Practical 1

Aim: - Write a program to implement the naïve Bayesian classifier for a sample training data set stored as a .CSV file. Compute the accuracy of the classifier, considering few test data sets.

Theory: -

The naïve Bayesian (or naïve Bayes) classifier is a simple probabilistic machine learning model based on Bayes' Theorem. It assumes that features (or predictors) are independent of each other, which is why it's called "naïve."

- Bayes' Theorem: It calculates the probability of a class given the features by combining the prior probability of the class with the likelihood of the features.
- Naïve independence assumption: It assumes that the features are conditionally independent of each other given the class.
- Despite the simplification, it often works surprisingly well in real-world applications, especially in text classification and spam filtering.

It's fast, easy to implement, and works well with large datasets.

$$P(C|X) = \frac{P(X|C) \cdot P(C)}{P(X)}$$

Where:

- P(C|X)= Posterior probability of class C given feature set X
- P(X|C) = Likelihood of feature set X given class C
- P(C) = Prior probability of class C
- P(X) = Probability of feature set X (this can be ignored in classification, as it is constant across classes)

For the **naïve** Bayes classifier, the assumption is that the features X1,X2,..., Xn are conditionally independent. So the likelihood P(X|C) becomes:

P(X|C)=P(X1|C)·P(X2|C)·...·P(Xn|C) Thus, the

naïve Bayes formula is:

$$P(C|X) \propto P(C) \cdot \prod_{i=1}^{n} P(X_i|C)$$

This is used to compute the probability for each class C, and the class with the highest probability is selected as the prediction.

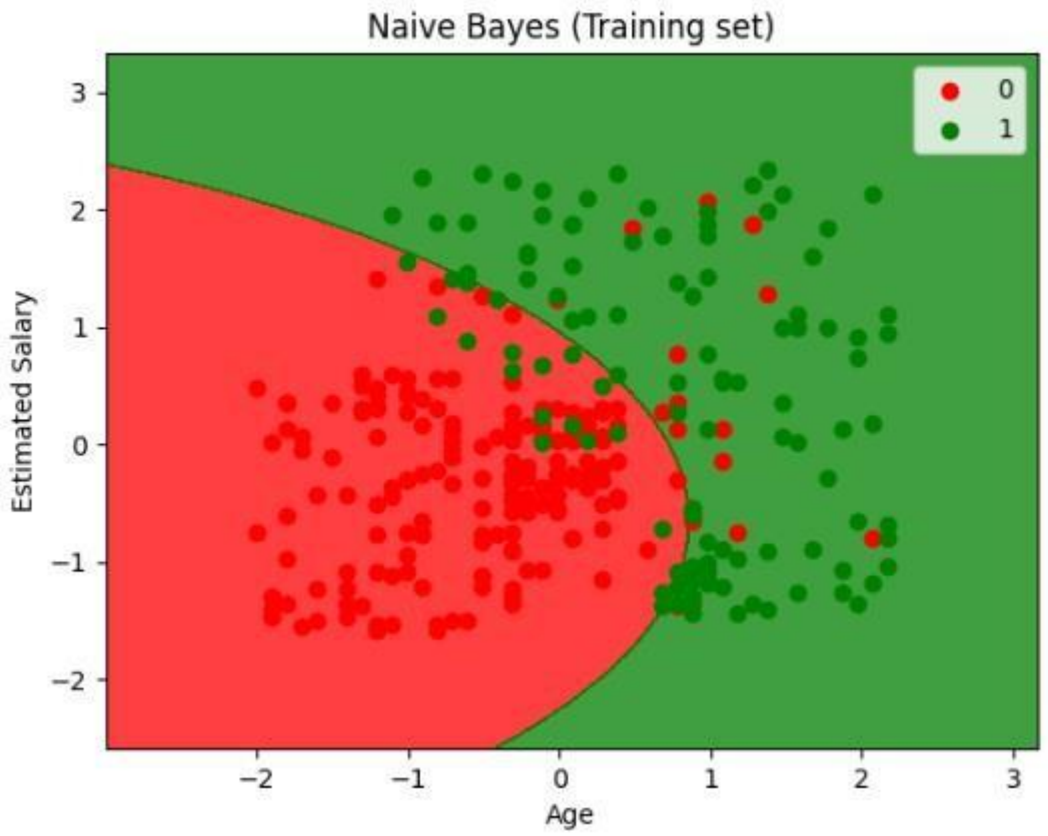
Code: -

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

# Importing the dataset
dataset =
pd.read_csv('Social_Network_Ads.csv')
X = dataset.iloc[:, [2, 3]].values y =
dataset.iloc[:, 4].values

# Splitting the dataset into the Training set
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 = 0.25, random_state = 0)

# Feature Scaling from
sklearn.preprocessing import
StandardScaler sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)

# Fitting classifier to the Training
set from sklearn.naive_bayes import
GaussianNB classifier = GaussianNB()
classifier.fit(X_train, y_train)

# Predicting the Test set
results y_pred =
classifier.predict(X_test)
```

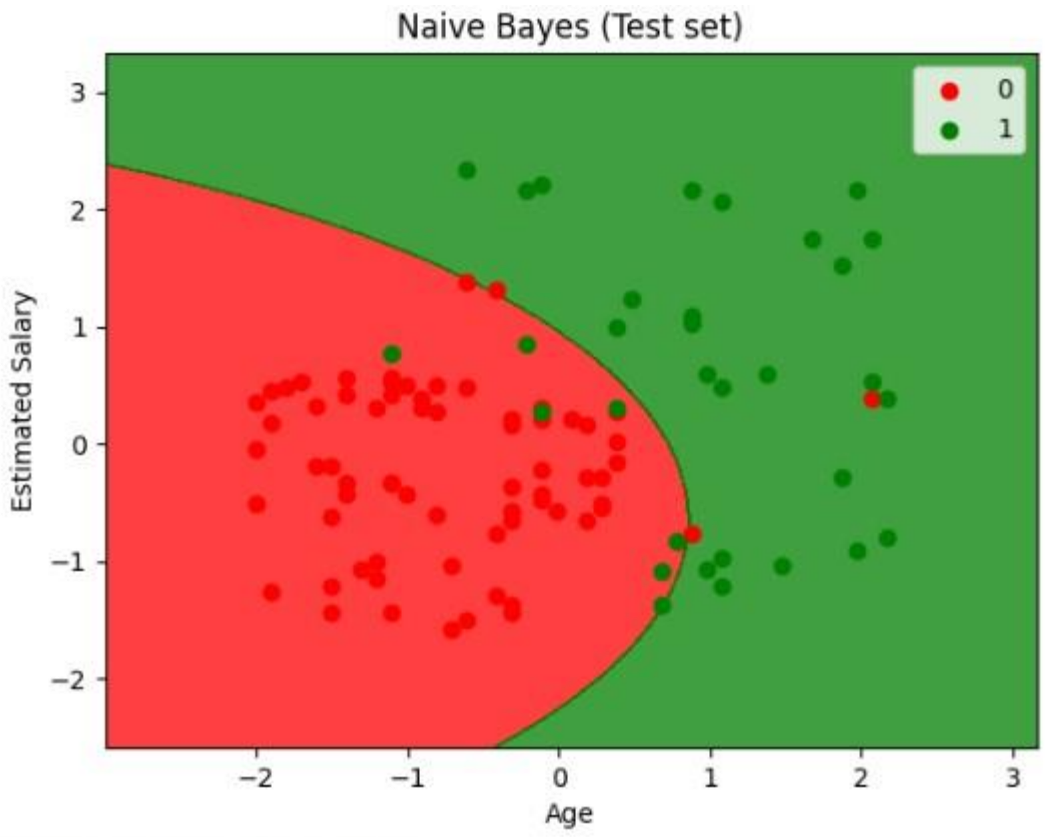
```
# Making the Confusion Matrix from
sklearn.metrics import
confusion_matrix cm =
confusion_matrix(y_test, y_pred)


# Visualising the Training set
results from matplotlib.colors
import ListedColormap
X_set, y_set = X_train, y_train
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1,
stop = X_set[:,
0].max() + 1, step = 0.01),
np.arange(start = X_set[:, 1].min() - 1, stop = X_set[:,
1].max() +
1, step = 0.01))
plt.contourf(X1,      X2,    classifier.predict(np.array([X1.ravel(),
X2.ravel()]).T).reshape(X1.shape),
             alpha = 0.75, cmap =
ListedColormap(('red', 'green')))
plt.xlim(X1.min(), X1.max()) plt.ylim(X2.min(),
X2.max()) for i, j in
enumerate(np.unique(y_set)):
    plt.scatter(X_set[y_set == j, 0],
X_set[y_set == j, 1],                c =
ListedColormap(('red', 'green'))(i), label =
j) plt.title('Naive Bayes (Training set)')
plt.xlabel('Age') plt.ylabel('Estimated
Salary')

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

```
# Visualising the Test set results
from matplotlib.colors import
ListedColormap
X_set, y_set = X_test, y_test
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1,
stop = X_set[:,
0].max() + 1, step = 0.01),
np.arange(start = X_set[:, 1].min() - 1, stop = X_set[:,
1].max() +
1, step = 0.01))
plt.contourf(X1,     X2,    classifier.predict(np.array([X1.ravel(),
X2.ravel()]).T).reshape(X1.shape),
alpha = 0.75, cmap = ListedColormap(('red',
'green'))) plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max()) for i, j in
enumerate(np.unique(y_set)):
    plt.scatter(X_set[y_set == j, 0],
X_set[y_set == j, 1],                 c =
ListedColormap(('red', 'green'))(i), label =
j) plt.title('Naive Bayes (Test set)')
plt.xlabel('Age') plt.ylabel('Estimated
Salary') plt.legend() plt.show()
```

Output:-



Naive Bayes (Training set)
0
1
Estimated Salary
Age



Naive Bayes (Test set)
0
1
Estimated Salary
Age

Practical 2

Aim: - Write a program to implement Decision Tree and Random Forest with Prediction, Test Score and Confusion Matrix.

Theory: -

In machine learning, **Decision Trees** and **Random Forests** are two powerful and widely-used algorithms for both classification and regression tasks. They are based on tree structures, where decisions are made based on the value of input features.

A **Decision Tree** is a flowchart-like structure where each internal node represents a decision based on a feature, each branch represents the outcome of that decision, and each leaf node represents a class label or a continuous value (in case of regression).

-**Recursive Splitting**: The dataset is split based on certain feature conditions to create "branches."

-**Impurity Measures**: At each split, an impurity measure like **Gini Index** or **Entropy (Information Gain)** is used to determine the best split, i.e., the feature that best separates the data.

-**Leaf Nodes**: The final nodes represent the predicted outcome (class or value).

A **Random Forest** is an ensemble learning method that builds multiple decision trees and combines their predictions. It improves upon decision trees by reducing overfitting and increasing accuracy.

Once a model is trained using a decision tree or random forest, its performance is evaluated on test data. The most common evaluation metrics include **accuracy**, **confusion matrix**, and **other metrics** like precision and recall.

A **confusion matrix** is used to visualize the performance of a classification model by comparing the actual labels with predicted labels.

It consists of 4 key terms:

- **True Positive (TP)**: Correctly predicted positive class.
- **True Negative (TN)**: Correctly predicted negative class.
- **False Positive (FP)**: Incorrectly predicted as positive.
- **False Negative (FN)**: Incorrectly predicted as negative.

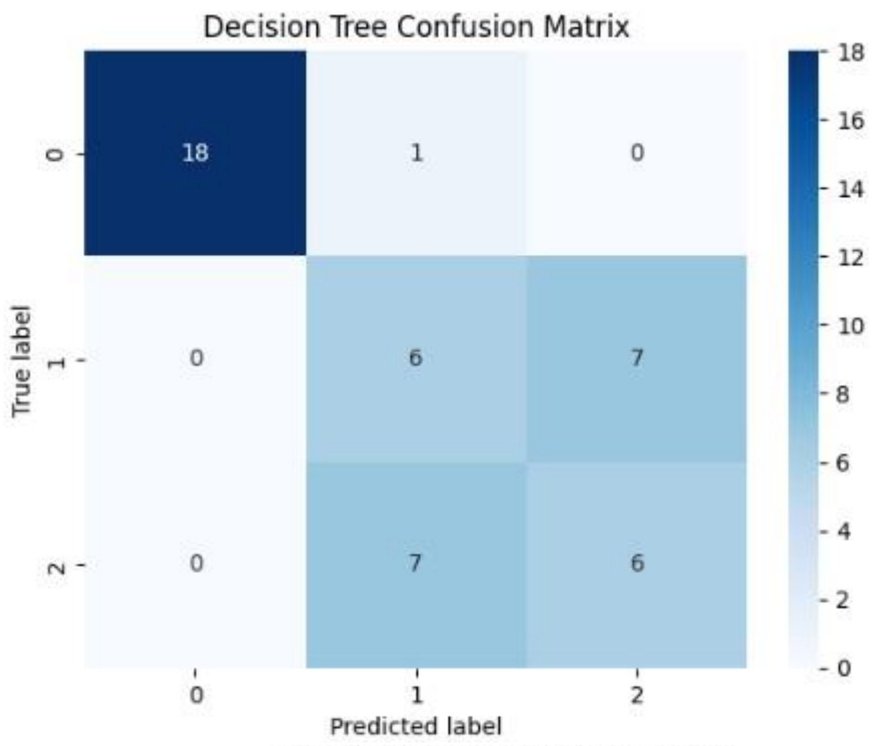
Code: -

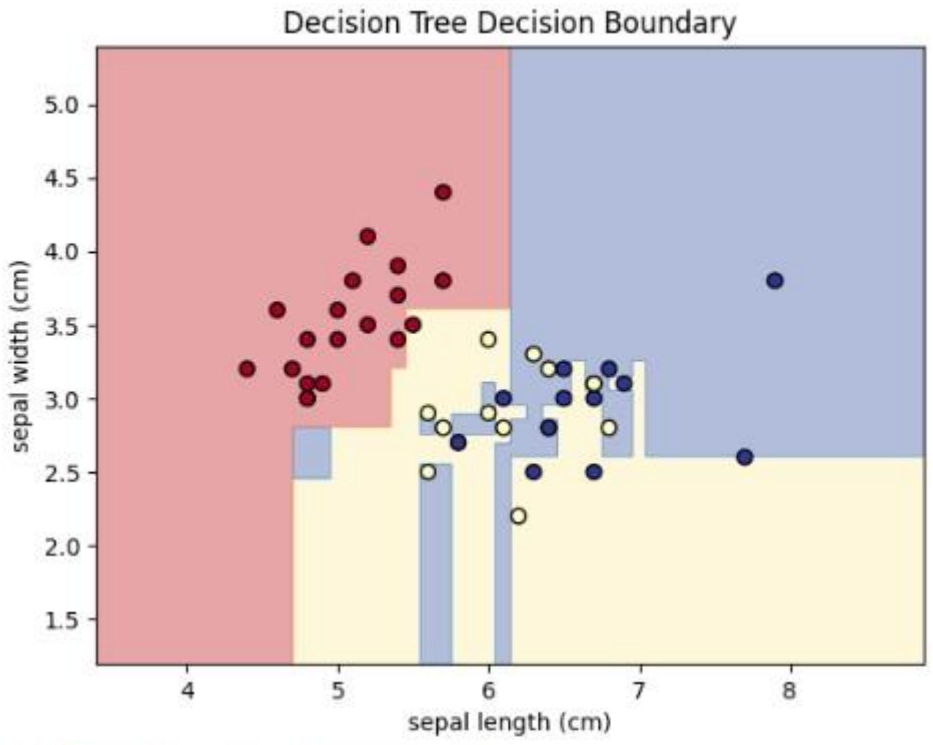
```
#Import necessary libraries import pandas as pd import numpy as
np from sklearn.model_selection import train_test_split from
sklearn.tree import DecisionTreeClassifier from
sklearn.ensemble import RandomForestClassifier from
sklearn.metrics   import   accuracy_score,   confusion_matrix,
classification_report   import   seaborn   as   sns   import
matplotlib.pyplot as plt from sklearn.datasets import load_iris


#Load the dataset into choose only two features for visualization
(we'll use the  first two) iris = load_iris()
X = pd.DataFrame(iris.data, columns=iris.feature_names).iloc[:,
:2]  #Only  first  two  features  y  =  pd.DataFrame(iris.target,
columns=['species'])


#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)


#Function to plot decision
boundary def
plot_decision_boundary(clf, X, y,
title):
    #Create a meshgrid
    x_min, x_max = X.iloc[:,0].min()-
1,X.iloc[:,0].max() + 1     y_min, y_max =
X.iloc[:,1].min()-1,X.iloc[:,1].max() + 1     xx,
yy = np.meshgrid(np.arange(x_min, x_max, 0.01),
np.arange(y_min, y_max, 0.01))

    #Predict for the entire grid
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    #Plot the contors and trainig points
plt.contourf(xx, yy, Z, alpha=0.4,
cmap=plt.cm.RdYlBu)
```

```
plt.scatter(X.iloc[:,0],X.iloc[:,1],c=y.values.ravel(),s=40,ed
gecolor='k',cmap=plt
.cm.RdYlBu)
    plt.title(title)
plt.xlabel(iris.feature_names[0])
plt.ylabel(iris.feature_names[1])
plt.show()
#Desion   tree   classifier   dt_model   =
DecisionTreeClassifier(random_state=42)
dt_model.fit(X_train, y_train)

dt_predictions = dt_model.predict(X_test) #Make
predictions with Decision Tree dt_accuracy =
accuracy_score(y_test, dt_predictions)
dt_confusion_matrix = confusion_matrix(y_test,
dt_predictions)

#Plot Confusion Matrix for Decision Tree
sns.heatmap(dt_confusion_matrix, annot=True, fmt='d',
cmap='Blues') plt.title('Decision Tree Confusion
Matrix') plt.ylabel('True label')
plt.xlabel('Predicted label') plt.show()

#Plot Decison Boundary for Decision Tree
plot_decision_boundary(dt_model,X_test,y_test,'Decision
Tree      Decision Boundary')

#Random      Forest      Classifier      rf_model      =
RandomForestClassifier(n_estimators=100,  random_state=42)
rf_model.fit(X_train, y_train.values.ravel())

#Make predictions with Random Forest
rf_predictions = rf_model.predict(X_test)
```

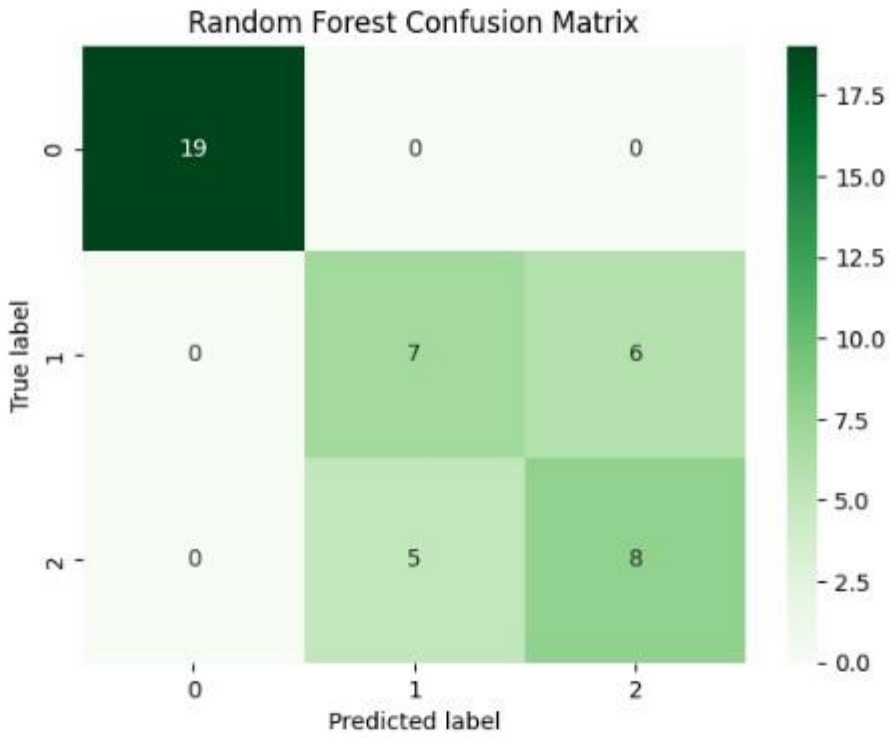
```
rf_confusion_matrix = confusion_matrix(y_test, rf_predictions)
#Random Forest Accuracy and Confusion Matrix
rf_accuracy = accuracy_score(y_test,
rf_predictions)

print(f"Random Forest Accuracy: {rf_accuracy}")
print("Random Forest Classification Report:")
print(classification_report(y_test,
rf_predictions))

#PLot Confusion Matrix for Random Forest
sns.heatmap(rf_confusion_matrix, annot=True, fmt='d',
cmap='Greens') plt.title('Random Forest Confusion Matrix')
plt.ylabel('True label')
plt.xlabel('Predicted label')
plt.show()

#plot Descision Boundary for Random Forest
plot_decision_boundary(rf_model, X_test, y_test, "Random
Forest Decision Boundary")
```

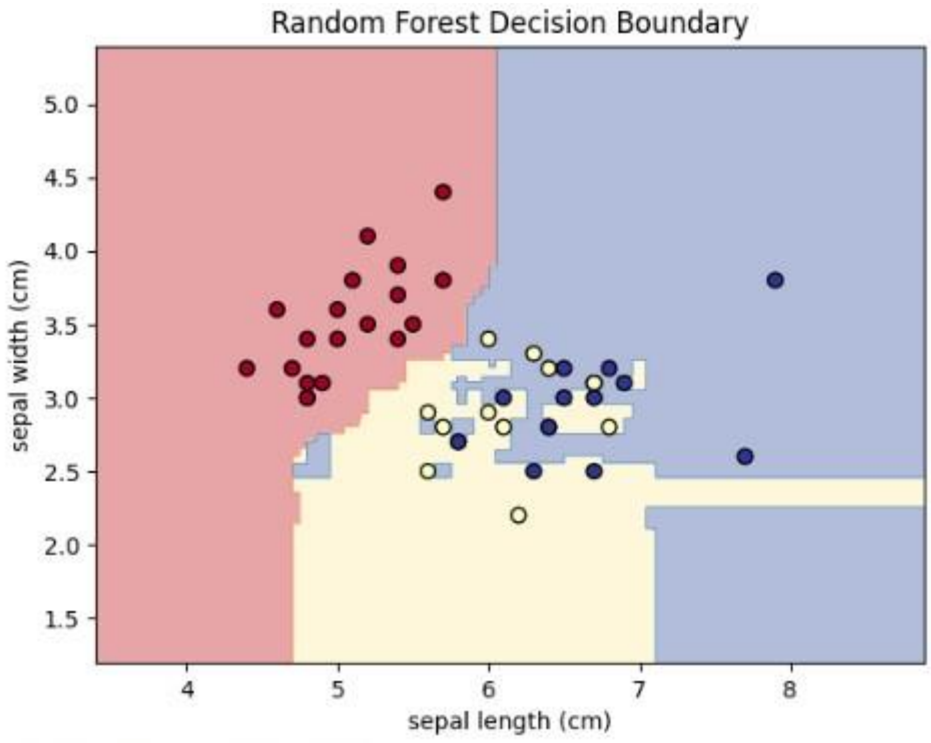
Output: -





```
Random Forest Accuracy: 0.7555555555555555
Random Forest Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        19
           1       0.58      0.54      0.56        13
           2       0.57      0.62      0.59        13

    accuracy                           0.76        45
   macro avg       0.72      0.72      0.72        45
weighted avg       0.76      0.76      0.76        45
```


Random Forest Confusion Matrix
True label
Predicted label
19 0 0
0 7 6
0 5 8
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0


Random Forest Decision Boundary
sepal width (cm)
sepal length (cm)

Practical 3

Aim: - For a given set of training data examples stored in a .CSV file implement Least Square Regression algorithm.

Theory: -

The **Least Squares Regression** algorithm finds the line (or curve) that best represents the relationship between the variables by minimizing the sum of the squared differences (or residuals) between the observed values and the values predicted by the model. These residuals represent the errors between the actual values and the values predicted by the regression line.

Mathematical Formulation:

For a simple linear regression model with one predictor variable:

y=β0+β1x+ϵ Where:

- yis the dependent variable (target).
- x is the independent variable (predictor).
- β0 is the intercept of the regression line.
- β1 is the slope of the regression line.
- ϵ is the error term (residual).

The objective of least squares regression is to minimize the **sum of squared residuals**:

$$\min \sum_{i=1}^{n} (y_i - \hat{y_i})^2$$

Where:

- yi is the actual value.
- yi^ is the predicted value.
- n is the number of data points.

This optimization problem gives us the values of β0 and β1that minimize the error. Steps followed are initialization, compute residuals, minimize sum of Squared Residuals, Prediction.

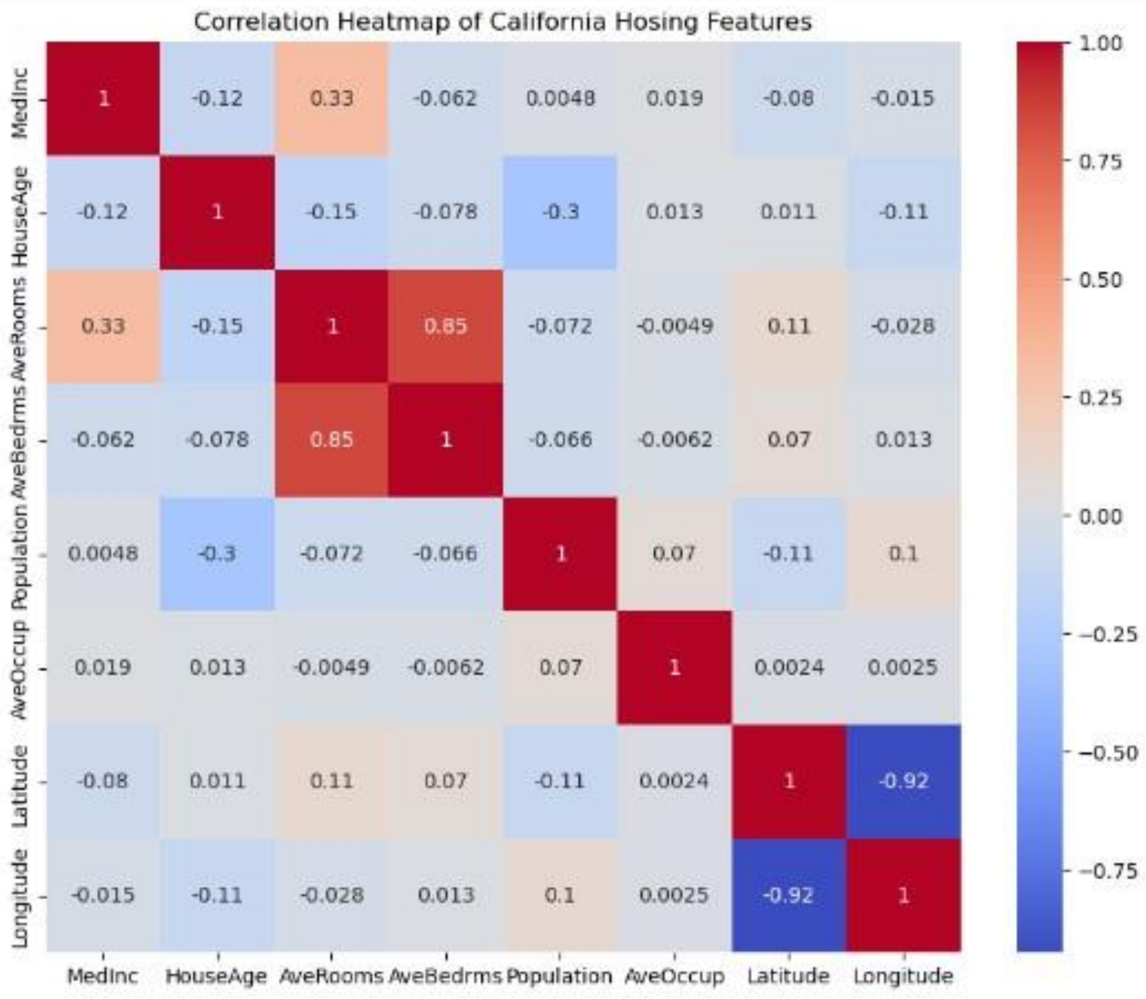
Code: -

```
#Import necessary
libraries import
numpy as np import
pandas as pd import
matplotlib.pyplot as
plt import seaborn
as sns
from sklearn.model_selection import
train_test_split from
sklearn.linear_model import
LinearRegression from sklearn.metrics
import mean_squared_error, r2_score
from sklearn.datasets import
fetch_california_housing

#Load California Hosuing
dataset housing =
fetch_california_housing()
X = pd.DataFrame(housing.data,
columns=housing.feature_names) y =
pd.DataFrame(housing.target, columns=['MEDV'])

#Visualise dataset correlation
heatmap plt.figure(figsize=(10, 8))
sns.heatmap(X.corr(), annot=True,
cmap='coolwarm') plt.title("Correlation Heatmap
of California Hosing Features") plt.show()

#Split the dataset into training
and testingf sets
```

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

#Implement Least Squares Regression (Linear
Regression) reg_model = LinearRegression()
reg_model.fit(X_train, y_train)



#Make predictions
y_train_pred =
reg_model.predict(X_train)
y_test_pred =
reg_model.predict(X_test)

#Generate relevant metrics
train_mse =
mean_squared_error(y_train,
y_train_pred) test_mse =
mean_squared_error(y_test,
y_test_pred) train_r2 =
r2_score(y_train, y_train_pred)
test_r2 = r2_score(y_test,
y_test_pred)

 #Print the results

print(f"Training Mean Squared
Error: {train_mse}") print(f"Test
Mean Squared Error: {test_mse}")
print(f"Training R^2 Score:
```

```
{train_r2}") print(f"Test R^2
Score: {test_r2}")


# Visualise regression coefficients
coefficients = pd.DataFrame(reg_model.coef_.T, X.columns,
columns=['Coefficient']) print(coefficients)



#Plot predicted vs actual values
for test set
plt.figure(figsize=(8,6))
plt.scatter(y_test, y_test_pred,
c='blue')
plt.plot([y_test.min(), y_test.max()], [y_test.min(),
y_test.max()], '--r', lw=3) plt.xlabel("Actual Values")
plt.ylabel("Predicted Values")
plt.title("Actual vs Predicted
Values (Test Set)") plt.show()
```

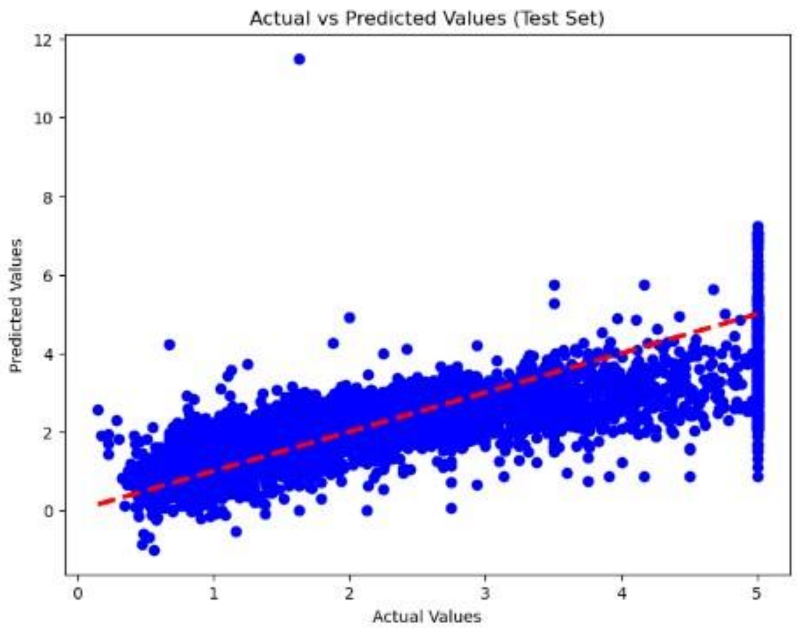
Output: -



```
Training Mean Squared Error: 0.5233576288267754
Test Mean Squared Error: 0.5305677824766757
Training R^2 Score: 0.609345972797216
Test R^2 Score: 0.595770232606166
            Coefficient
MedInc       4.458226e-01
HouseAge     9.681868e-03
AveRooms    -1.220951e-01
AveBedrms    7.785996e-01
Population  -7.757404e-07
AveOccup    -3.370027e-03
Latitude    -4.185367e-01
Longitude   -4.336880e-01
```

```
         MedInc    HouseAge  AveRooms AveBedrms Population  AveOccup   Latitude   Longitude
```

```
Training Mean Squared Error: 0.5233576288267754
Test Mean Squared Error: 0.5305677824766757
Training R^2 Score: 0.609345972797216
Test R^2 Score: 0.595770232606166
             Coefficient
MedInc      4.458226e-01
HouseAge    9.681868e-03
AveRooms   -1.220951e-01
AveBedrms   7.785996e-01
Population -7.757404e-07
AveOccup   -3.370027e-03
Latitude   -4.185367e-01
Longitude  -4.336880e-01
```



Practical 4

Aim: - For a given set of training data examples stored in a .CSV file implement Logistic Regression algorithm.

Theory: -

Logistic regression models the probability that a given input belongs to a particular class. It estimates the **log odds** of the dependent variable being 1 (positive class) rather than 0 (negative class). The output of logistic regression is a probability value between 0 and 1, which is then used to classify the data points.

Logistic Function (Sigmoid Function):

The key to logistic regression is the **logistic function**, also known as the **sigmoid function**, which maps any real-valued number to the range (0, 1). The formula for the logistic function is:

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

Where:

- z is the linear combination of the input features (like in linear regression),
  i.e., z= β0+β1x1+β2x2+...+βnxn.
- e is Euler's number (approximately 2.718).

This function ensures that the predicted value is between 0 and 1, representing a probability. **Logistic Regression** is a powerful, simple, and effective algorithm for binary and multi-class classification tasks. It models the relationship between input features and the probability of a particular outcome using the logistic (sigmoid) function. Despite its simplicity, it works well in many practical scenarios and serves as a good baseline model for classification tasks.

Code: -

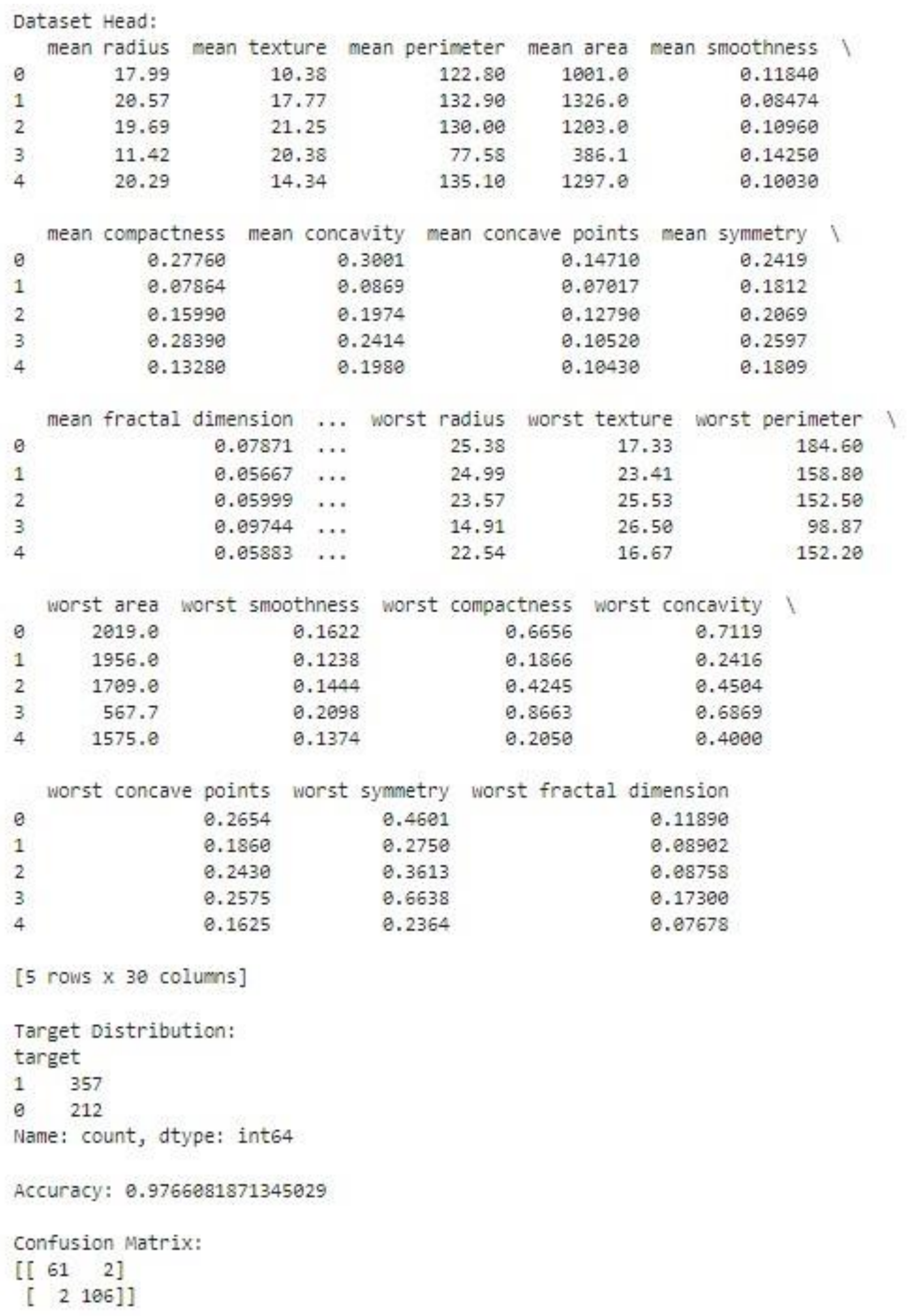
```
#Import necessary
libraries import pandas
as pd import numpy as
np import
matplotlib.pyplot as
plt import seaborn as
sns from
sklearn.model_selection
import train_test_split
from
```

```
sklearn.linear_model
import
LogisticRegression from
sklearn.metrics
import
accuracy_score,
confusion_matrix,
classification_report
from sklearn.datasets
import
load_breast_cancer

#Step 1: Load the inbulit Breast Cancer
dataset cancer_data = load_breast_cancer()
X = pd.DataFrame(cancer_data.data,
columns=cancer_data.feature_names)
#Feature y = pd.DataFrame(cancer_data.target,
columns=['target']) # Target

#Step 2: Explore the dataset print("Dataset
Head:") print(X.head()) # Preview the first few
rows of the feature set

print("\nTarget Distribution:")
print(y['target'].value_counts()) # Distribution of the target
variable (0 = malignant 1= benign)

#Step 3: Split the databaset into traning and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3, random_state=42)

#Step 4 : Implement Logistic Regression logreg =
LogisticRegression(max_iter=10000, random_state=42) #Incresed
max_itr to ensure convergence
```

```
logreg.fit(X_train, y_train.values.ravel()) # y_train must be
passed as a flat aaray


#Step 5 ; Make predictions on the test
set y_pred = logreg.predict(X_test)


#Step 6: Evaluate the model accuracy =
accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test,
y_pred) class_report =
classification_report(y_test, y_pred)


print(f"\nAccuracy: {accuracy}")
print("\nConfusion Matrix:")
print(conf_matrix)
print("\nClassification
Report:") print(class_report)


# Step 7: Visualize the Confusion Matrix
plt.figure(figsize=(6,4))
sns.heatmap(conf_matrix, annot=True, fmt='d',
cmap='Blues') plt.title('Confusion Matric')
plt.ylabel('True Label') plt.xlabel('Predicted
Label') plt.show()


#Step 8:Make a prediction on a  new input sample
#Example:Let's create a new sample input (using the mean of
feature for simplicity)
#You can replace these values with actual feature values you'd
like to predict for


new_input = np.array([X.mean().values])
```

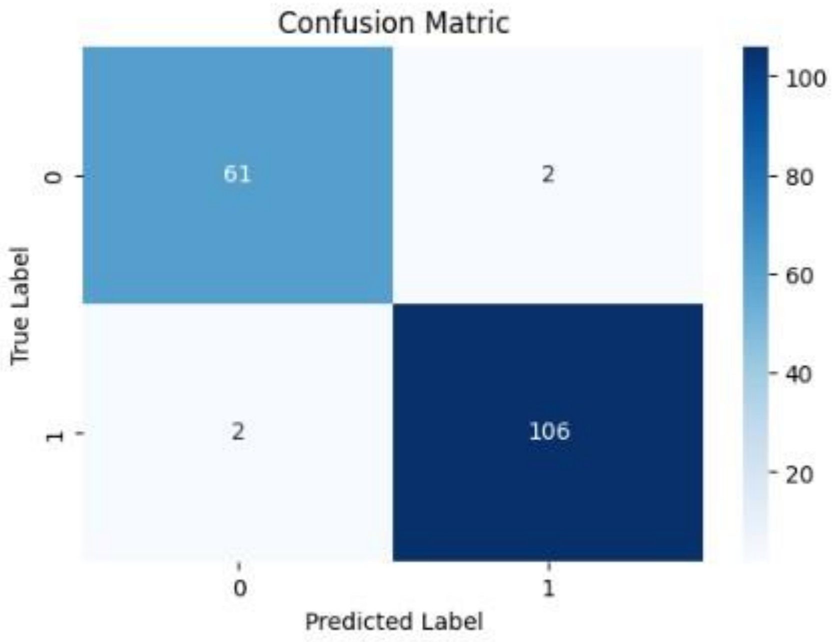
```
# Ensure the new input has the correct shape (1,
n_features) print(f"\nNew Input for
Prediction:\n{new_input}")

#Make a prediction on the new input
new_prediction =
logreg.predict(new_input)

#Get the predictied class (0= malignant, 1=bening)
predicted_class = 'bening' if new_prediction == 1
else 'malignant'

print(f"\nPredited class for the new input:
{predicted_class}")

#Step 9 : Visualise the Confusion Matrix for the
test set plt.figure(figsize=(6, 4))
sns.heatmap(conf_matrix, annot=True, fmt='d',
cmap='Blues') plt.title('Confusion Matrix- Test
Set') plt.ylabel('True Label')
plt.xlabel('Predicted Label') plt.show()
```

Output: -

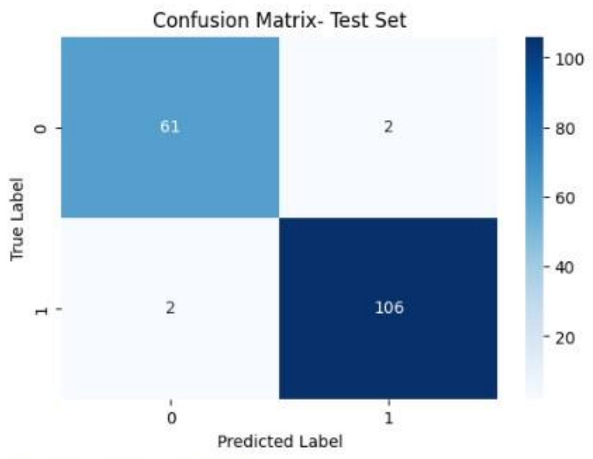


```
Dataset Head:
   mean radius  mean texture  mean perimeter  mean area  mean smoothness  \
0        17.99         10.38          122.80     1001.0          0.11840
1        20.57         17.77          132.90     1326.0          0.08474
2        19.69         21.25          130.00     1203.0          0.10960
3        11.42         20.38           77.58      386.1          0.14250
4        20.29         14.34          135.10     1297.0          0.10030

   mean compactness  mean concavity  mean concave points  mean symmetry  \
0           0.27760          0.3001              0.14710         0.2419
1           0.07864          0.0869              0.07017         0.1812
2           0.15990          0.1974              0.12790         0.2069
3           0.28390          0.2414              0.10520         0.2597
4           0.13280          0.1980              0.10430         0.1809

   mean fractal dimension  ...  worst radius  worst texture  worst perimeter  \
0                 0.07871  ...         25.38          17.33           184.60
1                 0.05667  ...         24.99          23.41           158.80
2                 0.05999  ...         23.57          25.53           152.50
3                 0.09744  ...         14.91          26.50            98.87
4                 0.05883  ...         22.54          16.67           152.20

   worst area  worst smoothness  worst compactness  worst concavity  \
0      2019.0            0.1622             0.6656           0.7119
1      1956.0            0.1238             0.1866           0.2416
2      1709.0            0.1444             0.4245           0.4504
3       567.7            0.2098             0.8663           0.6869
4      1575.0            0.1374             0.2050           0.4000

   worst concave points  worst symmetry  worst fractal dimension
0                0.2654          0.4601                  0.11890
1                0.1860          0.2750                  0.08902
2                0.2430          0.3613                  0.08758
3                0.2575          0.6638                  0.17300
4                0.1625          0.2364                  0.07678

[5 rows x 30 columns]

Target Distribution:
target
1    357
0    212
Name: count, dtype: int64

Accuracy: 0.9766081871345029

Confusion Matrix:
[[ 61   2]
 [  2 106]]
```

```
Classification Report:
              precision    recall  f1-score   support

           0       0.97      0.97      0.97        63
           1       0.98      0.98      0.98       108

    accuracy                           0.98       171
   macro avg       0.97      0.97      0.97       171
weighted avg       0.98      0.98      0.98       171
```



```
New Input for Prediction:
[[1.41272917e+01 1.92896485e+01 9.19690334e+01 6.54889104e+02
  9.63602812e-02 1.04340984e-01 8.87993158e-02 4.89191459e-02
  1.81161863e-01 6.27976098e-02 4.05172056e-01 1.21685343e+00
  2.86605923e+00 4.03370791e+01 7.04097891e-03 2.54781388e-02
  3.18937163e-02 1.17961371e-02 2.05422988e-02 3.79490387e-03
  1.62691898e+01 2.56772232e+01 1.07261213e+02 8.80583128e+02
  1.32368594e-01 2.54265044e-01 2.72188483e-01 1.14606223e-01
  2.90075571e-01 8.39458172e-02]]

Predited class for the new input: malignant
```



Practical 5

Aim: - Write a program to demonstrate the working of the decision tree based ID3 algorithm. Use an appropriate data set for building the decision tree and apply this knowledge to classify a new sample.

Theory: -

ID3 is a decision tree-building algorithm that is based on the concept of **entropy** and **information gain** from information theory. It works by repeatedly splitting the dataset into smaller subsets based on the feature that provides the highest information gain until it reaches leaf nodes, which represent class labels.

At each node in the tree, ID3 selects the feature that best separates the data into the target classes. The algorithm continues this process recursively for each subset of the data, creating a tree structure that can be used for classification.

Entropy is a measure of the uncertainty or impurity in the data. It quantifies how mixed or homogeneous a set of classes is. If all examples in a subset belong to a single class, the entropy is 0 (pure), and if the examples are equally divided among classes, the entropy is 1 (maximum impurity).

$$\text{Entropy}(S) = -\sum_{i=1}^{k} p_i \log_2(p_i)$$

Where:

- pi is the proportion of examples in class iii.
- k is the total number of classes.

Information gain measures how much a feature reduces the entropy (impurity) in a dataset. ID3 chooses the feature with the highest information gain to split the data at each step.

The formula for information gain when splitting on feature A is:

$$\text{Gain}(S, A) = \text{Entropy}(S) - \sum_{v \in \text{Values}(A)} \frac{|S_v|}{|S|} \cdot \text{Entropy}(S_v)$$

Where:

- S is the original set.
- A is the feature being considered for the split.
- Sv is the subset of SSS where the feature A takes value v.
- ∣S_v∣/∣Sv∣ is the proportion of examples in subset Sv

The **ID3 algorithm** is a foundational decision tree algorithm that uses **entropy** and **information gain** to create decision trees for classification problems. By recursively splitting the dataset based on the most informative features, ID3 constructs a tree that can classify new data points. Although simple and easy to understand, ID3 can overfit and is sensitive to noise and features with many values. However, it remains a fundamental algorithm in machine learning and provides the basis for more advanced decision tree algorithms, such as C4.5 and CART.

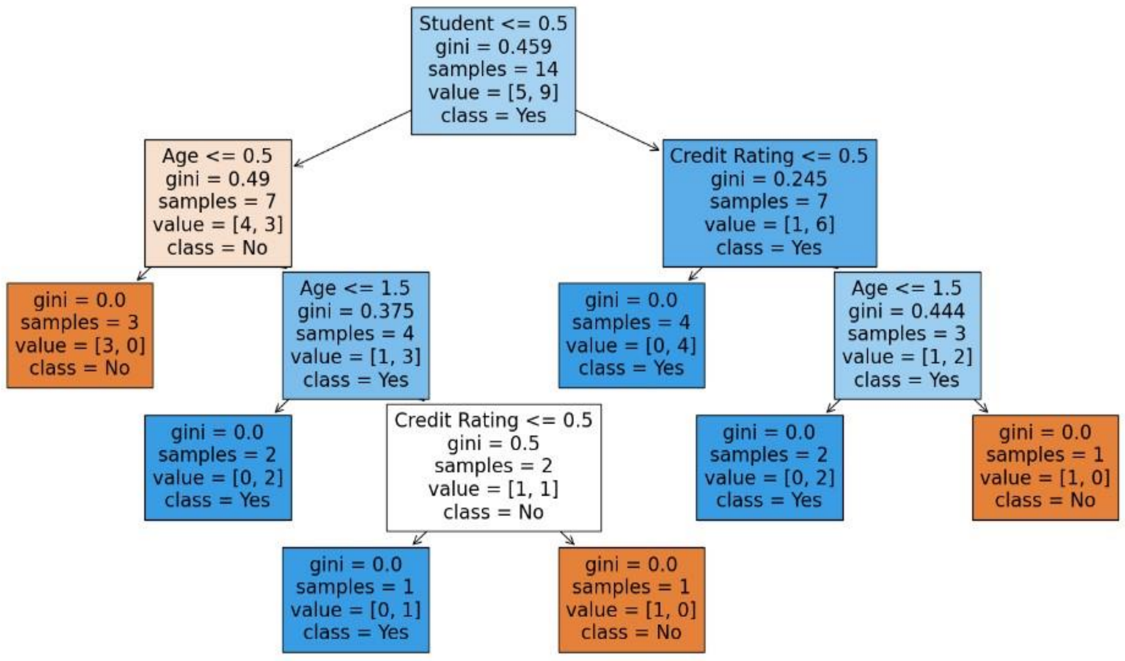
Code: -

```
import pandas as pd import
numpy as np import
matplotlib.pyplot as plt
from sklearn import tree as
sk_tree


# Step 1: Parse the
dataset data = {
    'Age': ['<=30', '<=30', '31-40', '>40', '>40', '>40', '31-
40', '<=30', '<=30', '>40', '<=30', '31-40', '31-40', '>40'],
    'Income': ['High', 'High', 'High', 'Medium', 'Low', 'Low',
'Low', 'Medium', 'Low', 'Medium', 'Medium', 'Medium', 'High',
'Medium'],
    'Student': ['No', 'No', 'No', 'No', 'Yes', 'Yes', 'Yes',
'No', 'Yes', 'Yes', 'Yes', 'No', 'Yes', 'No'],
    'Credit Rating': ['Fair', 'Excellent', 'Fair', 'Fair',
'Fair', 'Excellent', 'Excellent', 'Fair', 'Fair', 'Fair',
'Excellent', 'Excellent', 'Fair', 'Excellent'],
    'Buys Computer': ['No', 'No', 'Yes', 'Yes', 'Yes', 'No',
'Yes', 'No', 'Yes', 'Yes', 'Yes', 'Yes', 'Yes', 'No']
}


df = pd.DataFrame(data)


# Encode the categorical variables
df_encoded = df.apply(lambda x:
pd.factorize(x)[0])
```

```
# Fit the decision tree classifier using Gini impurity
clf_gini = sk_tree.DecisionTreeClassifier(criterion='gini')
clf_gini = clf_gini.fit(df_encoded.iloc[:, :-1],
df_encoded['Buys Computer'])

# Convert the feature names from Index to list
feature_names = df.columns[:-1].tolist()

# Convert the class names to a list
class_names = df['Buys
Computer'].unique().tolist()

# Plot the decision tree plt.figure(figsize=(20,10))
sk_tree.plot_tree(clf_gini,feature_names=feature_names,class_
names=class_na mes, filled=True) plt.show()

# Function to print Gini impurity and chosen attribute
at each split def print_gini_and_splits(tree,
feature_names):
    tree_ = tree.tree_     feature_name = [
feature_names[i]    if   i    !=
sk_tree._tree.TREE_UNDEFINED  else "undefined!"
        for i in tree_.feature
    ]


    print("Decision tree splits and Gini
impurities:")     for i in
range(tree_.node_count):         if
tree_.children_left[i] !=
sk_tree._tree.TREE_LEAF:
            print(f"Node {i} (Gini: {tree_.impurity[i]:.4f}):
split on feature
'{feature_name[i]}'"
)        else:
```

```
            print(f"Node {i} (Gini: {tree_.impurity[i]:.4f}):
leaf node")

print_gini_and_splits(clf_gini, feature_names)

# Example test sample
test_sample = {
    'Age': '<=30',
    'Income': 'Medium',
    'Student': 'Yes',
    'Credit Rating': 'Fair'
}

# Encode the test sample
encoded_sample    = pd.DataFrame([test_sample]).apply(lambda x:
pd.factorize(df[x.name])[0][df[x.name].tolist().index(x[0])])

# Predict using sklearn decision tree
sklearn_prediction =
clf_gini.predict([encoded_sample])
decoded_prediction  =
pd.factorize(df['Buys
Computer'])[1][sklearn_prediction[0]]
print("Prediction  for  sklearn  decision  tree:",
decoded_prediction)
```

Output: -



```
Decision tree splits and Gini impurities:
Node 0 (Gini: 0.4592): split on feature 'Student'
Node 1 (Gini: 0.4898): split on feature 'Age'
Node 2 (Gini: 0.0000): leaf node
Node 3 (Gini: 0.3750): split on feature 'Age'
Node 4 (Gini: 0.0000): leaf node
Node 5 (Gini: 0.5000): split on feature 'Credit Rating'
Node 6 (Gini: 0.0000): leaf node
Node 7 (Gini: 0.0000): leaf node
Node 8 (Gini: 0.2449): split on feature 'Credit Rating'
Node 9 (Gini: 0.0000): leaf node
Node 10 (Gini: 0.4444): split on feature 'Age'
Node 11 (Gini: 0.0000): leaf node
Node 12 (Gini: 0.0000): leaf node
Prediction for sklearn decision tree: Yes
```

Practical 6

Aim: - Write a program to implement k-Nearest Neighbour algorithm to classify the iris data set.

Theory: -

The **k-Nearest Neighbours (k-NN)** algorithm is a simple, non-parametric, and instance-based learning algorithm widely used for **classification** and **regression** tasks. It is one of the most straightforward machines learning algorithms, making decisions by looking at the **'k' closest data points** (neighbours) in the training set and using them to predict the outcome for a new data point.

The k-NN algorithm works by comparing a new data point with its nearest neighbours in the feature space, based on some distance metric, such as **Euclidean distance**. The class of the new point is then determined by the **majority class** of its neighbours (for classification) or by averaging the values of the neighbours (for regression).

The main idea is that **similar data points** (those close to each other) tend to have similar classifications or target values.

Here's a step-by-step process for how the k-NN algorithm works:

1. **Step 1: Choose the value of 'k'**:
   - The number kkk represents the number of nearest neighbors to consider when making a prediction. A small value of kkk can be more sensitive to noise, while a larger value smooths the decision boundary.
2. **Step 2: Calculate Distance**:
   - For each new data point, the algorithm calculates the distance between the new point and all points in the training dataset. Common distance metrics include:
     - **Euclidean Distance** (for continuous variables):

$$d(p,q) = \sqrt{\sum_{i=1}^{n}(p_i - q_i)^2}$$

     - **Manhattan Distance**: Measures distance as the sum of absolute differences across dimensions.
3. **Step 3: Identify Neighbors**:
   - The algorithm identifies the k data points in the training set that are closest to the new data point based on the calculated distances.
4. **Step 4: Voting (for Classification)**:
   - The algorithm assigns the new data point to the most common class among the k-nearest neighbors.

- **Majority Voting**: If more neighbors belong to a particular class, the new point is assigned that class.

For Regression:

- The algorithm averages the target values of the k-nearest neighbors to predict a continuous value.

5. **Step 5: Make Prediction**:
    - Based on the voting (classification) or averaging (regression), the algorithm predicts the class label or continuous value for the new data point.

The **k-Nearest Neighbors (k-NN)** algorithm is a simple yet powerful machine learning method used for classification and regression tasks. By relying on the concept that similar data points tend to belong to the same class or have similar values, k-NN makes predictions by looking at the k-nearest data points. Although simple and effective for small datasets, k-NN can be computationally intensive for large datasets and sensitive to irrelevant features, which necessitates careful data preprocessing and selection of the appropriate value for k.

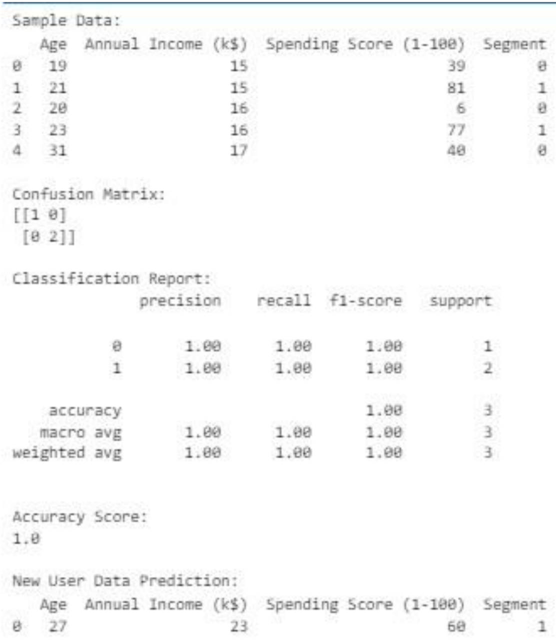
Code: -

```
# Step 1: Import necessary libraries
import pandas as pd import
matplotlib.pyplot as plt import seaborn
as sns from sklearn.model_selection
import train_test_split from
sklearn.preprocessing import
StandardScaler from sklearn.neighbors
import KNeighborsClassifier from
sklearn.metrics     import
classification_report,
confusion_matrix, accuracy_score from
mpl_toolkits.mplot3d import Axes3D


# Step 2: Load and display the sample
data data = {
    'Age': [19, 21, 20, 23, 31, 22, 35, 25, 23, 64, 30, 67,
35, 58, 24],
    'Annual Income (k$)': [15, 15, 16, 16, 17, 17, 18, 18, 19,
19, 20, 20, 21, 21, 22],
    'Spending Score (1-100)': [39, 81, 6, 77, 40, 76, 6, 94,
3, 72, 79, 65, 76, 76, 94],

    'Segment': [0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1]
# 0: Low-value, 1: Highvalue

}


df = pd.DataFrame(data)

print("Sample Data:")

print(df.head())


# Step 3: Data Preprocessing

X = df[['Age', 'Annual Income (k$)', 'Spending

Score (1-100)']] y = df['Segment']


scaler = StandardScaler()

X_scaled =

scaler.fit_transform(X)


# Step 4: Train-Test Split

X_train, X_test, y_train, y_test = train_test_split(X_scaled,
y, test_size=0.2, random_state=42)


# Step 5: Apply KNN Algorithm knn =

KNeighborsClassifier(n_neighbors=3

) knn.fit(X_train, y_train) y_pred

= knn.predict(X_test)


# Step 6: Evaluation

print("\nConfusion Matrix:")

print(confusion_matrix(y_test,

y_pred)) print("\nClassification

Report:")

print(classification_report(y_tes

t, y_pred)) print("\nAccuracy
```

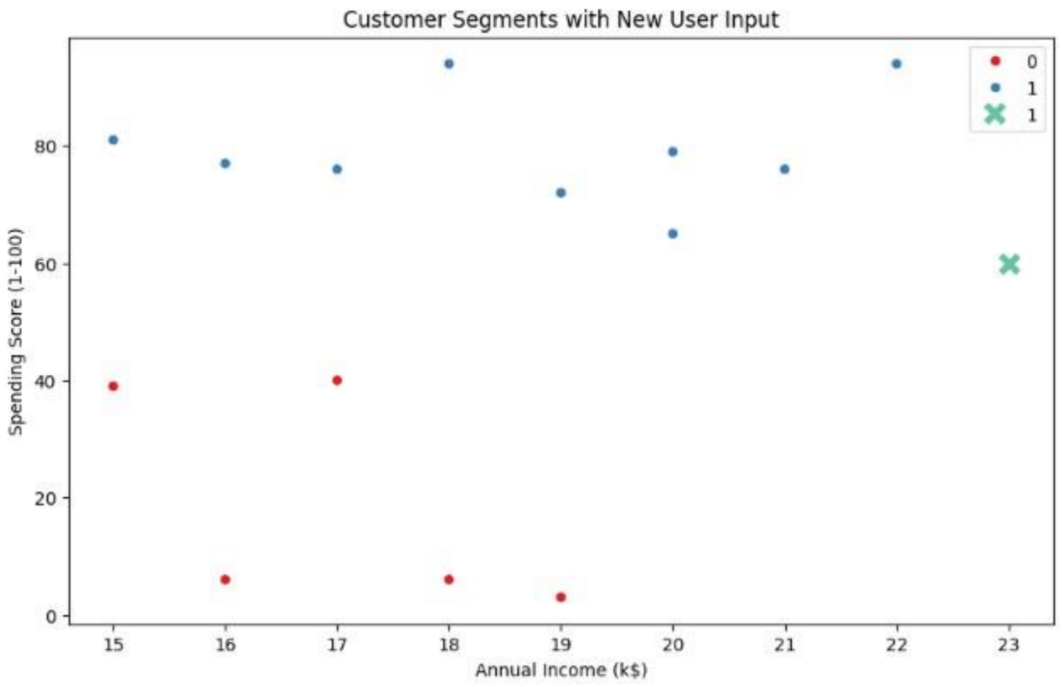
```
Score:")
print(accuracy_score(y_test,
y_pred))

# Step 7: Classify new user input new_user_data = {'Age':
[27], 'Annual Income (k$)': [23], 'Spending Score (1-
100)': [60]} new_user_df =
pd.DataFrame(new_user_data)
new_user_scaled =
scaler.transform(new_user_df)

new_user_segment =
knn.predict(new_user_scaled)
new_user_df['Segment'] = new_user_segment
print("\nNew User Data Prediction:")
print(new_user_df)

# Visualization: Scatter plot of the customer segments
plt.figure(figsize=(10, 6))
sns.scatterplot(x='Annual Income (k$)', y='Spending Score (1-
100)', hue='Segment', data=df, palette='Set1', marker='o')#,
label='Existing Data' sns.scatterplot(x='Annual Income (k$)',
y='Spending Score (1-100)', hue='Segment', data=new_user_df,
palette='Set2', marker='X', s=200)#, label='New User Data'
plt.title('Customer Segments with New User Input')
plt.xlabel('Annual Income (k$)') plt.ylabel('Spending Score (1-
100)') plt.legend() plt.show()

# Visualization: 3D plot for KNN decision boundaries and customer
segments including new user input fig = plt.figure(figsize=(10,
6)) ax = fig.add_subplot(111, projection='3d')
```

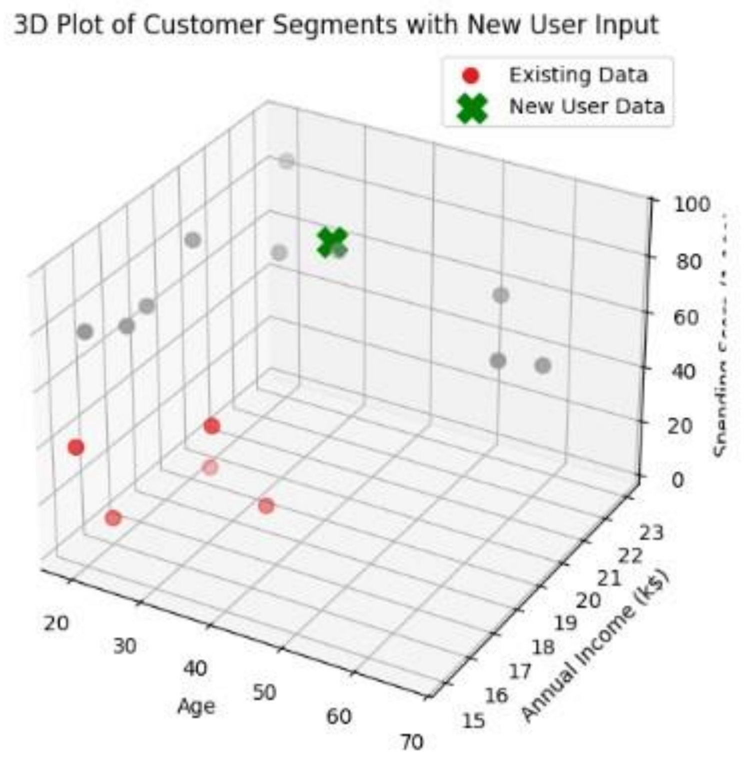
```
# Plot the existing data with original values
ax.scatter(X['Age'], X['Annual Income (k$)'], X['Spending Score (1-100)'], c=y, cmap='Set1', s=50, label='Existing Data')


# Plot the new user input with original values
ax.scatter(new_user_df['Age'], new_user_df['Annual Income (k$)'], new_user_df['Spending Score (1-100)'], c='green', marker='X', s=200, label='New User Data') ax.set_xlabel('Age') ax.set_ylabel('Annual Income (k$)') ax.set_zlabel('Spending Score (1-100)') plt.title('3D Plot of Customer Segments with New User Input') ax.legend()

plt.show()
```

Output: -



```
Sample Data:
   Age  Annual Income (k$)  Spending Score (1-100)  Segment
0   19                  15                      39        0
1   21                  15                      81        1
2   20                  16                       6        0
3   23                  16                      77        1
4   31                  17                      40        0

Confusion Matrix:
[[1 0]
 [0 2]]

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00         1
           1       1.00      1.00      1.00         2

    accuracy                           1.00         3
   macro avg       1.00      1.00      1.00         3
weighted avg       1.00      1.00      1.00         3


Accuracy Score:
1.0

New User Data Prediction:
   Age  Annual Income (k$)  Spending Score (1-100)  Segment
0   27                  23                      60        1
```




3D Plot of Customer Segments with New User Input
Existing Data
New User Data
Age
Annual Income (k$)
Spending Score
20
30
40
50
60
70
15
16
17
18
19
20
21
22
23
0
20
40
60
80
100

Practical 7

Aim: - Implement the different Distance methods (Euclidean) with Prediction, Test Score and Confusion Matrix.

Theory: -

In machine learning, particularly for algorithms like **k-Nearest Neighbours (kNN)**, **Clustering** (e.g., k-means), and **distance-based anomaly detection**, the choice of distance metrics is critical. The distance metric determines how "close" or "far" two data points are from each other, influencing the model's predictions.

Different tasks and types of data require different distance measures, and the **Euclidean distance** is one of the most commonly used, but it is not always the best. In this introduction, we'll explore **Euclidean distance**, as well as other commonly used distance methods such as **Manhattan**, **Minkowski**, and **Cosine Similarity**.

Euclidean distance is the most widely used distance metric, particularly for continuous, real-valued data. It represents the straight-line (or shortest) distance between two points in a multi-dimensional space.

The formula for **Euclidean Distance** between two points p=(p1,p2,...,pn)p = and q=(q1,q2,...,qn) in an n-dimensional space is:

$$d(p,q) = \sqrt{\sum_{i=1}^{n}(p_i - q_i)^2}$$

From the confusion matrix, we can calculate the following metrics:

- **Accuracy** = $\frac{TP+TN}{TP+TN+FP+FN}$
- **Precision** = $\frac{TP}{TP+FP}$
- **Recall** $\frac{TP}{TP+FN}$ (Sensitivity) =
- **F1 Score** = Harmonic mean of precision and recall:
  $2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$

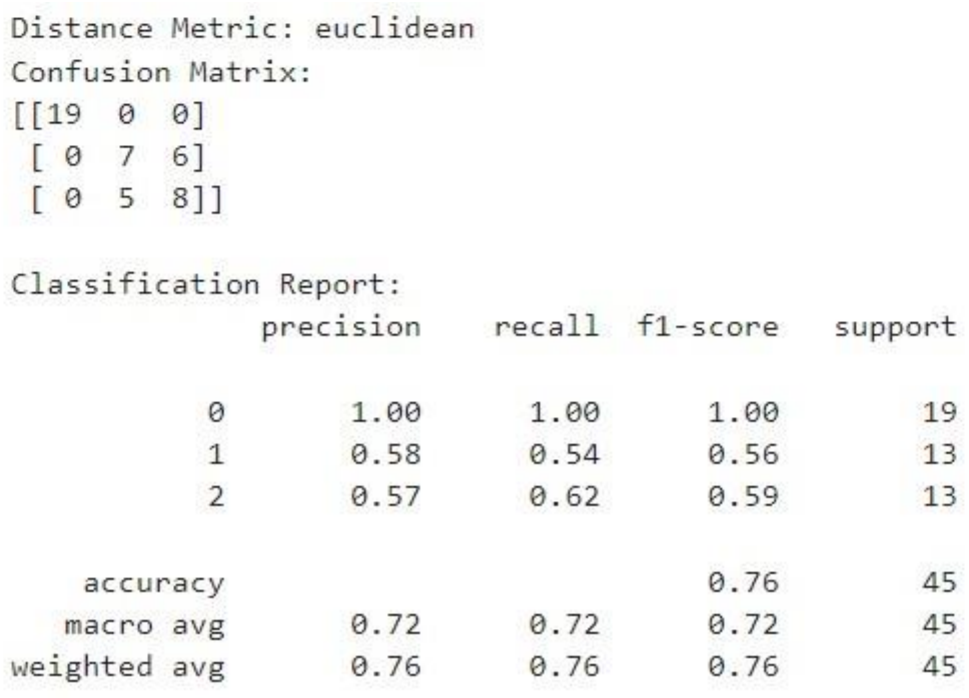
Code: -

```
import numpy as np import pandas as pd import
matplotlib.pyplot as plt from sklearn.datasets
import load_iris from sklearn.model_selection import
train_test_split from sklearn.neighbors import
```

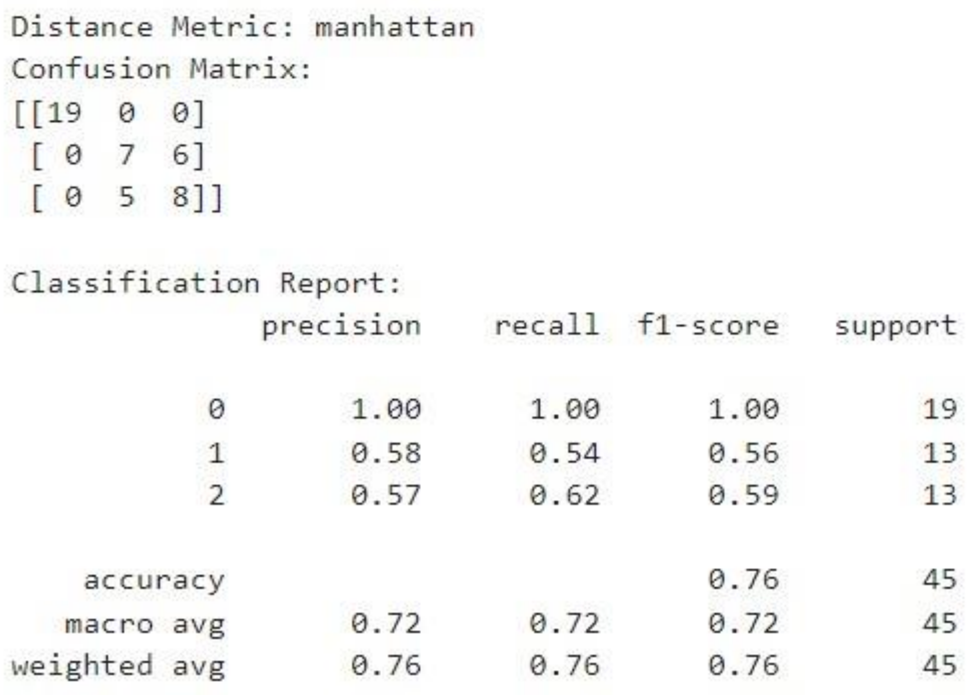
```
KNeighborsClassifier from sklearn.metrics import
classification_report,confusion_matrix


# Load the Iris
dataset iris =
load_iris()
X = iris.data[:, :2]  # Select only the first two features (sepal
length and sepal width) y = iris.target


# 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)
# Initialize k-NN classifier with different
distance metrics k = 3


# List of distance metrics to test
distance_metrics = ['euclidean',
'manhattan', 'chebyshev']


# Create subplots for each distance metric fig,
axes = plt.subplots(1, len(distance_metrics),
figsize=(15, 5)) for i, metric in
enumerate(distance_metrics):
    knn_classifier = KNeighborsClassifier(n_neighbors=k,
metric=metric)
    # Fit the classifier to the
training data
knn_classifier.fit(X_train,
y_train)     # Make predictions on
the test data     y_pred =
knn_classifier.predict(X_test)
# Evaluate the classifier's
performance     print(f"Distance
Metric: {metric}")
```

```
print("Confusion Matrix:")
print(confusion_matrix(y_test,
y_pred))
print("\nClassification Report:")
print(classification_report(y_test,
y_pred))     print("\n")


    # Visualize the dataset and decision boundaries for
the current metric     ax = axes[i]
# Plot the training data points     ax.scatter(X_train[:,
0],  X_train[:,     1],  c=y_train,     cmap='viridis',
label='Training Data')


    # Plot the testing data points     ax.scatter(X_test[:, 0],
X_test[:,  1],  c=y_test,  cmap='viridis',  marker='x',  s=100,
label='Testing Data')
#   Plot   decision   boundaries   using   the   current   metric
knn_classifier      =      KNeighborsClassifier(n_neighbors=k,
metric=metric)     knn_classifier.fit(X, y)     x_min, x_max =
X[:, 0].min() - 1, X[:, 0].max() + 1      y_min, y_max = X[:,
1].min()  -  1,  X[:,  1].max()  +  1           xx,  yy  =
np.meshgrid(np.arange(x_min,  x_max,  0.01),  np.arange(y_min,
y_max, 0.01))

    Z = knn_classifier.predict(np.c_[xx.ravel(),
yy.ravel()])     Z = Z.reshape(xx.shape)
ax.contourf(xx, yy, Z, cmap='viridis', alpha=0.5,
levels=range(4))     ax.set_title(f'K-NN
({metric.capitalize()} Metric)')
ax.set_xlabel('Sepal Length (cm)')
ax.set_ylabel('Sepal Width (cm)')     ax.legend()

plt.show()
```

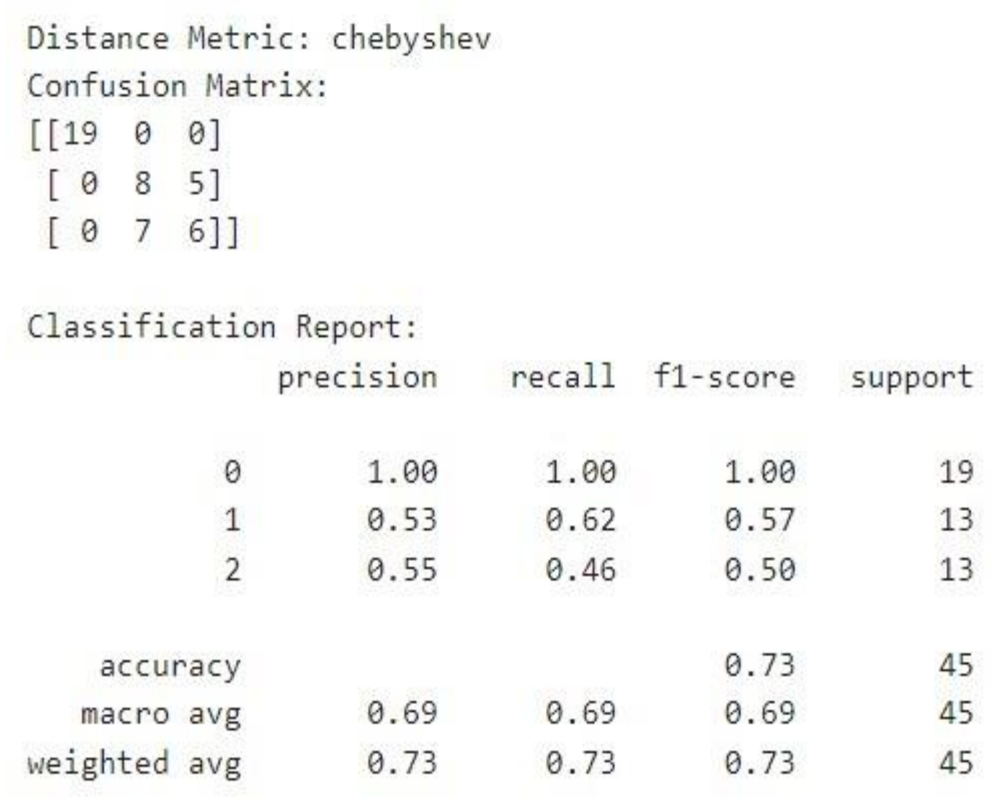
Output: -



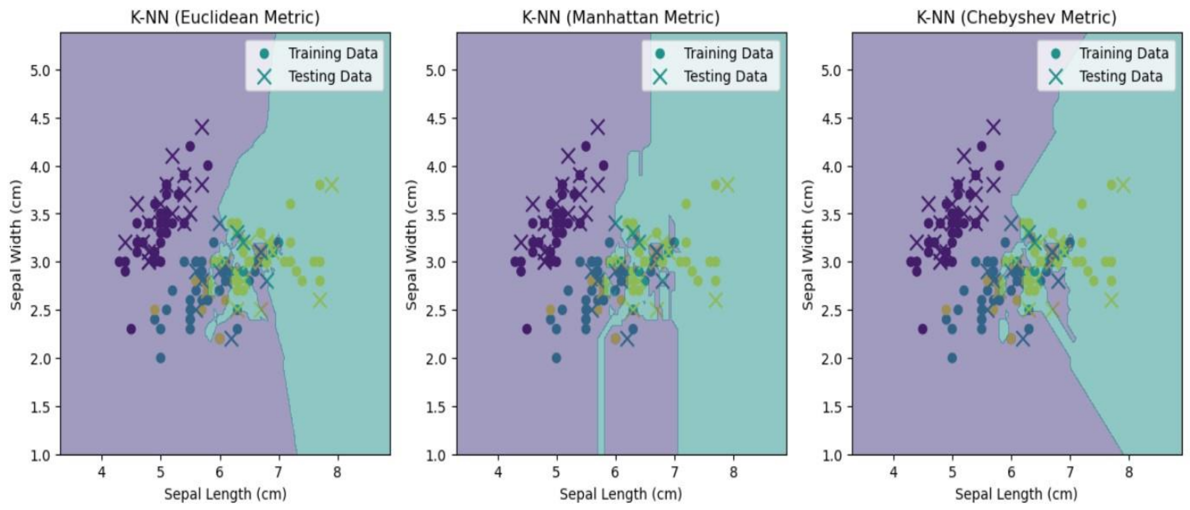
```
Distance Metric: euclidean
Confusion Matrix:
[[19  0  0]
 [ 0  7  6]
 [ 0  5  8]]

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        19
           1       0.58      0.54      0.56        13
           2       0.57      0.62      0.59        13

    accuracy                           0.76        45
   macro avg       0.72      0.72      0.72        45
weighted avg       0.76      0.76      0.76        45
```



```
Distance Metric: manhattan
Confusion Matrix:
[[19  0  0]
 [ 0  7  6]
 [ 0  5  8]]

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        19
           1       0.58      0.54      0.56        13
           2       0.57      0.62      0.59        13

    accuracy                           0.76        45
   macro avg       0.72      0.72      0.72        45
weighted avg       0.76      0.76      0.76        45
```



```
Distance Metric: chebyshev
Confusion Matrix:
[[19  0  0]
 [ 0  8  5]
 [ 0  7  6]]

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        19
           1       0.53      0.62      0.57        13
           2       0.55      0.46      0.50        13

    accuracy                           0.73        45
   macro avg       0.69      0.69      0.69        45
weighted avg       0.73      0.73      0.73        45
```



Practical 8

Aim: - Implement the classification model using clustering for the following techniques with K means clustering with Prediction, Test Score and Confusion Matrix

Theory: -

K-Means clustering works by minimizing the **intra-cluster variance** (distance between the points within a cluster) and maximizing the **inter-cluster variance** (distance between the centroids of different clusters). Each data point is assigned to the cluster whose centroid is closest to it, based on some distance metric (usually **Euclidean distance**).

Code: -

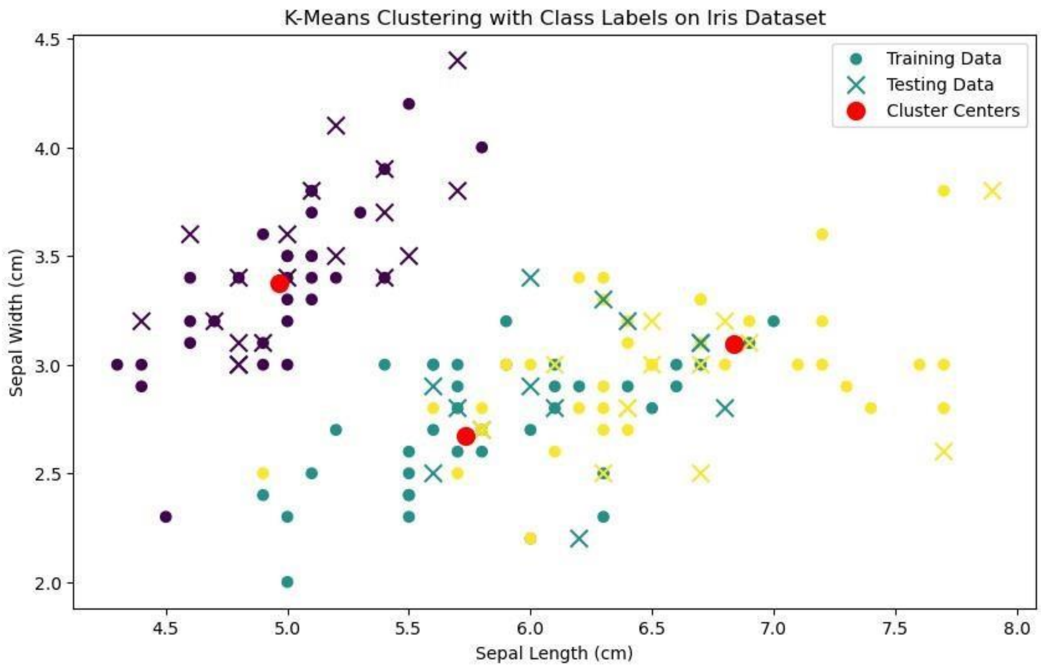
```
import numpy as np import pandas as pd import
matplotlib.pyplot as plt from sklearn.datasets
import load_iris from sklearn.model_selection import
train_test_split from sklearn.cluster import KMeans
from sklearn.metrics import classification_report,
confusion_matrix


#Load the Iris
dataset iris =
load_iris()
X = iris.data[:, :2] #Select only the features (sepal
lengthy and sepal width) y = iris.target


#Split database into traini9ng and testing
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3, random_state=42)


#Initalize K-Means clustering with the number of clusters equal
to the number of classes n_clusters = len(np.unique(y)) kmeans
= KMeans(n_clusters=n_clusters, random_state=42)


#Fit K-Means clustering to the training data
kmeans.fit(X_train)
```

```
#Assign cluster labels to data points in test
set cluster_labels = kmeans.predict(X_test)

#Assign class labels to clusters based on thge most frequent
class label in each cluster cluster_class_labels = [] for i in
range(n_clusters):
    cluster_indices = np.where(cluster_labels ==i)[0]
cluster_class_labels.append(np.bincount(y_test[cluster_indices
]).argmax())

#Assign cluster class labels to data points in the test set
y_pred    =
np.array([cluster_class_labels[cluster_labels[i]]      for  i
in range(len(X_test))])

#Evaluate the classifier's performance
print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred))
print("\nClassification Report:")
print(classification_report(y_test,
y_pred)) #Visualize the dataset and cluster
cemters plt.figure(figsize=(10, 6))

#Plot the training data points
plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train,
cmap='viridis', label='Training Data')

#Plot  testing  data  plt.scatter(X_test[:,  0],  X_test[:,  1],
c=y_test,  cmap='viridis',  marker='x',  s=100,  label='Testing
Data')

#plt cluster centers plt.scatter(kmeans.cluster_centers_[:, 0],
kmeans.cluster_centers_[:,  1],  c='red',  marker='o',  s=100,
label='Cluster Centers')
```

```
plt.xlabel('Sepal Length (cm)') plt.ylabel('Sepal
Width (cm)') plt.title('K-Means Clustering with
Class Labels on Iris Dataset') plt.legend()
plt.show()
```

Output: -

```
Confusion Matrix:
[[19  0  0]
 [ 0  8  5]
 [ 0  3 10]]

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        19
           1       0.73      0.62      0.67        13
           2       0.67      0.77      0.71        13

    accuracy                           0.82        45
   macro avg       0.80      0.79      0.79        45
weighted avg       0.82      0.82      0.82        45
```



Practical 9

Aim: - Implement the classification model using clustering for the following techniques with hierarchical clustering with Prediction, Test Score and Confusion Matrix

Theory: -

Hierarchical Clustering is a popular **unsupervised machine learning** technique used to group similar data points into clusters based on their distances from each other. Unlike K-Means clustering, which requires the number of clusters to be specified in advance, hierarchical clustering builds a hierarchy of clusters that can be represented in a **dendrogram** (a tree-like diagram), allowing users to determine the number of clusters by cutting the tree at a desired level.

Agglomerative Clustering:

- This is a bottom-up approach where each data point starts as its own cluster, and pairs of clusters are merged as one moves up the hierarchy.
- The steps are as follows:
  1. Start with each data point as its own cluster.
  2. Compute the proximity (or distance) between each pair of clusters.
  3. Merge the two closest clusters.
  4. Repeat steps 2-3 until all points are merged into a single cluster or a desired number of clusters is reached.

Divisive Clustering:

- This is a top-down approach where all data points start in a single cluster, and splits are performed recursively to divide the clusters into smaller ones.
- It is less commonly used than agglomerative clustering due to its computational complexity

A **dendrogram** is a visual representation of the hierarchical clustering process. It illustrates the arrangement of clusters and the distances at which clusters are merged. Each leaf node represents an individual data point, and the branches represent merges. The height at which two clusters are merged indicates the distance between them.

Hierarchical clustering is a versatile and intuitive method for grouping similar data points without requiring prior knowledge of the number of clusters. Its ability to provide a detailed visual representation of cluster relationships through dendrograms makes it a valuable tool in exploratory data analysis. While it has certain limitations, such as computational complexity and sensitivity to noise, it remains widely used in various fields for its interpretability and flexibility.

Code: -

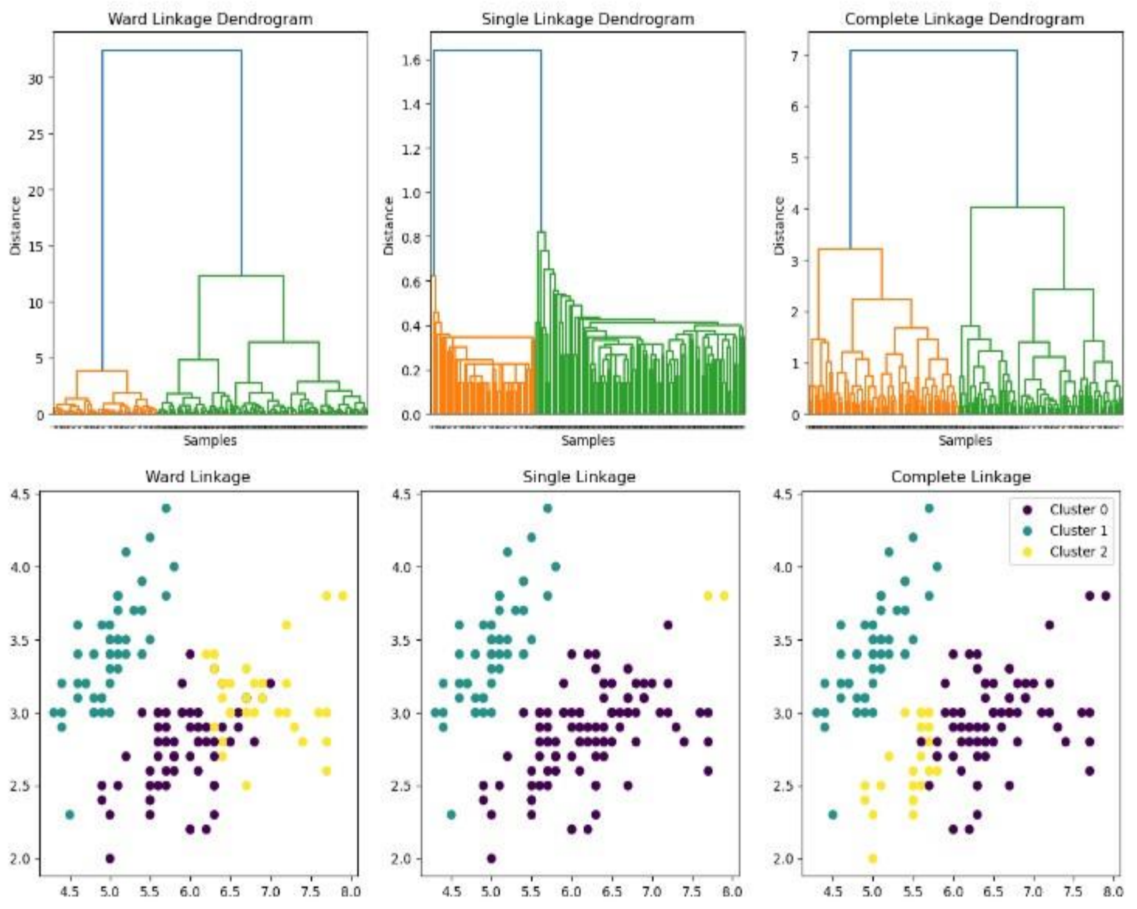
```
import pandas as pd import numpy as np from sklearn.cluster
import AgglomerativeClustering from sklearn.model_selection
import train_test_split from sklearn.ensemble import
RandomForestClassifier from sklearn.metrics import
accuracy_score, confusion_matrix, classification_report from
sklearn.datasets import load_iris import matplotlib.pyplot as
plt from scipy.cluster.hierarchy import dendrogram, linkage

#Load the Iris
dataset iris =
load_iris() X =
iris.data y =
iris.target

#Step 1: Hierarchical Clustering with different Linkage Methods
and Draw denograms n_clusters = 3 # Number of clusters
linkage_methods = ['ward', 'single', 'complete'] # Different
Linkage methods cluster_labels = []

#Define figure and axes for
dendrograms
plt.figure(figsize=(15, 5))
dendrogram_axes = []

for i, linkage_method in enumerate(linkage_methods):
          labels    =
AgglomerativeClustering(n_clusters=n_clusters,
linkage=linkage_method).fit_predict(X)
cluster_labels.append(labels)

#Create  a dendrgram for the current linkage method
dendrogram_data = linkage(X, method=linkage_method)
```

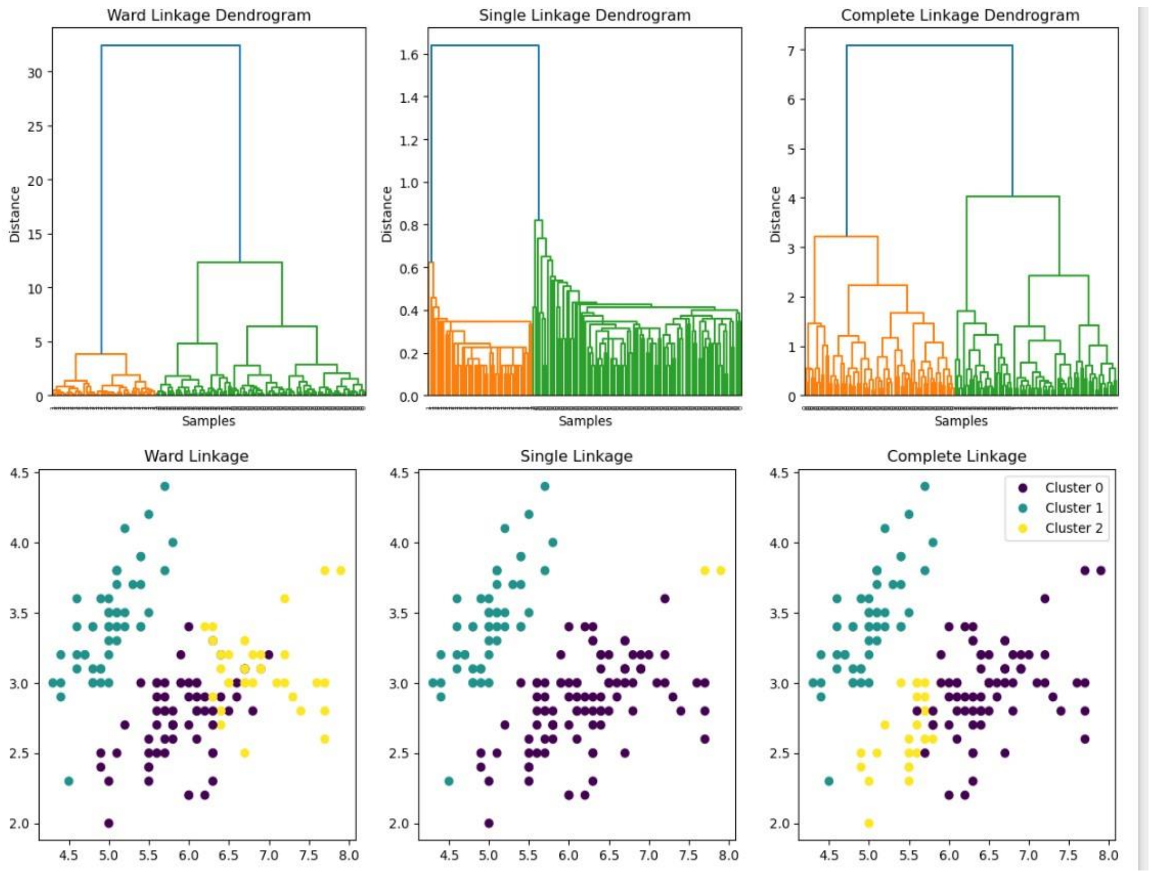
```
dendrogram_axes.append(plt.subplot(1,
len(linkage_methods), i+1))
dendrogram(dendrogram_data, orientation='top',
labels=labels)
plt.title(f"{linkage_method.capitalize()} Linkage
Dendrogram")     plt.xlabel('Samples')
plt.ylabel('Distance')

#Plot clustering results for different
linkage methods plt.figure(figsize=(15,
5)) for i, linkage_method in
enumerate(linkage_methods):
    plt.subplot(1, len(linkage_methods), i + 1)
scatter = plt.scatter(X[:, 0], X[:, 1],
c=cluster_labels[i], cmap='viridis',
label=f'Clusters ({linkage_method.capitalize()} Linkage)')
plt.title(f"{linkage_method.capitalize()} Linkage")
#Add         legend         to         scatter         plots
plt.legend(handles=scatter.legend_elements()[0],
labels=[f'Cluster {i}' for i in range(n_clusters)])


#sTEP 2 :fEATURE ENGINEERING (uSING CLUSTER ASSIGNMENT AS A
feature)
X_with_cluster = np.column_stack((X, cluster_labels[-1])) #
using complete linkage


#Step 3: Classification
X_train,  X_test,   y_train,  y_test    =
train_test_split(X_with_cluster,   y, test_size=0.2,
random_state=42) classifier =
RandomForestClassifier(n_estimators=100, random_state=42)
classifier.fit(X_train, y_train)
```

```
#Step 4: Prediction y_pred
=
classifier.predict(X_test)

#Step 5 : Test Score and Confusion
Matrix accuracy = accuracy_score(y_test,
y_pred) conf_matrix =
confusion_matrix(y_test, y_pred)

#Genrate classification report with zero_division parametrs
classification_rep = classification_report(y_test, y_pred,
zero_division=0)

#Print cluster description
cluster_descriptions = {
    'ward': 'Clusters based on Ward linkage interpretation.',
    'single': 'Cluster based on Single linkage
interpretation.',
    'complete': 'Clusters based on Complete linkage
interpretation.'
}
for method in linkage_methods:
    print(f"Cluster     Descriptions     ({method.capitalize()}
Linkage):")      print(cluster_descriptions[method.lower()])  #
Convert to lowercase for dictionary access
# Print accuracy, confusion matrix, and
classification report print("Accuracy:",
accuracy) print("Confusion Matrix:\n",
conf_matrix) print("Classification Report:\n",
classification_rep) plt.show()
```

Output: -

```
Cluster Descriptions (Ward Linkage):
Clusters based on Ward linkage interpretation.
Cluster Descriptions (Single Linkage):
Cluster based on Single linkage interpretation.
Cluster Descriptions (Complete Linkage):
Clusters based on Complete linkage interpretation.
Accuracy: 1.0
Confusion Matrix:
 [[10  0  0]
 [ 0  9  0]
 [ 0  0 11]]
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        10
           1       1.00      1.00      1.00         9
           2       1.00      1.00      1.00        11

    accuracy                           1.00        30
   macro avg       1.00      1.00      1.00        30
weighted avg       1.00      1.00      1.00        30
```




Ward Linkage Dendrogram
Single Linkage Dendrogram
Complete Linkage Dendrogram
Distance
Samples
Ward Linkage
Single Linkage
Complete Linkage
Cluster 0
Cluster 1
Cluster 2

Practical 10

Aim: - Implement the Rule based method and test the same.

Theory: -

Rule-Based Methods are a type of artificial intelligence and machine learning approach that uses a set of "if-then" rules to make decisions, classify data, or derive insights from datasets. These methods are particularly useful in applications where human expertise can be encoded into a systematic approach, making them highly interpretable and easy to understand.

Rules: The fundamental building blocks of rule-based systems. A rule typically consists of two parts:

- **Condition (Antecedent)**: The "if" part of the rule, specifying a condition that must be satisfied.
- **Conclusion (Consequent)**: The "then" part of the rule, specifying the action to be taken or the result to be achieved if the condition is met.

For example:

- **Rule**: If a customer's age is greater than 30 and their income is above $50,000, then classify them as a "high-income customer."

Knowledge Base: A collection of rules that govern the decision-making process. This knowledge base can be created through domain expertise or learned from data using various algorithms.

Inference Engine: The component that applies the rules to the knowledge base and makes decisions or classifications based on the given input data. It evaluates the rules and determines which ones are applicable.

Explanation Facility: This provides insight into how a decision was made based on the rules applied. It enhances the interpretability of the system and helps users understand the reasoning behind specific outcomes.

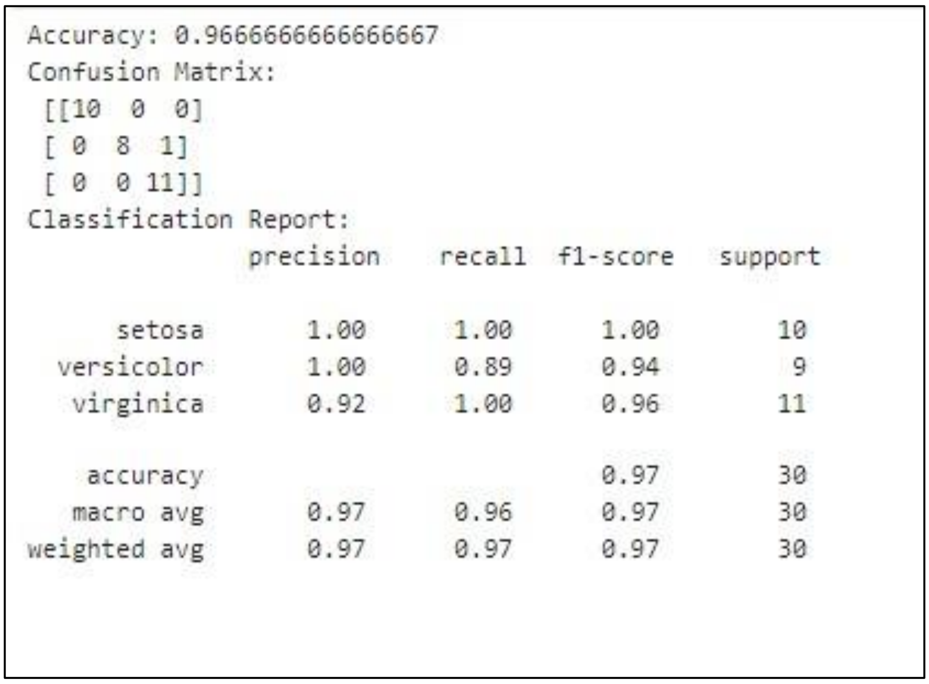
Code: -

```
import numpy as np from sklearn.datasets
import load_iris from
sklearn.model_selection import
train_test_split from    sklearn.metrics
import    accuracy_score,
confusion_matrix, classification_report
```

```
#Load the Iris
dataset iris =
load_iris() X =
iris.data y =
iris.target

#Split the data for testing
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)


#Define a simple rule-based
classifier function def
rule_based_classifier(x):     if
x[2] < 2.0:
        rule = "If feature 2 < 2.0,
assign to Classd 0"         return 0 #
Class 0     elif x[3] > 1.5:
        rule = "If feature 2 >= 2.0 and feature 3 >
1.5, assign to Class 2"         return 2 # Class 2
else:
        rule = "If feature 2 >= 2.0 and feature 3
<=1.5, assign to Class 1"         return 1 # Class 1
print("Rule:", rule)

# Apply the rule-based classifier to make predictions on
the test set y_pred = [rule_based_classifier(x) for x in
X_test]

# Calculate accuracy, confusion matrix, and
classification report accuracy =
accuracy_score(y_test, y_pred) conf_matrix =
confusion_matrix(y_test, y_pred)
classification_rep  =    classification_report(y_test,
y_pred, target_names=iris.target_names)
```

```
# Print the results print("Accuracy:",
accuracy) print("Confusion Matrix:\n",
conf_matrix) print("Classification
Report:\n", classification_rep)
```

Output: -



```
Accuracy: 0.9666666666666667
Confusion Matrix:
 [[10  0  0]
 [ 0  8  1]
 [ 0  0 11]]
Classification Report:
               precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        10
  versicolor       1.00      0.89      0.94         9
   virginica       0.92      1.00      0.96        11

    accuracy                           0.97        30
   macro avg       0.97      0.96      0.97        30
weighted avg       0.97      0.97      0.97        30
```

Practical 11

Aim: - Write a program to construct a Bayesian network considering medical data. Use this model to demonstrate the diagnosis of heart patients using standard Heart Disease Data Set.

Theory: -

Bayesian Networks (also known as **Bayesian Belief Networks** or **Bayesian Models**) are probabilistic graphical models that represent a set of variables and their conditional dependencies using a directed acyclic graph (DAG). These networks allow for the representation of complex relationships between random variables, making them useful in various fields such as machine learning, artificial intelligence, decision-making, and data analysis.

- **Nodes**: Each node in the graph represents a random variable, which can be discrete or continuous. The variable can represent observable quantities, latent variables, or even hypotheses.

- **Edges**: Directed edges between nodes indicate a probabilistic relationship, showing how one variable influences another. An edge from node A to node B implies that A is a parent of B, suggesting that the probability of B depends on the state of A.

- **Conditional Probability Distributions**: Each node has an associated conditional probability distribution (CPD) that quantifies the effect of its parent nodes on the node itself. For a node with no parents, the CPD is simply the prior probability of that node.

- **Directed Acyclic Graph (DAG)**: The structure of a Bayesian Network is represented as a DAG, ensuring that there are no cycles. This property guarantees that the dependencies between variables are well-defined.

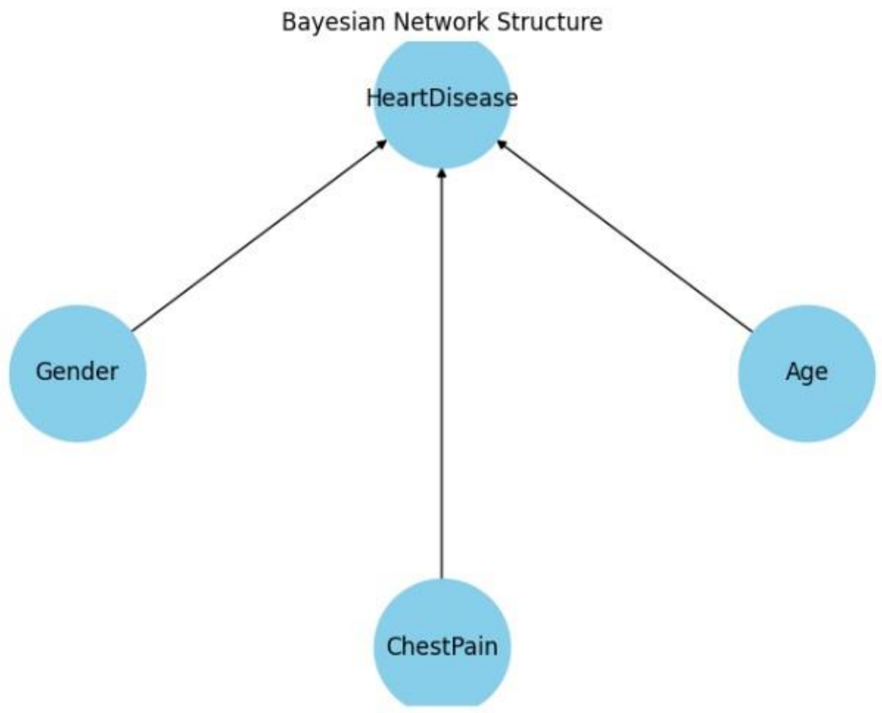
Bayesian Networks offer a powerful framework for modeling uncertainty and making probabilistic inferences in complex systems. Their ability to represent relationships between variables through a directed acyclic graph makes them valuable in a wide range of applications. Despite some challenges in structure learning and scalability, Bayesian Networks remain a fundamental tool in datadriven decision-making and artificial intelligence.

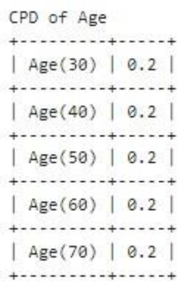
Code: -

```
import numpy as np import pandas as

pd from pgmpy.models import

BayesianNetwork

from      pgmpy.estimators      import      ParameterEstimator,
MaximumLikelihoodEstimator from

pgmpy.inference import
```

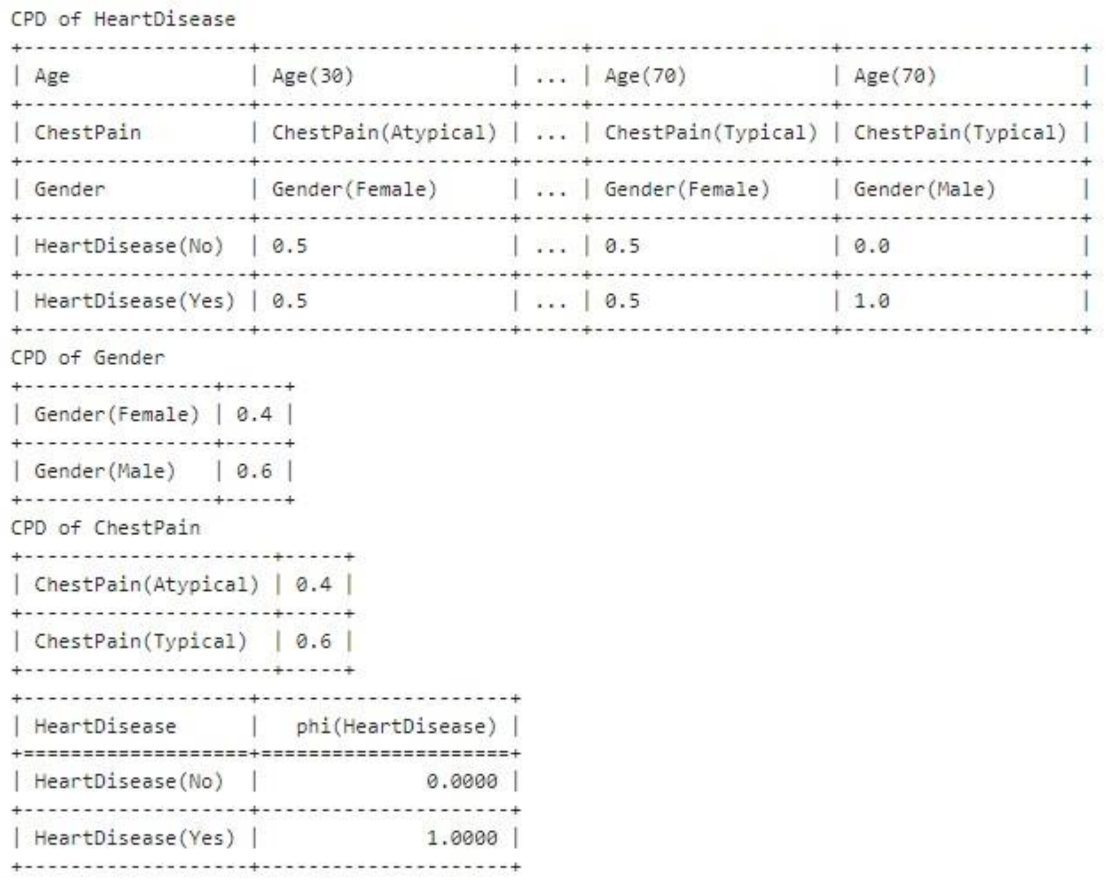
```
VariableElimination import networkx as
nx import matplotlib.pyplot as plt

data = pd.DataFrame (data={'Age': [30, 40, 50, 60, 70],
                           'Gender': ['Male', 'Female',
'Male', 'Female', 'Male'],
                           'ChestPain': ['Typical',
'Atypical',    'Typical',     'Atypical', 'Typical'],
                           'HeartDisease': ['Yes',
'No', 'Yes', 'No', 'Yes']}) model =
BayesianNetwork([('Age', 'HeartDisease'),
                         ('Gender', 'HeartDisease'),
                         ('ChestPain', 'HeartDisease')])

model.fit(data, estimator=MaximumLikelihoodEstimator)

pos = nx.circular_layout(model)
nx.draw(model,    pos,    with_labels=True,    node_size=5000,
node_color="skyblue", font_size=12,
font_color="black") plt.title("Bayesian
Network Structure") plt.show()

for cpd in model.get_cpds():
    print("CPD of", cpd.variable)
print(cpd)

inference = VariableElimination(model) query =
inference.query(variables=['HeartDisease'],
evidence={'Age':50,
'Gender': 'Male', 'ChestPain':
'Typical'}) print(query)
```

Output: -





```
CPD of Age
+---------+-----+
| Age(30) | 0.2 |
+---------+-----+
| Age(40) | 0.2 |
+---------+-----+
| Age(50) | 0.2 |
+---------+-----+
| Age(60) | 0.2 |
+---------+-----+
| Age(70) | 0.2 |
+---------+-----+
```



```
CPD of HeartDisease
+-------------------+---------------------+-----+---------------------+---------------------+
| Age               | Age(30)             | ... | Age(70)             | Age(70)             |
+-------------------+---------------------+-----+---------------------+---------------------+
| ChestPain         | ChestPain(Atypical) | ... | ChestPain(Typical)  | ChestPain(Typical)  |
+-------------------+---------------------+-----+---------------------+---------------------+
| Gender            | Gender(Female)      | ... | Gender(Female)      | Gender(Male)        |
+-------------------+---------------------+-----+---------------------+---------------------+
| HeartDisease(No)  | 0.5                 | ... | 0.5                 | 0.0                 |
+-------------------+---------------------+-----+---------------------+---------------------+
| HeartDisease(Yes) | 0.5                 | ... | 0.5                 | 1.0                 |
+-------------------+---------------------+-----+---------------------+---------------------+
CPD of Gender
+----------------+-----+
| Gender(Female) | 0.4 |
+----------------+-----+
| Gender(Male)   | 0.6 |
+----------------+-----+
CPD of ChestPain
+---------------------+-----+
| ChestPain(Atypical) | 0.4 |
+---------------------+-----+
| ChestPain(Typical)  | 0.6 |
+---------------------+-----+
+-------------------+---------------------+
| HeartDisease      |   phi(HeartDisease) |
+===================+=====================+
| HeartDisease(No)  |              0.0000 |
+-------------------+---------------------+
| HeartDisease(Yes) |              1.0000 |
+-------------------+---------------------+
```

Practical 12

Aim: - Implement the non-parametric Locally Weighted Regression algorithm in order to fit data points. Select appropriate data set for your experiment and draw graphs.

Theory: -

Locally Weighted Regression (LWR), also known as **Locally Weighted Scatterplot Smoothing (LOWESS)** or **LOESS**, is a non-parametric regression technique that fits multiple regressions in localized subsets of the data to create a smooth curve through a scatterplot of the data. This approach is particularly useful when the relationship between the independent and dependent variables is complex and cannot be adequately captured by traditional parametric models.

- **Non-Parametric Nature**: Unlike parametric methods that assume a specific functional form for the relationship between variables (e.g., linear, quadratic), non-parametric methods like LWR do not assume any global structure. Instead, they focus on local relationships, making them flexible for capturing intricate patterns in data.

- **Locally Weighted**: LWR fits a separate regression line to a subset of data points that are near the target point (the point where the prediction is being made). The weight assigned to each data point is based on its distance from the target point, with closer points receiving higher weights. This local approach allows LWR to adapt to changes in the underlying data structure.

- **Kernel Function**: The weighting is often done using a kernel function, such as the **Gaussian kernel**, which assigns weights that decrease with distance. The bandwidth (or the width of the kernel) controls how many points influence the estimate and how localized the fit is. A smaller bandwidth results in a more sensitive fit, while a larger bandwidth yields a smoother curve.

The process of Locally Weighted Regression can be summarized in the following steps:

1. **Select a Point**: Choose the point xxx at which you want to predict the value of the dependent variable.
2. **Calculate Weights**: For each data point (xi,yi) in the dataset, calculate the weight wi using a kernel function:

wi=K(x,xi, τ)

where K is the kernel function, and τ is the bandwidth parameter.

3. **Fit Local Model**: Fit a weighted linear regression model using the weights calculated in the previous step. This involves minimizing the weighted sum of squared errors:

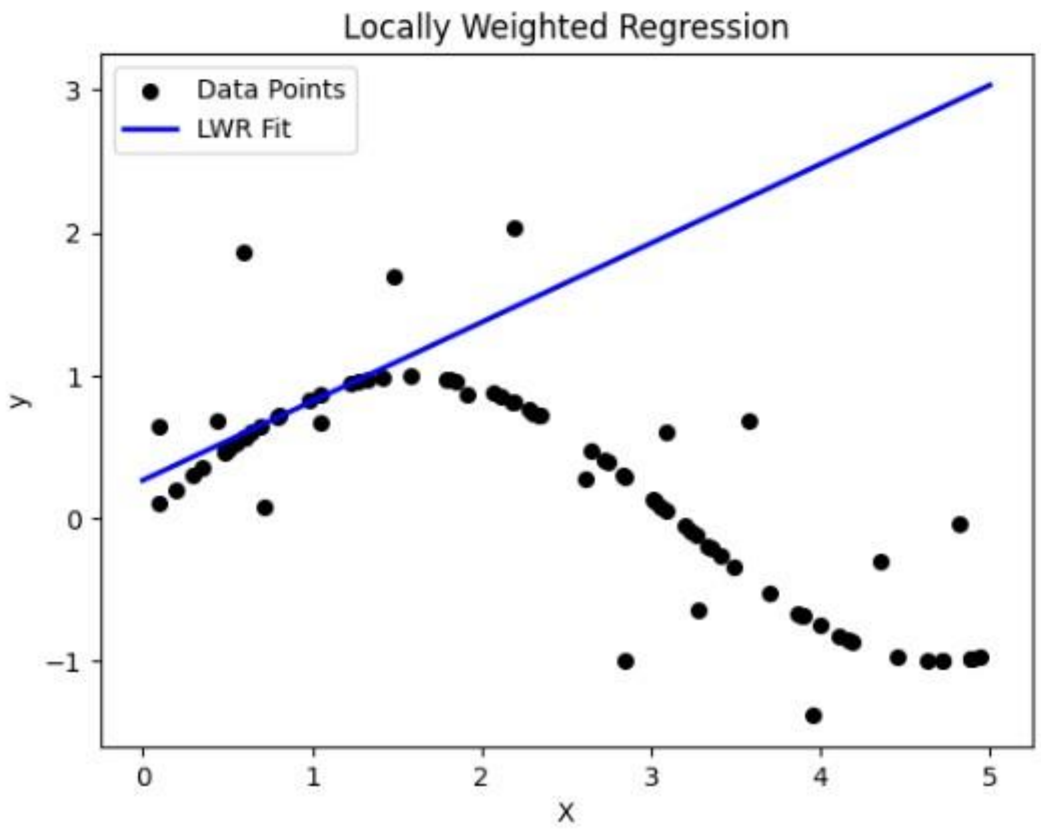
$$\hat{y}(x) = \arg\min_{\beta} \sum_{i=1}^{n} w_i (y_i - \beta^T x_i)^2$$

4. **Make Prediction**: Use the fitted model to predict the value of the dependent variable at the point xxx.
5. **Repeat**: Repeat the process for each point of interest to construct the overall fitted curve.

Code: -

```
import numpy as np
import
matplotlib.pyplot as
plt


# Seed for reproducibility
np.random.seed(0)


# Generate random dataset
X = np.sort(5 * np.random.rand(80,
1), axis=0) y = np.sin(X).ravel()
y[::5] += 3 * (0.5 -
np.random.rand(16))


# Locally Weighted Regression function def
locally_weighted_regression(query_point, X, y,
tau=0.1):
    m =
X.shape[0]     #
Calculate weights
    weights = np.exp(-((X - query_point) * 2).sum(axis=1) / (2
* tau * 2))
    W = np.diag(weights)


    # Add bias term to X
    X_bias = np.c_[np.ones((m, 1)), X]


    # Calculate theta using weighted least squares
    theta                                                           =
np.linalg.inv(X_bias.T.dot(W).dot(X_bias)).dot(X_bias.T).dot(W
).dot(y)


    # Predict for query_point
x_query = np.array([1,
query_point])     prediction =
x_query.dot(theta)     return
prediction


# Generate test points
X_test = np.linspace(0, 5, 100)


#  Predict  using  locally  weighted  regression  predictions  =
[locally_weighted_regression(query_point,  X,  y,  tau=0.1)  for
query_point in X_test]


# Plot results plt.scatter(X, y, color='black', s=30,
marker='o', label='Data Points') plt.plot(X_test,
predictions, color='blue', linewidth=2, label='LWR
Fit') plt.xlabel('X') plt.ylabel('y')
plt.title('Locally Weighted Regression') plt.legend()
plt.show()
```

Output: -


Locally Weighted Regression
Data Points
LWR Fit
y
X
3
2
1
0
−1
0
1
2
3
4
5

Practical 13

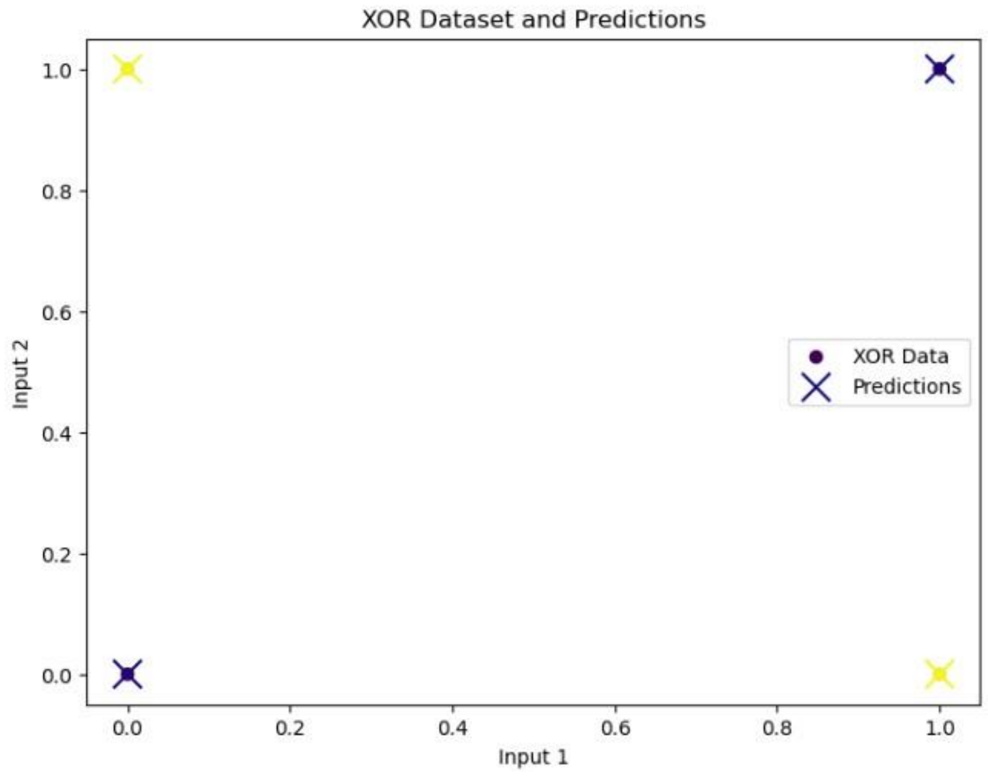
EXTRA Aim: - XOR NEURAL NETWORKS Theory: -

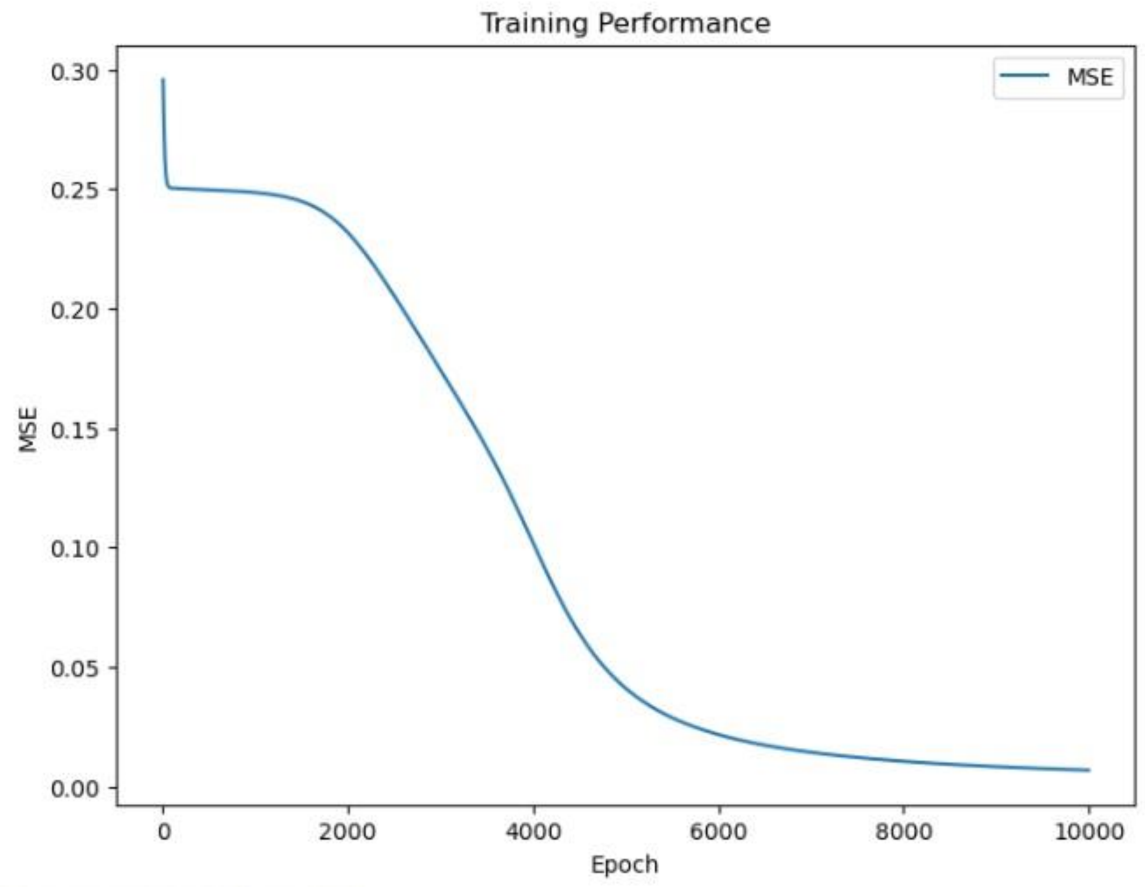
Code: -

```
import numpy as np

import

matplotlib.pyplot as

plt


# Sigmoid activation function and its

derivative def sigmoid(x):

    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):

    return x * (1 - x)


# Define the neuralNetwork class class

NeuralNetwork:     def __init__(self,

input_size, hidden_size, output_size):

        #Initialize     weights     with     random     values
self.weights_input_hidden = np.random.uniform(size=(input_size,
hidden_size))                    self.weights_hidden_output  =
np.random.uniform(size=(hidden_size, output_size))


    def forward(self, X):         #Forward

propagation         self.hidden_input =

np.dot(X, self.weights_input_hidden)

self.hidden_output = sigmoid(self.hidden_input)

        self.output    =    sigmoid(np.dot(self.hidden_output,
self.weights_hidden_output))         return self.output


    def backward(self, X, y, learning_rate):
        #Backpropagation         error_output = y-
self.output         delta_output = error_output *
sigmoid_derivative(self.output)

        error_hidden =
delta_output.dot(self.weights_hidden_output.T)
delta_hidden = error_hidden *
sigmoid_derivative(self.hidden_output)
self.weights_hidden_output +=
self.hidden_output.T.dot(delta_output) * learning_rate
self.weights_input_hidden += X.T.dot(delta_hidden) *
learning_rate

    def train(self, X, y, learning_rate, epochs):
        self.loss_history = [] #Track loss
dusring training         for _ in
range(epochs):
            output = self.forward(X)
error = y-output
self.loss_history.append(np.mean(error**2)) #Track
MSE             self.backward(X, y, learning_rate)

    def predict(self, X):
        return self.forward(X)

#XOR dataset
X = np.array([[0, 0], [0, 1], [1,
0],[1, 1]]) y = np.array([[0],
[1], [1], [0]])

#Initalize and train the
neural network input_size = 2
hidden_size = 4 output_size =
1 learning_rate = 0.1 epochs
= 10000

nn = NeuralNetwork(input_size, hidden_size,
output_size) nn.train(X, y, learning_rate, epochs)


#Make predictions
predictions =
nn.predict(X)

#Plot the XOR dataset and predictions plt.figure(figsize=(8, 6))
plt.scatter(X[:,0],  X[:,1],  c=y,  cmap='viridis',  label='XOR
Data')  plt.scatter(X[:,0],  X[:,1],  c=np.round(predictions),
cmap='plasma',    marker='x',    s=200,    label='Predictions')
plt.title('XOR Dataset and Predictions') plt.xlabel('Input 1')
plt.ylabel('Input 2') plt.legend()

#PLot the performance (MSE) during
training plt.figure(figsize=(8,6))
plt.plot(nn.loss_history,
label='MSE') plt.title('Training
Performance') plt.xlabel('Epoch')
plt.ylabel('MSE') plt.legend()

# Print predictions and actual
values for i in range(len(X)):
    print(f"Input: {X[i]}, Actual: {y[i]}, Predicted:
{np.round(predictions[i])}")

plt.show()
```

Output: -

```
Input: [0 0], Actual: [0], Predicted: [0.]
Input: [0 1], Actual: [1], Predicted: [1.]
Input: [1 0], Actual: [1], Predicted: [1.]
Input: [1 1], Actual: [0], Predicted: [0.]
```




Training Performance
MSE
0.30
0.25
0.20
0.15
0.10
0.05
0.00
0
2000
4000
6000
8000
10000
Epoch
MSE