy.

How does the Decision Tree Algorithm Work?

The basic idea behind any decision tree algorithm is as follows:

1. Select the best attribute using Attribute Selection Measures(ASM) to split the records.
2. Make that attribute a decision node and breaks the dataset into smaller subsets.

3. Starts tree building by repeating this process recursively for each child until one of the condition will match:
    - All the tuples belong to the same attribute value.
    - There are no more remaining attributes.
    - There are no more instances.



Attribute Selection Measures

Attribute selection measure is a heuristic for selecting the splitting criterion that partition data into the best possible manner. It is also known as splitting rules because it helps us to determine breakpoints for tuples on a given node. ASM provides a rank to each feature(or attribute) by explaining the given dataset. Best score attribute will be selected as a splitting attribute (Source). In the case of a continuous-valued attribute, split points for branches also need to define. Most popular selection measures are Information Gain, Gain Ratio, and Gini Index.

Information Gain

Shannon invented the concept of entropy, which measures the impurity of the input set. In physics and mathematics, entropy referred as the randomness or the impurity in the system. In information theory, it refers to the impurity in a group of examples. Information gain is the decrease in entropy. Information gain computes the difference between entropy before split and average entropy after split of the dataset based on given attribute values. ID3 (Iterative Dichotomiser) decision tree algorithm uses information gain.

$$\text{Info(D)} = -\sum_{i=1}^{m} pi \log_2 pi$$

Where, Pi is the probability that an arbitrary tuple in D belongs to class Ci.

$$Info_A(D)=\sum_{j=1}^{V}\frac{|Dj|}{|D|} \quad X\ Info(D_j)$$

$$Gain(A)=Info(D)- Info_A(D)$$

Where,

- Info(D) is the average amount of information needed to identify the class label of a tuple in D.
- |Dj|/|D| acts as the weight of the jth partition.
- InfoA(D) is the expected informa-tion required to classify a tuple from D based on the partitioning by A.

The attribute A with the highest information gain, Gain(A), is chosen as the splitting attribute at node N().

Gain Ratio

Information gain is biased for the attribute with many outcomes. It means it prefers the attribute with a large number of distinct values. For instance, consider an attribute with a unique identifier such as customer_ID has zero info(D) because of pure partition. This maximizes the information gain and creates useless partitioning.

C4.5, an improvement of ID3, uses an extension to information gain known as the gain ratio. Gain ratio handles the issue of bias by normalizing the information gain using Split Info. Java implementation of the C4.5 algorithm is known as J48, which is available in WEKA data mining tool.

$$SplitInfo_A(D) = -\sum_{j=1}^{v}\frac{|D_j|}{|D|}\times \log_2\left(\frac{|D_j|}{|D|}\right)$$

Where,

- |Dj|/|D| acts as the weight of the jth partition.
- v is the number of discrete values in attribute A.

The gain ratio can be defined as

$$GainRatio(A) = \frac{Gain(A)}{SplitInfo_A(D)}$$

The attribute with the highest gain ratio is chosen as the splitting attribute (Source).

Gini index

Another decision tree algorithm CART (Classification and Regression Tree) uses the Gini method to create split points.

$$\text{Gini}(D)=1-\sum_{i=1}^{m} P_i^2$$

Where, pi is the probability that a tuple in D belongs to class Ci.

The Gini Index considers a binary split for each attribute. You can compute a weighted sum of the impurity of each partition. If a binary split on attribute A partitions data D into D1 and D2, the Gini index of D is:

$$\text{Gini}_A(D)=\frac{|D1|}{|D|}\text{Gini}(D_1)+\frac{|D2|}{|D|}\text{Gini}(D_2)$$

In case of a discrete-valued attribute, the subset that gives the minimum gini index for that chosen is selected as a splitting attribute. In the case of continuous-valued attributes, the strategy is to select each pair of adjacent values as a possible split-point and point with smaller gini index chosen as the splitting point.

$$\Delta Gini(A) = Gini(D) - Gini_A(D).$$

The attribute with minimum Gini index is chosen as the splitting attribute.

Code:

```
# Load libraries
import pandas
as pd
from sklearn.tree import DecisionTreeClassifier # Import Decision Tree Classifier
from sklearn.model_selection import train_test_split # Import train_test_split
function
from sklearn import metrics #Import scikit-learn metrics module for accuracy calculation
col_names = ['pregnant', 'glucose', 'bp', 'skin', 'insulin', 'bmi', 'pedigree', 'age', 'label']
# load dataset
pima = pd.read_csv("diabetes.csv", header=None,
names=col_names) pima.head()
#split dataset in features and target variable
feature_cols = ['pregnant', 'insulin', 'bmi', 'age','glucose','bp','pedigree']
X = pima[feature_cols] #
Features y = pima.label #
Target variable
```

```
# Split dataset into training set and test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=1) #
70% training and 30% test
# Create Decision Tree classifer
object clf =
DecisionTreeClassifier()
# Train Decision Tree
Classifer clf =
clf.fit(X_train,y_train)
print(X_train.head())
print(y_train.head())
#Predict the response for test
dataset y_pred =
clf.predict(X_test)
print(X_test.head())
print(y_pred.head())
print(y_test.head())

# Model Accuracy, how often is the classifier correct?
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
```

Output: Accuracy:
0.6753246753246753 X_train
y_t
rai
n
X_t
est
y_t
est
y_p
red

Exercise:10.1: Implement a program to construct a decision tree from a dataset.

 # Python program to implement decision tree algorithm and plot the tree

```
# Importing the required libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt from
sklearn import metrics import
seaborn as sns
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn import tree

# Loading the dataset iris =
load_iris()
```

```
#converting the data to a pandas dataframe
data = pd.DataFrame(data = iris.data, columns = iris.feature_names)

#creating a separate column for the target variable of iris dataset
data['Species'] = iris.target

#replacing the categories of target variable with the actual names of the species target
= np.unique(iris.target)
target_n = np.unique(iris.target_names)
target_dict = dict(zip(target, target_n))
data['Species'] = data['Species'].replace(target_dict)

# Separating the independent dependent variables of the dataset x =
data.drop(columns = "Species")
y = data["Species"] names_features
= x.columns target_labels =
y.unique()

# Splitting the dataset into training and testing datasets
  x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.3, random_state = 93) #
   Importing the Decision Tree classifier class from sklearn
from sklearn.tree import DecisionTreeClassifier


# Creating an instance of the classifier class
  dtc = DecisionTreeClassifier(max_depth = 3, random_state = 93) #
    Fitting the training dataset to the model
dtc.fit(x_train, y_train)

# Plotting the Decision Tree plt.figure(figsize =
(30, 10), facecolor = 'b')
Tree = tree.plot_tree(dtc, feature_names = names_features, class_names = target_labels, rounded
= True, filled = True, fontsize = 14)
plt.show()
y_pred = dtc.predict(x_test)

# Finding the confusion matrix
confusion_matrix = metrics.confusion_matrix(y_test, y_pred) matrix
= pd.DataFrame(confusion_matrix)
axis = plt.axes() sns.set(font_scale =
1.3)
plt.figure(figsize = (10,7))

# Plotting heatmap
sns.heatmap(matrix, annot = True, fmt = "g", ax = axis, cmap = "magma")
axis.set_title('Confusion Matrix')
axis.set_xlabel("Predicted Values", fontsize = 10)
axis.set_xticklabels(['] + target_labels) axis.set_ylabel(
"True Labels", fontsize = 10)
axis.set_yticklabels(list(target_labels), rotation = 0)
plt.show()
```

Exercise:10.2: Implement a program to calculate the accuracy of a decision tree model.

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier from
sklearn.metrics import accuracy_score
# Load or create your dataset
# For simplicity, let's create a sample dataset data
= {
      'Feature1': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
      'Feature2': [2, 4, 1, 3, 6, 8, 5, 7, 10, 9],
      'Target': [0, 0, 0, 1, 1, 1, 1, 1, 1, 1]
}
df = pd.DataFrame(data)
# Split the dataset into features (X) and target variable (y) X =
df[['Feature1', 'Feature2']]
y = df['Target']
# 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) #
Train a decision tree model
model = DecisionTreeClassifier()
model.fit(X_train, y_train)
# Make predictions on the test set
y_pred = model.predict(X_test)

# Calculate the accuracy of the model accuracy =
accuracy_score(y_test, y_pred) print(f'Accuracy:
{accuracy:.2f}')
# Check if the accuracy meets the desired threshold (97%)
desired_accuracy = 0.97
if accuracy >= desired_accuracy:
   print(f'Model accuracy meets the desired threshold of {desired_accuracy:.2%}')
else:
      print(f'Model accuracy does not meet the desired threshold of {desired_accuracy:.2%}')
```

Output:

```
Accuracy: 1.00

      Model accuracy meets the desired threshold of 97.00%
```

11. Implementation of Support Vector Machine (SVM)

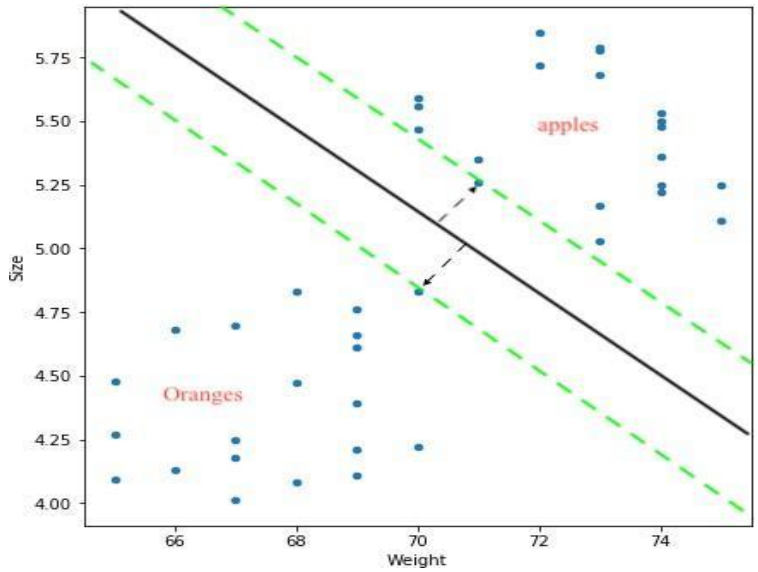
SVM which stands for Support Vector Machine is one of the most popular classification algorithms used in Machine Learning. Support Vector Machine or SVM is a supervised and linear Machine Learning algorithm most commonly used for solving classification problems and is also referred to as Support Vector Classification. There is also a subset of SVM called SVR which stands for Support Vector Regression which uses the same principles to solve regression problems. SVM also supports the kernel method also called the kernel SVM which allows us to tackle non-linearity.

How SVM works?

Just for the sake of understanding, we will leave the machines out of the picture for a minute. Now how would a human being like you and me classify a set of objects scattered on the surface of a table? Ofcourse we will consider all their physical and visual characteristics and then identify based on our prior knowledge. We can easily identify and distinguish apples and oranges based on their colour, texture, shape etc.

Now bringing back the machines, how would a machine identify an apple or an orange. Not surprisingly, it is based on the characteristics that we provide the machine with. It can be size, shape, weight etc. The more features we consider the easier it is to identify and distinguish both.

For the time being, we will just focus on the weight and size(diameter) of apples and oranges. Now how would a machine using SVM, classify a new fruit as either apple or orange just based on the data on the size and weights of some 20 apples and oranges that were observed and labelled? The below image depicts how.



The objective of SVM is to draw a line that best separates the two classes of data points.

SVM generates a line that can cleanly separate the two classes. How clean, you may ask. There are many possible ways of drawing a line that separates the two classes, however, in SVM, it is determined by the **margins** and the **support vectors**.

The margin is the area separating the two dotted green lines as shown in the image above. The more the margin the better the classes are separated. The support vectors are the data points through which each of the green lines passes through. These points are called support vectors as they contribute to the margins and hence the classifier itself. These support vectors are simply the data points lying closest to the border of either of the classes which has a probability of being in either one.

The SVM then generates a hyperplane which has the maximum margin, in this case the black bold line that separates the two classes which is at an optimum distance between both the classes. In case of more than 2 features and multiple dimensions, the line is replaced by a hyperplane that separates multidimensional spaces.

Implementing SVM in Python

Code:

```
import pandas as pd

data = pd.read_csv("apples_and_oranges.csv")

#Splitting the dataset into training and test

samples from sklearn.model_selection import

train_test_split

training_set, test_set = train_test_split(data, test_size = 0.2, random_state = 1)

#Classifying the predictors and target

X_train =

training_set.iloc[:,0:2].values

Y_train =

training_set.iloc[:,2].values X_test

= test_set.iloc[:,0:2].values Y_test =

test_set.iloc[:,2].values

#Initializing Support Vector Machine and fitting the training

data from sklearn.svm import SVC

classifier = SVC(kernel='rbf', random_state = 1)
```

```
classifier.fit(X_train,Y_train)
#Predicting the classes for test
set Y_pred =
classifier.predict(X_test)
#Attaching the predictions to test set for
comparing test_set["Predictions"] = Y_pred
from sklearn.metrics import
confusion_matrix cm =
confusion_matrix(Y_test,Y_pred)
accuracy = float(cm.diagonal().sum())/len(Y_test)
print("\nAccuracy Of SVM For The Given Dataset : ",
accuracy)
```

Output:

```
Accuracy Of SVM For The Given Dataset : 0.875
```

Exercise:11.1: Implement a program to fit a support vector machine model to a dataset.

```
import pandas as pd
data = pd.read_csv("apples_and_oranges.csv")
#Splitting the dataset into training and test samples
from sklearn.model_selection import train_test_split
training_set, test_set = train_test_split(data, test_size = 0.2, random_state = 1)
#Classifying the predictors and target
X_train = training_set.iloc[:,0:2].values
Y_train = training_set.iloc[:,2].values
X_test = test_set.iloc[:,0:2].values Y_test =
test_set.iloc[:,2].values
#Initializing Support Vector Machine and fitting the training data
from sklearn.svm import SVC
```

```
classifier = SVC(kernel='rbf', random_state = 1)

classifier.fit(X_train,Y_train)

#Predicting the classes for test set Y_pred =

classifier.predict(X_test)

#Attaching the predictions to test set for comparing

test_set["Predictions"] = Y_pred

from sklearn.metrics import confusion_matrix cm

= confusion_matrix(Y_test,Y_pred)

accuracy = float(cm.diagonal().sum())/len(Y_test) print("\nAccuracy

of SVM for the Given Dataset : ", accuracy)
```

Output:

```
Accuracy of SVM for the Given Dataset: 0.375
```

12. Study and Implementation of Random Forest

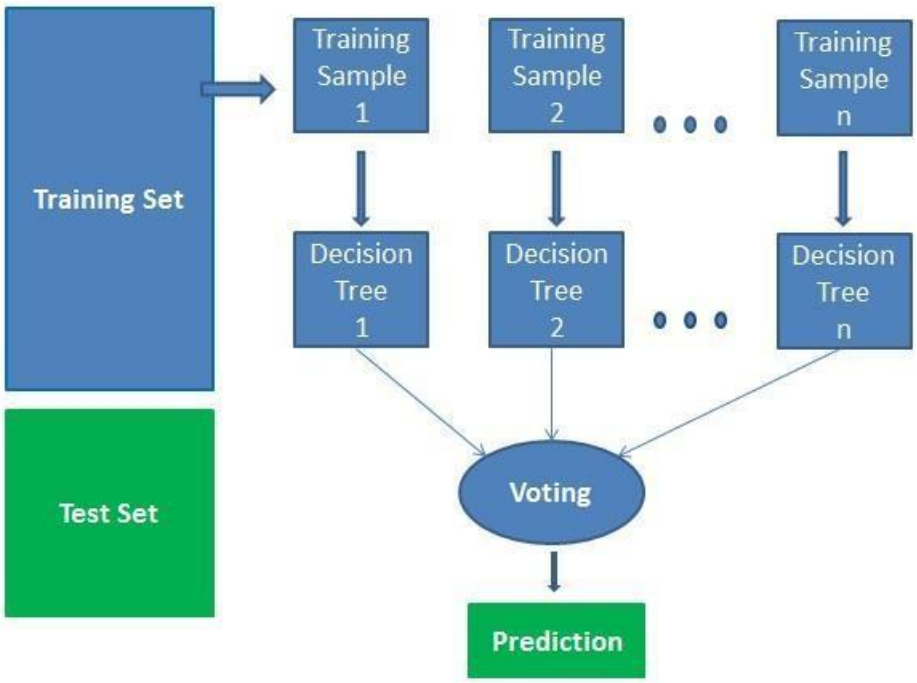
Random forest is a supervised learning algorithm. It can be used both for classification and regression. It is also the most flexible and easy to use algorithm. A forest is comprised of trees. It is said that the more trees it has, the more robust a forest is. Random forest creates decision trees on randomly selected data samples, gets prediction from each tree and selects the best solution by means of voting. It also provides a pretty good indicator of the feature importance.

Random forest has a variety of applications, such as recommendation engines, image classification and feature selection. It can be used to classify loyal loan applicants, identify fraudulent activity and predict diseases. It lies at the base of the Boruta algorithm, which selects important features in a dataset.

How does the Algorithm Work?

It works in four steps:

1. Select random samples from a given dataset.
2. Construct a decision tree for each sample and get a prediction result from each decision tree.
3. Perform a vote for each predicted result.
4. Select the prediction result with the most votes as the final prediction.



Advantages:

- Random forests is considered as a highly accurate and robust method because of the number of decision trees participating in the process.
- It does not suffer from the overfitting problem. The main reason is that it takes the average of all the predictions, which cancels out the biases.
- The algorithm can be used in both classification and regression problems.
- Random forests can also handle missing values. There are two ways to handle these: using median values to replace continuous variables, and computing the proximity- weighted average of missing values.
- You can get the relative feature importance, which helps in selecting the most contributing features for the classifier.

Disadvantages:

- Random forests is slow in generating predictions because it has multiple decision trees. Whenever it makes a prediction, all the trees in the forest have to make a prediction for the same given input and then perform voting on it. This whole process is time- consuming.
- The model is difficult to interpret compared to a decision tree, where you can easily make a decision by following the path in the tree.

Code:

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

#Load dataset
iris = datasets.load_iris()
# print the label species(setosa, versicolor,virginica)
print(iris.target_names)

# print the names of the four features
print(iris.feature_names)# print the iris data (top 5
records) print(iris.data[0:5])

# print the iris labels (0:setosa, 1:versicolor,
2:virginica) print(iris.target)
# Creating a DataFrame of given iris
dataset. import pandas as pd
```

```
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()
# Import train_test_split function
from sklearn.model_selection import train_test_split

X=data[['sepal length', 'sepal width', 'petal length', 'petal width']] # Features
y=data['species'] # Labels

# Split dataset into training set and test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3) # 70% training and
30% test
#Import Random Forest Model
from sklearn.ensemble import RandomForestClassifier

#Create a Gaussian Classifier
clf=RandomForestClassifier(n_estimators=100)

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

y_pred=clf.predict(X_test)
#Import scikit-learn metrics module for accuracy
calculation from sklearn import metrics
# Model Accuracy, how often is the classifier correct?
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
clf.predict([[3, 5, 4, 2]])
#Finding Important Features in Scikit-learn
from sklearn.ensemble import RandomForestClassifier

#Create a Gaussian Classifier
clf=RandomForestClassifier(n_estimators=100)

#Train the model using the training sets
y_pred=clf.predict(X_test) clf.fit(X_train,y_train)
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
            max_depth=None, max_features='auto', max_leaf_nodes=None,
```

```
        min_impurity_decrease=0.0, min_impurity_split=None,
        min_samples_leaf=1, min_samples_split=2,
        min_weight_fraction_leaf=0.0, n_estimators=100, n_jobs=1,
        oob_score=False, random_state=None, verbose=0,
        warm_start=False)
import pandas as pd
feature_imp=
pd.Series(clf.feature_importances_,index=iris.feature_names).sort_values(ascending=False)
feature_imp
```

Output:

Accuracy: 0.9333333333333333
petal length (cm) 0.443844 petal width (cm) 0.438182 sepal length (cm) 0.097607 sepal width (cm) 0.020368 dtype: float64

Exercise:12.1: Implement a program to implement random forests for a decision tree model.

```
from sklearn.datasets import make_classification
from sklearn.ensemble import RandomForestClassifier from
sklearn.model_selection import train_test_split from sklearn.metrics
import accuracy_score

# Generate synthetic data
X, y = make_classification(n_samples=1000, n_features=20, n_informative=10, n_classes=2, random_state=42)
# 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) # Define the
Random Forest classifier
random_forest_model = RandomForestClassifier(n_estimators=100, random_state=42) # Train the
Random Forest model
random_forest_model.fit(X_train, y_train) # Make
predictions on the test set
predictions = random_forest_model.predict(X_test) # Calculate
accuracy
accuracy = accuracy_score(y_test, predictions)
print(f"Accuracy: {accuracy}")
```

Output:

Accuracy: 0.92

"A thought beyond horizons of success committed for professional excellence"

Vision

To be a world class university visualizing a great future for the young aspirants, with innovative nature, research culture and ethical sensitivities to meet the global challenges improving the quality of human life.

Mission

To impart value based futuristic higher education moulding students into globally competent empowered youth, rich in culture and ethics along with professional expertise.

To promote innovation, entrepreneurship, research and development for the broad purpose of fulfilling societal goals such as societal welfare and benefit.

Quality Policy

To pursue continual improvement of teaching learning process of Undergraduate, Post Graduate programs and Research Programs vigorously.

To provide state of the art infrastructure and expertise to impart the quality education.

To groom the students to become intellectually creative and professionally competitive.

To explore the opportunities in the professional fields.

Academic Best Practices

Industry Oriented Curriculum

Technology Training and Certifications

Institution to Corporate Exposure

Interactive Learning

International Career Guidance

Choice Based Flexible Curriculum

MALLA REDDY UNIVERSITY

(Telangana State Private Universities Act No.13 of 2020 and G.O.Ms.No.14, Higher Education (UE) Department)

Maisammaguda, Kompally, Hyderabad - 500 100

Telangana State

www.mallareddyuniversity.ac.in