INDEX

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: A naïve Bayesian classifier is a simple and efficient probabilistic classifier based on Bayes' theorem. It assumes that the features (or attributes) used for classification are independent of each other given the class label, which is often not true in real-world scenarios. Despite this simplifying assumption, naïve Bayes can perform surprisingly well, especially in text classification tasks.

Key Components:

1. **Bayes' Theorem**: The foundation of the classifier, given by:

   P(C|X)= P(X|C)·P(C) / P(X)

   where:

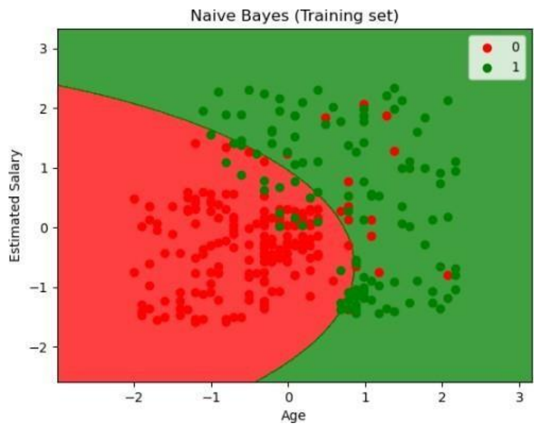
   - P(C|X) is the posterior probability of class C given features X.
   - P(X|C) is the likelihood of features X given class C.
   - P(C) is the prior probability of class C.
   - P(X) is the prior probability of features X.
   -

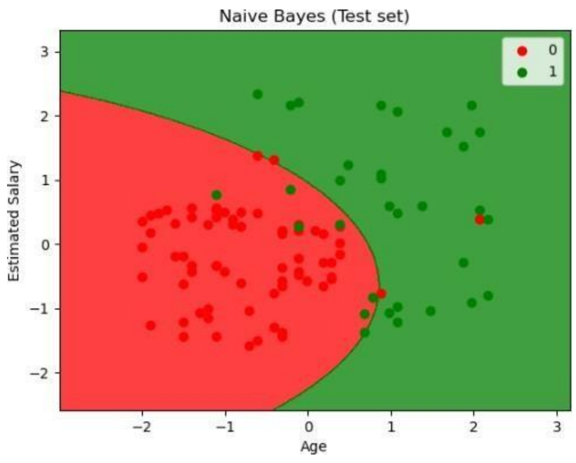
Code:

```
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:-





Practical 2

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

Theory:

Decision Tree Classifier: A Decision Tree is a flowchart-like structure that makes decisions based on feature values. It's easy to visualize and interpret, which makes it popular for both classification and regression tasks.

Key Steps:

1. Model Training: The tree is built by splitting the dataset into subsets based on feature values, minimizing impurity (like Gini impurity or entropy).
2. Prediction: To predict a class for a new instance, the model follows the branches of the tree based on the feature values until it reaches a leaf node.
3. Performance Evaluation: Common metrics include accuracy, test score, and confusion matrix.

Random Forest Classifier: A Random Forest is an ensemble of decision trees. It combines multiple trees to improve performance and control overfitting.

Key Steps:

1. Model Training: Randomly samples subsets of data and features to build multiple decision trees.
2. Prediction: Each tree votes for a class, and the class with the most votes is chosen as the final prediction.
3. Performance Evaluation: Same as decision trees, but generally shows improved metrics due to reduced variance.

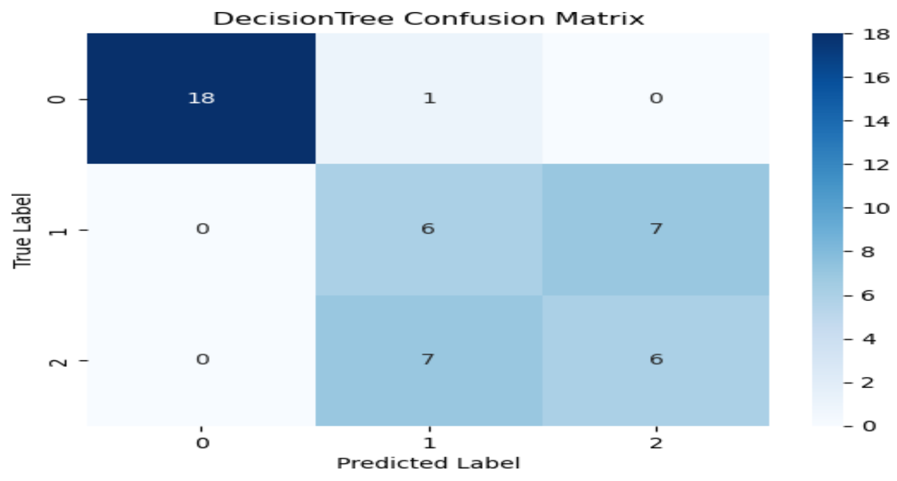
Code:-

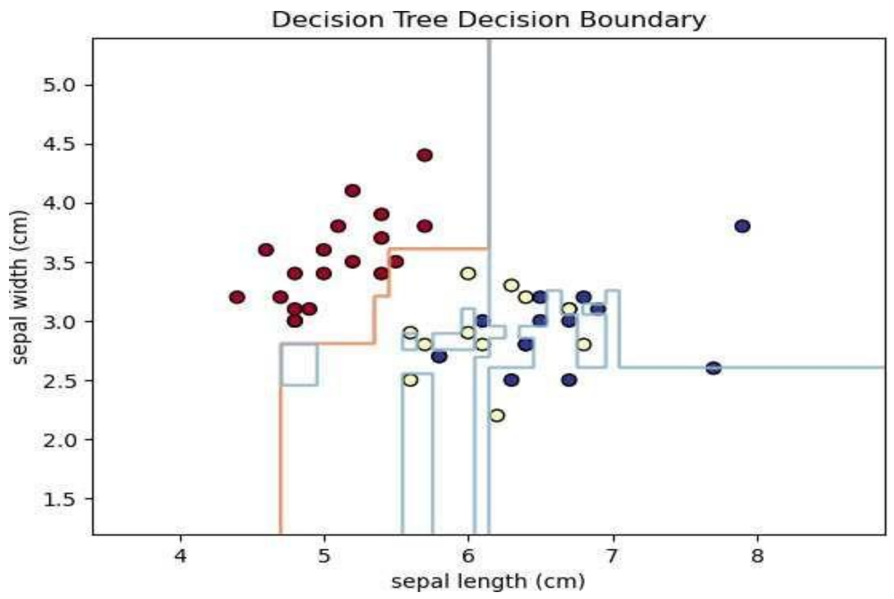
```
import numpy as np

import pandas as pd

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


iris = load_iris()

X = pd.DataFrame(iris.data, columns=iris.feature_names).iloc[:,:2]

y = pd.DataFrame(iris.target, columns=['species'])


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


def plot_decision_boundary(clf, X, y, title):

    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))


    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])

    Z = Z.reshape(xx.shape)


    plt.contour(xx, yy, Z, alpha=0.4, cmap=plt.cm.RdYlBu)

    plt.scatter(X.iloc[:,0], X.iloc[:,1], c=y.values.ravel(), s=40,
edgecolor='k', cmap=plt.cm.RdYlBu)

    plt.title(title)

    plt.xlabel(iris.feature_names[0])

    plt.ylabel(iris.feature_names[1])

    plt.show()


dt_model = DecisionTreeClassifier(random_state=42)

dt_model.fit(X_train, y_train)


dt_predictions = dt_model.predict(X_test)


dt_accuracy = accuracy_score(y_test, dt_predictions)
```

```
dt_confusion_matrix = confusion_matrix(y_test, dt_predictions)

print(f'Decision Tree Accuracy: {dt_accuracy}')
print('Decision Tree Classification Report:')
print(classification_report(y_test,  dt_predictions))

sns.heatmap(dt_confusion_matrix, annot=True, fmt='d', cmap='Blues')
plt.title('DecisionTree Confusion Matrix')
plt.ylabel('True Label')
plt.xlabel('Predicted Label')
plt.show()

plot_decision_boundary(dt_model, X_test, y_test, 'Decision Tree Decision
Boundary')

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

rf_predictions = rf_model.predict(X_test)

rf_accuracy = accuracy_score(y_test, rf_predictions)
rf_confusion_matrix = confusion_matrix(y_test, rf_predictions)

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

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_decision_boundary(rf_model, X_test, y_test, 'Random Forest Decision
Boundary')
```

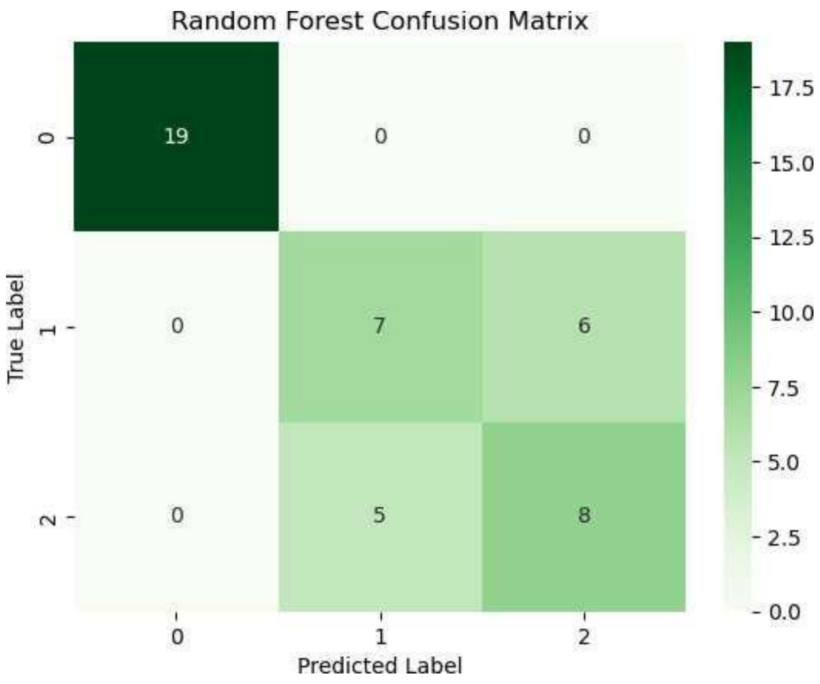
Output:-

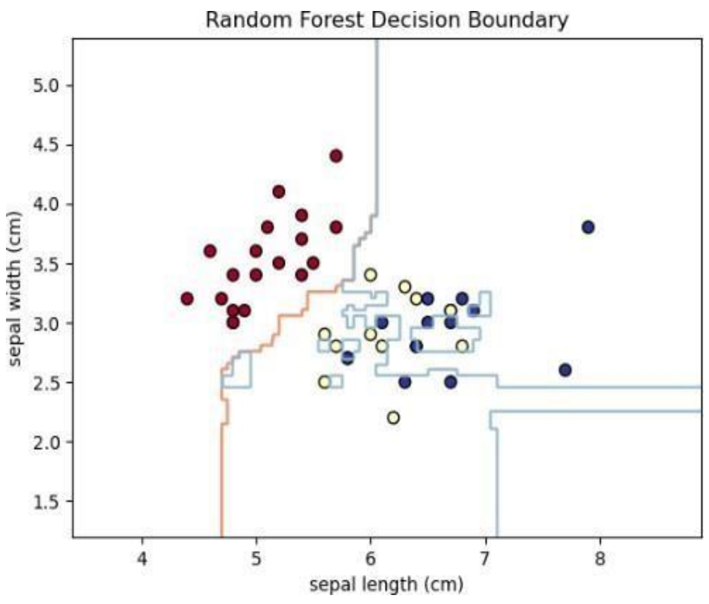
```
Decision Tree Accuracy: 0.6666666666666666
Decision Tree Classification Report:
              precision    recall  f1-score   support

           0       1.00      0.95      0.97        19
           1       0.43      0.46      0.44        13
           2       0.46      0.46      0.46        13

    accuracy                           0.67        45
   macro avg       0.63      0.62      0.63        45
weighted avg       0.68      0.67      0.67        45
```





```
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
```





Practical 3

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

Theory: Least Squares Regression is a statistical method used to estimate the relationship between one or more independent variables and a dependent variable. The goal is to minimize the sum of the squares of the differences (residuals) between observed and predicted values.

Key Concepts

1. Model Representation: For a linear regression model, the relationship can be represented as:

y=β0+β1x1+β2x2+...+βnxn+ϵ

where:

- y is the dependent variable.
- x1,x2,...,xn are independent variables.
- β0,β1,...,βn are coefficients to be estimated.
- ϵ is the error term.

2. Objective: The objective of the least squares method is to find the coefficients β\betaβ that minimize the cost function:

$$J(\beta) = \sum_{i=1}^{m}(y_i - \hat{y}_i)^2$$

where mmm is the number of observations, yi is the actual value, and y^I is the predicted value from the model.

Steps to Implement Least Squares Regression

1. Prepare the Data: Organize your dataset into independent variables (features) and the dependent variable (target).
2. Calculate the Coefficients: The coefficients can be estimated using the formula:

$$\beta = (X^T X)^{-1} X^T y$$

where X is the matrix of input features (with a column of ones for the intercept) and y is the vector of target values.

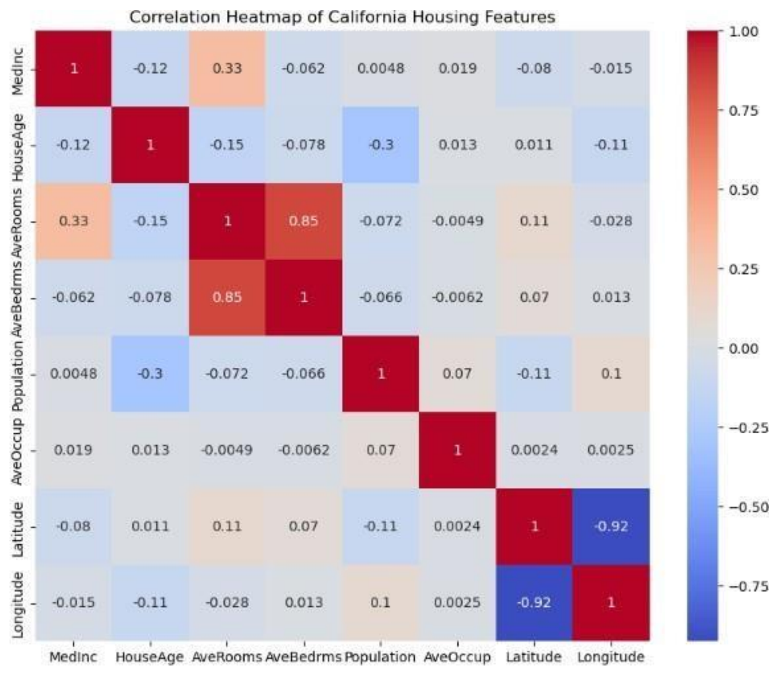
3. Make Predictions: Use the estimated coefficients to predict the values of the dependent variable.
4. Evaluate the Model: Common evaluation metrics include Mean Squared Error (MSE) and R-squared.

Code:-

```
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


housing = fetch_california_housing()

X = pd.DataFrame(housing.data, columns = housing.feature_names)

y = pd.DataFrame(housing.target, columns = ['MEDV'])


plt.figure(figsize=(10,8))

sns.heatmap(X.corr(), annot=True, cmap='coolwarm')

plt.title("Correlation Heatmap of California Housing Features")

plt.show()


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


reg_model = LinearRegression()

reg_model.fit(X_train, y_train)


y_train_pred = reg_model.predict(X_train)

y_test_pred = reg_model.predict(X_test)
```

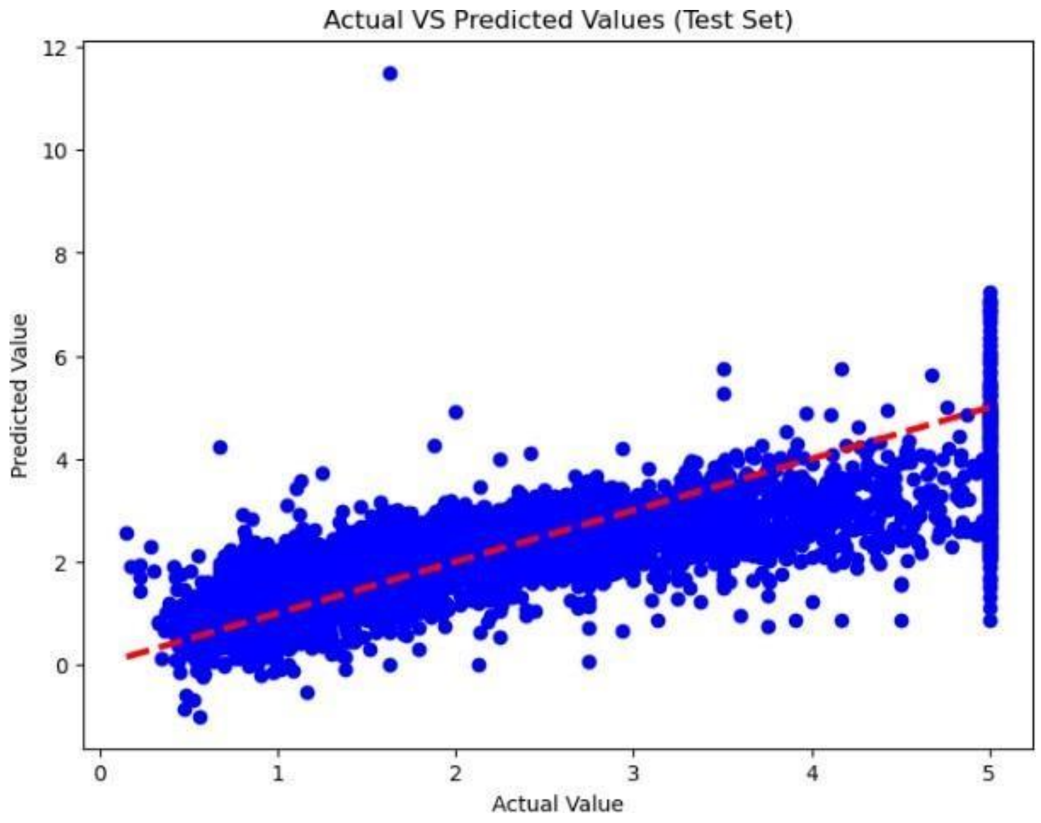
```
train_mse = mean_squared_error(y_train, y_train_pred)

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(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}')


coefficients = pd.DataFrame(reg_model.coef_.T, X.columns,
columns=['Coefficients'])

print(coefficients)

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 Value')

plt.ylabel('Predicted Value')

plt.title('Actual VS Predicted Values (Test Set)')
```

Output:-



	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Longitude
MedInc	1	-0.12	0.33	-0.062	0.0048	0.019	-0.08	-0.015
HouseAge	-0.12	1	-0.15	-0.078	-0.3	0.013	0.011	-0.11
AveRooms	0.33	-0.15	1	0.85	-0.072	-0.0049	0.11	-0.028
AveBedrms	-0.062	-0.078	0.85	1	-0.066	-0.0062	0.07	0.013
Population	0.0048	-0.3	-0.072	-0.066	1	0.07	-0.11	0.1
AveOccup	0.019	0.013	-0.0049	-0.0062	0.07	1	0.0024	0.0025
Latitude	-0.08	0.011	0.11	0.07	-0.11	0.0024	1	-0.92
Longitude	-0.015	-0.11	-0.028	0.013	0.1	0.0025	-0.92	1

```
Training Mean Squared Error: 0.5233576288267755
Test Mean Squared Error: 0.5305677824766757
Training R^2 Score: 0.6093459727972159
Test R^2 Score: 0.595770232606166
             Coefficients
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 is a statistical method used for binary classification problems, where the goal is to model the probability that a given input belongs to a particular category. Despite its name, it is a classification algorithm rather than a regression algorithm.

Key Concepts

1. **Sigmoid Function**: Logistic regression uses the logistic function (sigmoid function) to model probabilities. The sigmoid function maps any real-valued number into the range (0, 1):

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

where z is a linear combination of the input features.

2. **Model Representation**: The model can be expressed as:

$$P(y = 1|X) = \sigma(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n)$$

where P(y=1|X)P(y=1 | X)P(y=1|X) is the probability of the positive class given features X. x1,x2,...,xnx_1, x_2, ..., x_nx1,x2,...,xn are the independent variables. β0,β1,...,βn are the coefficients to be learned.

3. **Cost Function**: The cost function for logistic regression is based on the likelihood of the observed data, typically using the log loss (binary cross-entropy):

$$J(\beta) = -\frac{1}{m}\sum_{i=1}^{m}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)]$$

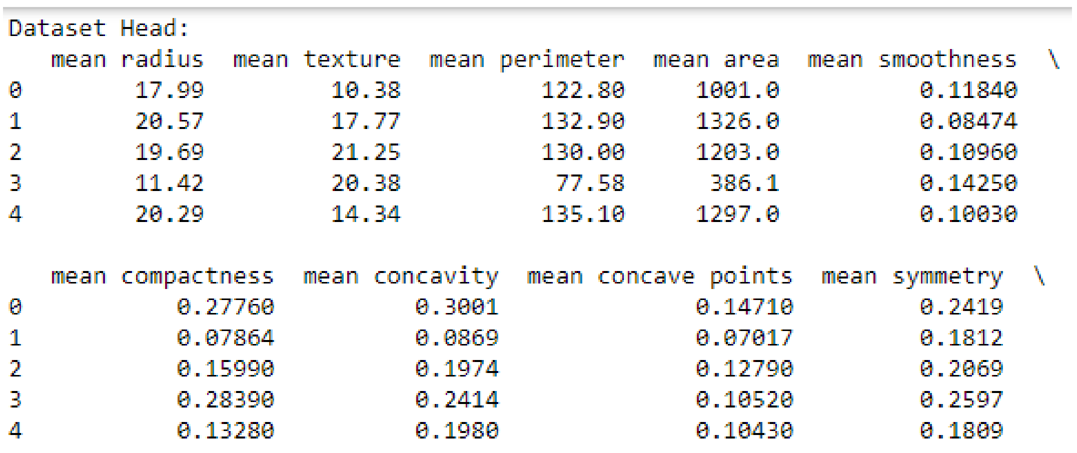
where y^i is the predicted probability for the instance.

4. **Optimization**: The coefficients are estimated using optimization techniques like gradient descent to minimize the cost function.

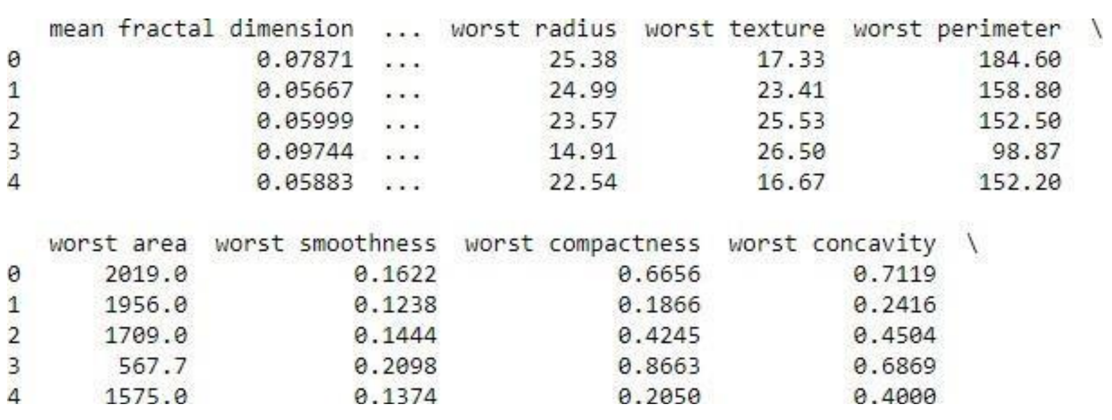
Code:-

```
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


cancer_data = load_breast_cancer()

X = pd.DataFrame(cancer_data.data, columns= cancer_data.feature_names)

y = pd.DataFrame(cancer_data.target, columns= ['target'])


print('Dataset Head:')

print(X.head())


print('Target Distribution:')

print(y['target'].value_counts())


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


logreg = LogisticRegression(max_iter=10000, random_state=42)

logreg.fit(X_train, y_train.values.ravel())


y_pred = logreg.predict(X_test)


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'Accuracy: {accuracy}')

print()

print('Confusion matrix:')
print(conf_matrix)

print()

print('Classification report:')

print(class_report)


plt.figure(figsize=(6,4))

sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues')

plt.title('Confusion Matrix')

plt.xlabel('Predicted Label')

plt.ylabel('True Label')

plt.show()

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

print(f'New input for prediction: {new_input}')

new_prediction = logreg.predict(new_input)

predicted_class = 'benign' if new_prediction == 1 else 'maligant'

print(f'Predicted class for the new input: {predicted_class}')

plt.figure(figsize=(6,4))

sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues')

plt.title('Confusion matrix - Test set')

plt.xlabel('Predicted Label')

plt.ylabel('True Label')

plt.show()
```

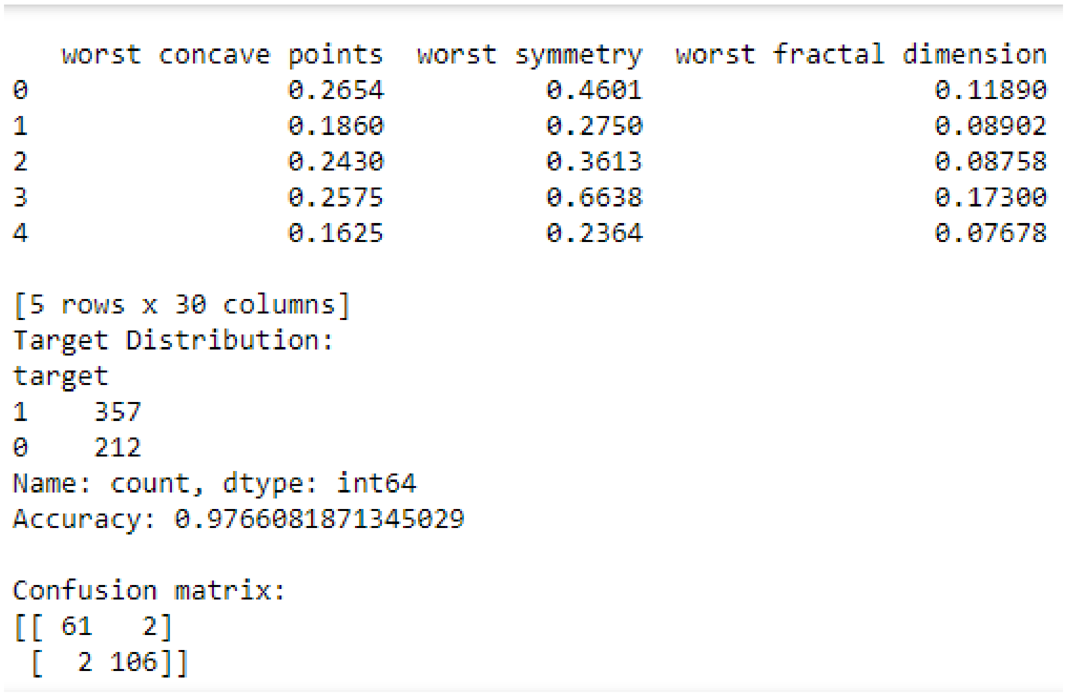
Output:-



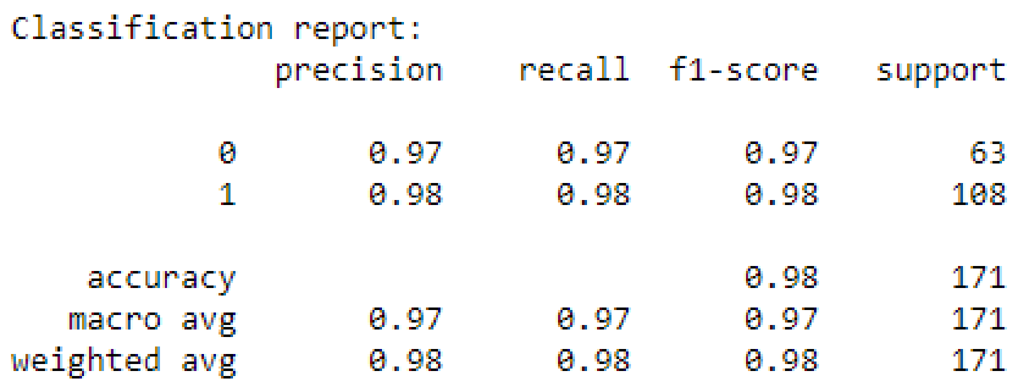
```
Dataset Head:
   mean radius  mean texture  mean perimeter  mean area  mean smoothness  \
0        17.99         10.38          122.80     1001.0          0.11840
1        20.57         17.77          132.90     1326.0          0.08474
2        19.69         21.25          130.00     1203.0          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
```



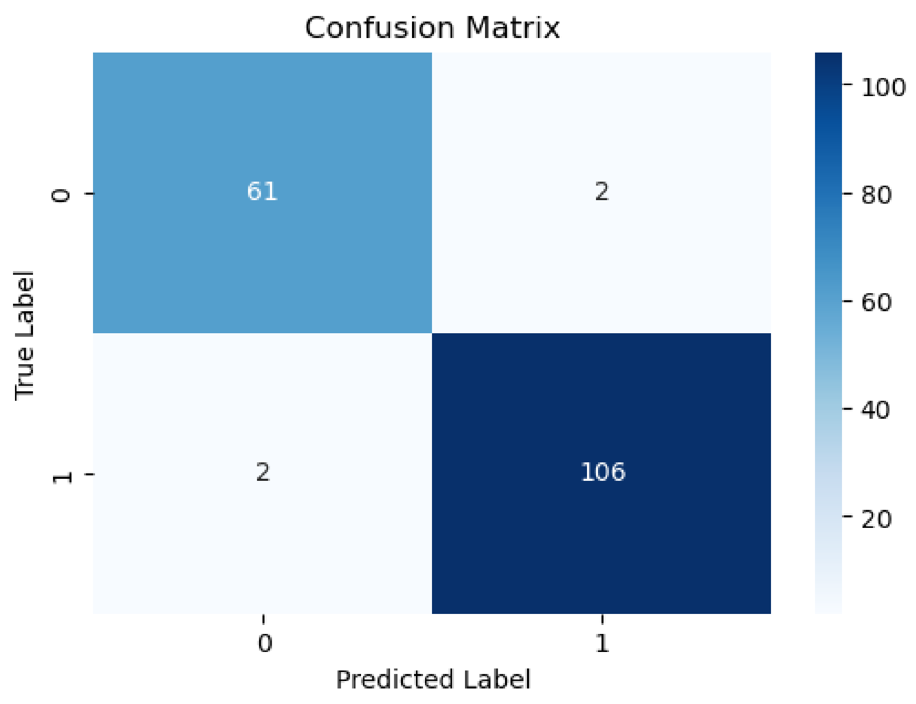
```
   mean fractal dimension  ...  worst radius  worst texture  worst perimeter  \
0                 0.07871  ...         25.38          17.33           184.60   
1                 0.05667  ...         24.99          23.41           158.80   
2                 0.05999  ...         23.57          25.53           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   
```

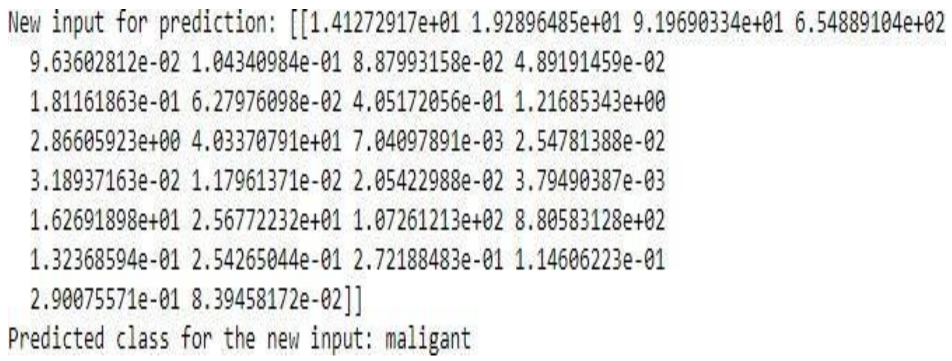


```
   worst concave points  worst symmetry  worst fractal dimension
0                0.2654          0.4601                  0.11890
1                0.1860          0.2750                  0.08902
2                0.2430          0.3613                  0.08758
3                0.2575          0.6638                  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]]
```

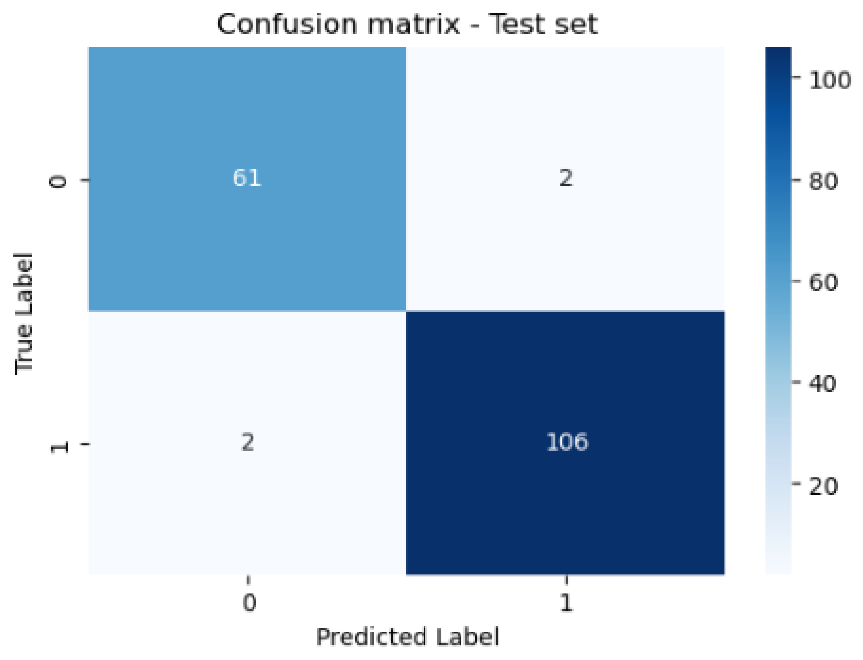


```
Classification report:
              precision    recall  f1-score   support

           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]]
Predicted class for the new input: maligant
```



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: The ID3 (Iterative Dichotomiser 3) algorithm is a popular decision tree algorithm used for classification tasks. It uses a top-down, recursive approach to build a tree based on the concept of information gain, which measures how much knowing the value of a feature improves our ability to classify the target variable.

Key Concepts:

1. **Entropy**: A measure of impurity or randomness in a dataset. Lower entropy indicates a more homogeneous dataset.

$$\text{Entropy}(S) = -\sum p_i \log_2(p_i)$$

   where pip_ipi is the proportion of class iii in the dataset SSS.

2. **Information Gain**: The reduction in entropy after splitting the dataset based on a feature

$$\text{Information Gain}(S, A) = \text{Entropy}(S) - \sum \frac{|S_v|}{|S|}\text{Entropy}(S_v)$$

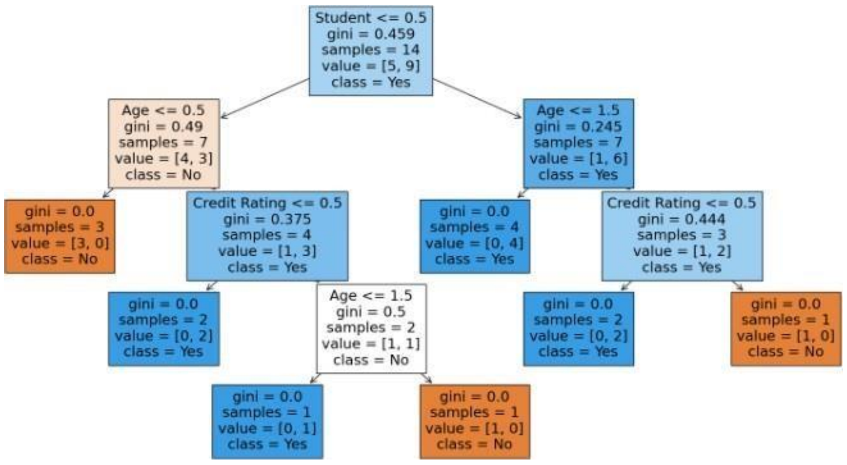
   where SvS_vSv is the subset of SSS for which attribute AAA has value vvv.

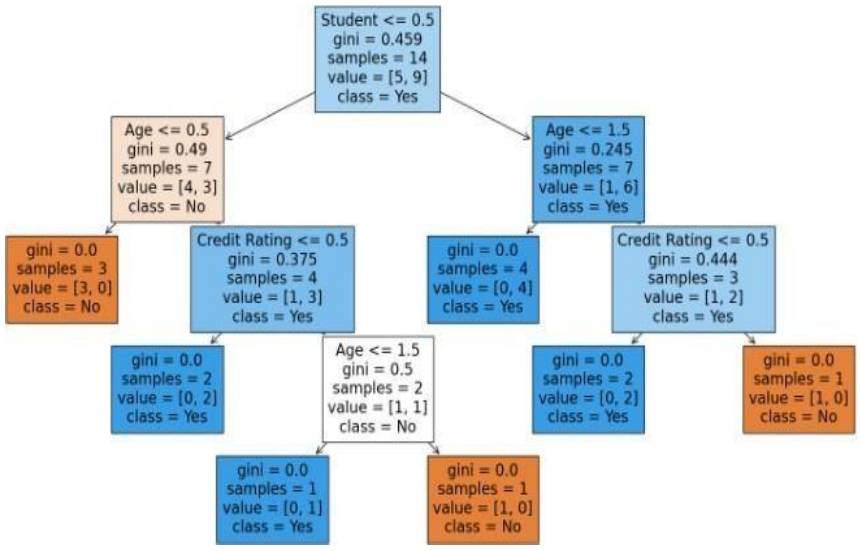
3. **Choosing the Best Feature**: The feature with the highest information gain is selected for splitting.

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_names, filled=True)

plt.show()


# Plot the decision tree

plt.figure(figsize=(20,10))

sk_tree.plot_tree(clf_gini, feature_names=feature_names,
class_names=class_names, 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)
print()
```

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 Neighbors (k-NN) algorithm is a simple, yet powerful, instance-based learning method used for classification and regression tasks. The core idea of k-NN is to classify a data point based on the majority class of its k-nearest neighbors in the feature space.

Key Concepts

1. **Distance Metric**: The most common distance metric used in k-NN is Euclidean distance, which measures the straight-line distance between two points in Euclidean space.

$$d(x, y) = \sqrt{\sum (x_i - y_i)^2}$$

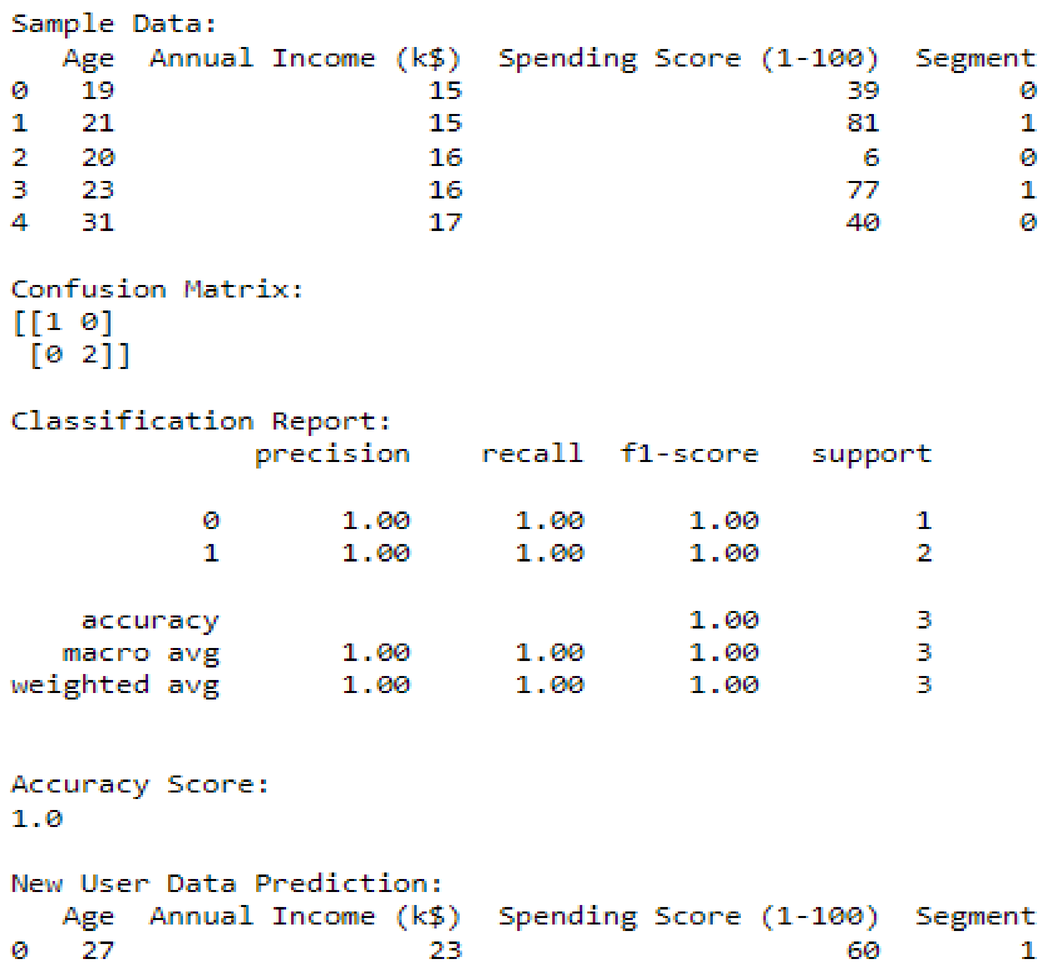
2. **Choosing k**: The parameter kkk determines how many neighbors to consider for classifying a new instance. A small kkk can make the model sensitive to noise, while a large kkk can smooth out class distinctions.
3. **Classification**: For classification tasks, the algorithm assigns the class label based on the majority class among the k-nearest neighbors.

Code:-

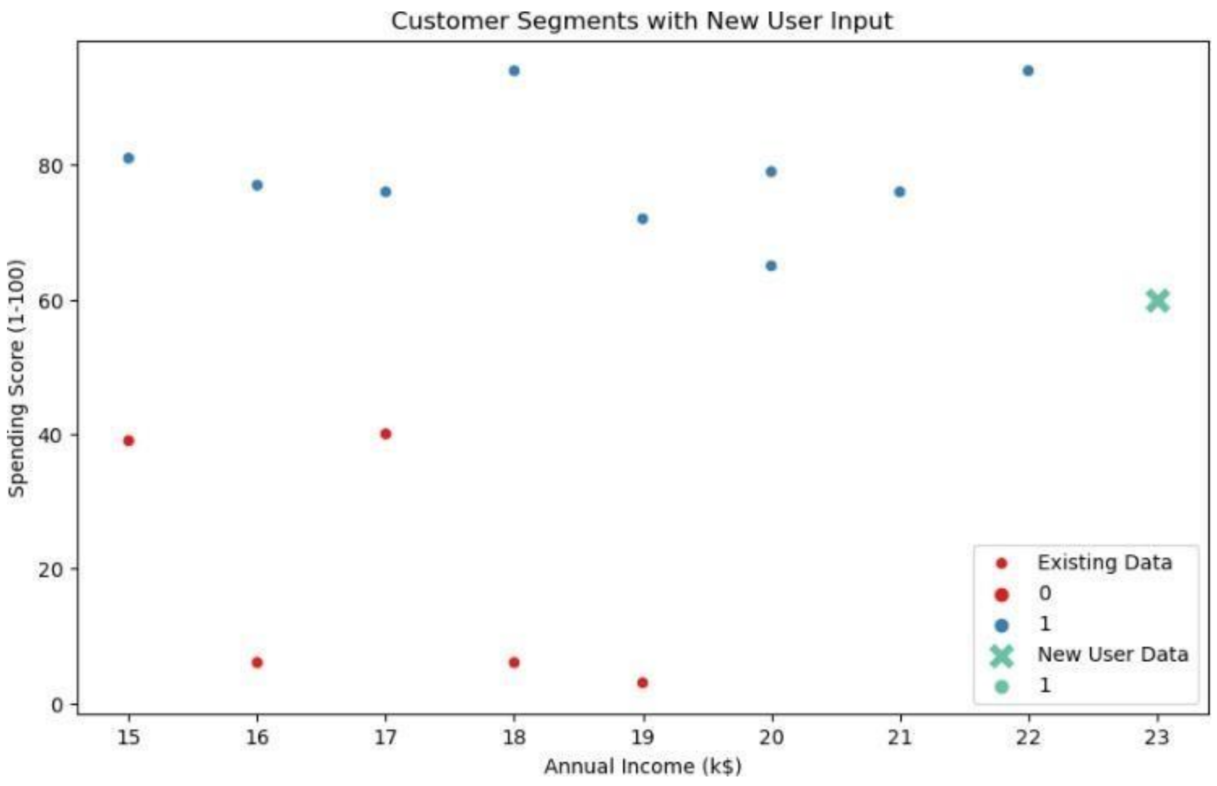
```
# Step 1: Import necessary libraries

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: High-value

}


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_test, y_pred))

print("\nAccuracy Score:")

print(accuracy_score(y_test, y_pred))
```

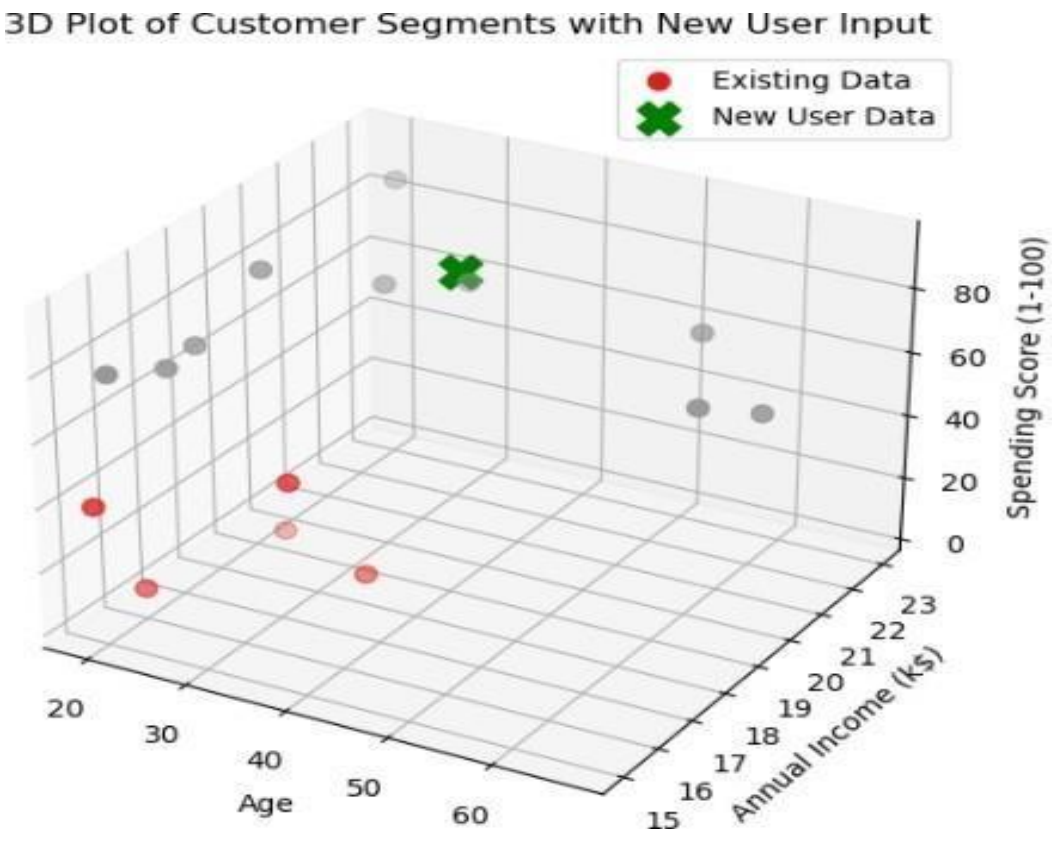
```
# Step 7: Classify new user input

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
```


Customer Segments with New User Input
Spending Score (1-100)
0
20
40
60
80
15
16
17
18
19
20
21
22
23
Annual Income (k$)
Existing Data
0
1
New User Data
1


3D Plot of Customer Segments with New User Input
Existing Data
New User Data
Spending Score (1-100)
0
20
40
60
80
Annual Income (k$)
15
16
17
18
19
20
21
22
23
Age
20
30
40
50
60

Practical 7

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

Theory: The k-Nearest Neighbors (k-NN) algorithm can use various distance metrics to determine the similarity between data points. The most common distance metric is **Euclidean distance**, but there are others such as **Manhattan distance**, **Minkowski distance**, and **Hamming distance**.

Distance Metrics

1. **Euclidean Distance**: Measures the straight-line distance between two points in Euclidean space.

$$d(x,y) = \sqrt{\sum (x_i - y_i)^2}$$

2. **Manhattan Distance**: Measures the distance between two points in a grid-based path (like moving along the axes).

$$d(x,y) = \sum |x_i - y_i|$$

3. **Minkowski Distance**: Generalized distance metric where ppp defines the type of distance. For p=1p = 1p=1, it's Manhattan; for p=2p = 2p=2, it's Euclidean.

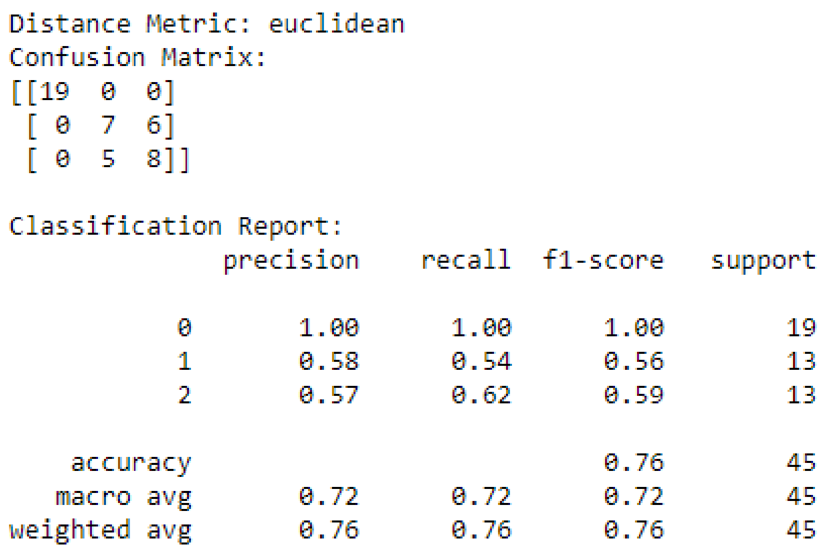
$$d(x,y) = \left(\sum |x_i - y_i|^p\right)^{1/p}$$

4. **Hamming Distance**: Measures the distance between two strings of equal length, counting the positions where the corresponding symbols are different. It is mainly used for categorical data.

Code:-

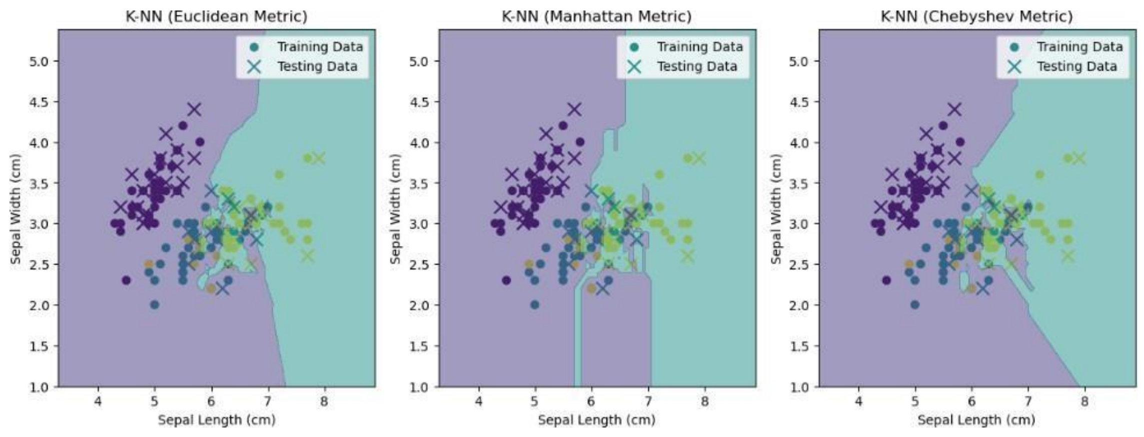
```
import numpy as np

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 is primarily an unsupervised clustering algorithm that partitions data into kkk clusters based on feature similarity. Although it's not inherently a classification method, it can be used for classification tasks by first clustering the data and then assigning labels based on the majority class in each cluster.

Key Concepts

1. **K-Means Algorithm**:
   - Initialize kkk centroids randomly from the dataset.
   - Assign each data point to the nearest centroid.
   - Recalculate the centroids as the mean of all points assigned to each cluster.
   - Repeat the assignment and update steps until convergence (no change in centroids).

2. **Using K-Means for Classification**:
   - Fit K-Means on the training data.
   - Assign cluster labels to the training data.
   - Use a majority vote to assign labels to the clusters.
   - Predict the cluster of test data points and assign the corresponding labels based on majority class.

Code:-

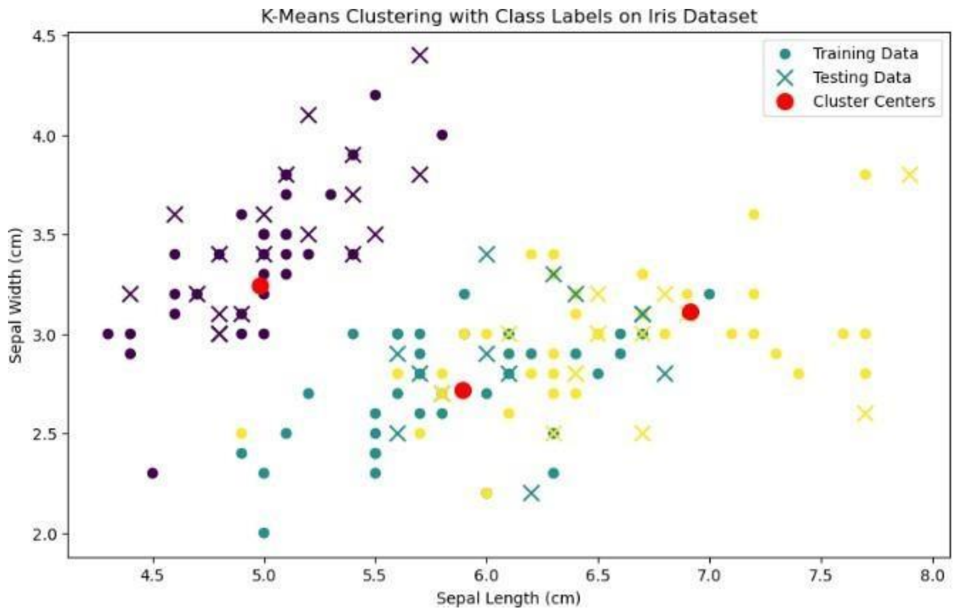
```
import numpy as np

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  5  8]]

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        19
           1       0.62      0.62      0.62        13
           2       0.62      0.62      0.62        13

    accuracy                           0.78        45
   macro avg       0.74      0.74      0.74        45
weighted avg       0.78      0.78      0.78        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 another unsupervised clustering method that builds a hierarchy of clusters. Unlike K-Means, which requires the number of clusters to be specified in advance, hierarchical clustering creates a tree-like structure (dendrogram) that allows you to choose the number of clusters after examining the results.

While hierarchical clustering is not inherently a classification method, we can use it to group similar data points and then assign labels based on majority voting within each cluster.

Key Concepts

1. **Hierarchical Clustering**:
    - It can be agglomerative (bottom-up) or divisive (top-down).
    - Agglomerative clustering starts with each point as a single cluster and merges them based on a distance metric (e.g., Euclidean distance).

2. **Linkage Criteria**: Determines how the distance between clusters is calculated.
    - **Single Linkage**: Distance between the closest points of two clusters.
    - **Complete Linkage**: Distance between the farthest points of two clusters.
    - **Average Linkage**: Average distance between points in two clusters.
    - **Ward's Linkage**: Minimizes the total within-cluster variance.

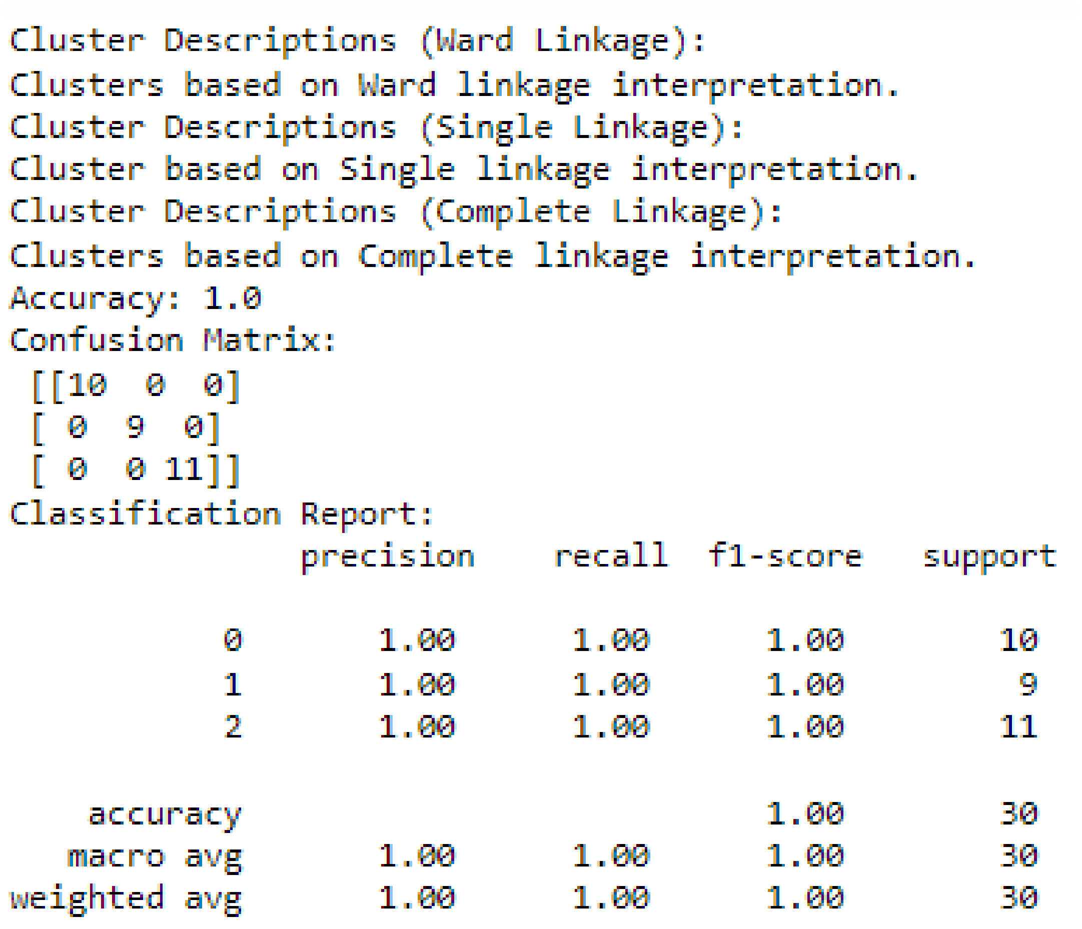
3. **Classification**:
    - After clustering, we assign the true labels to each cluster and use them for predictions.

Code:-

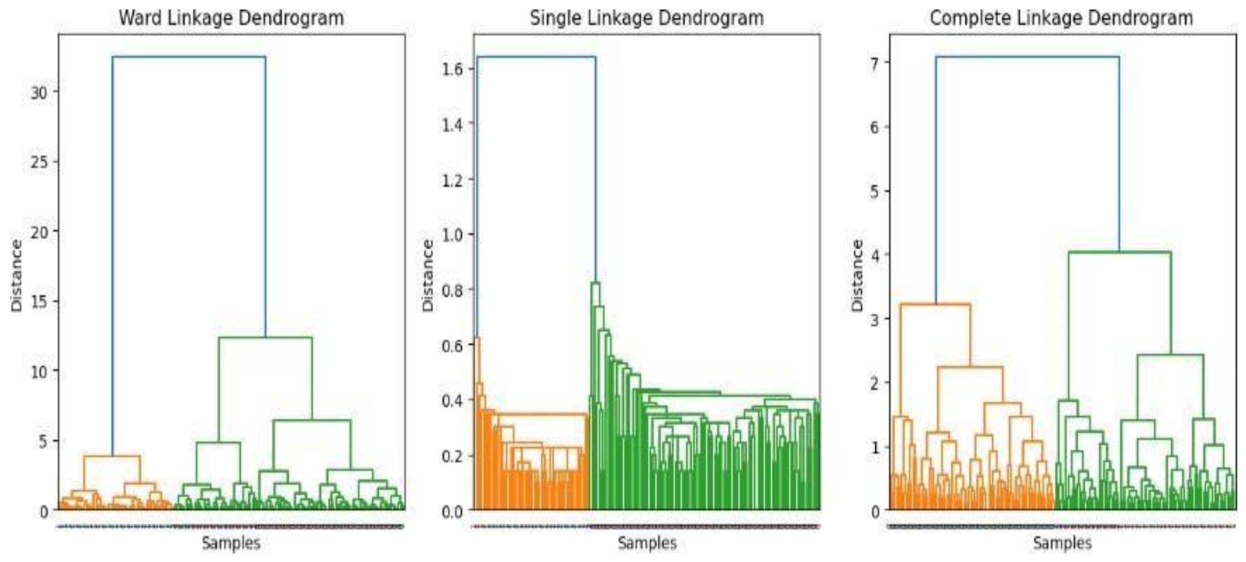
```
import pandas as pd

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
```

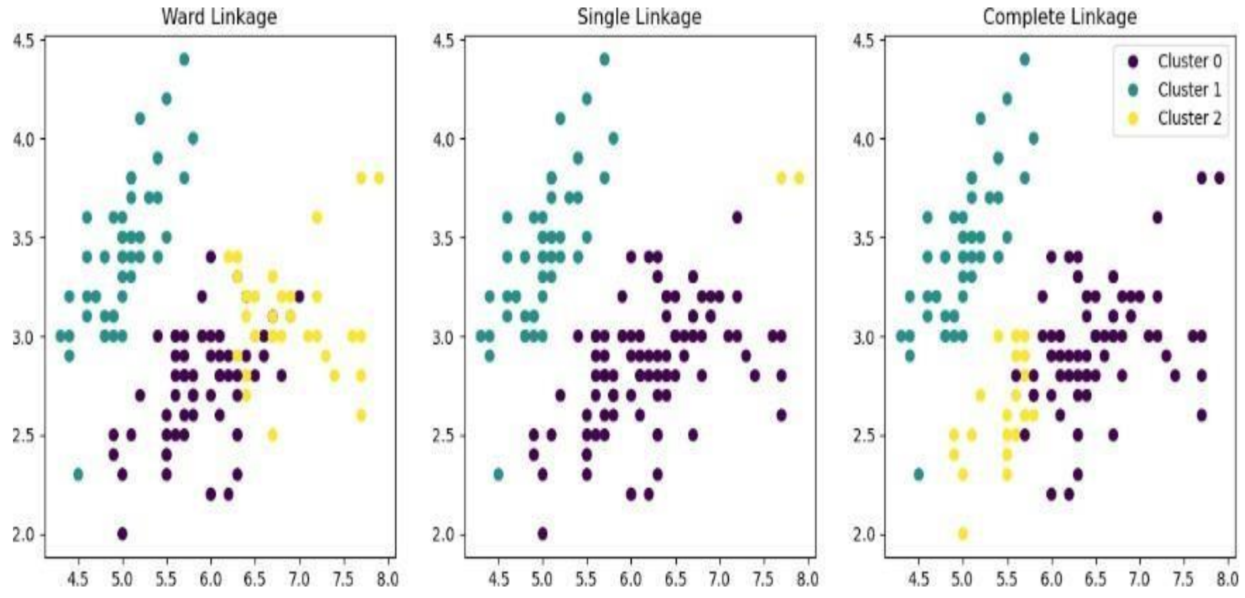
```
plt.figure(figsize=(15, 5))

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 classification is a type of supervised learning where the model makes predictions based on a set of "if-then" rules derived from the training data. These rules can be simple or complex and are often derived from decision trees or created manually based on domain knowledge.

Key Concepts

1. **Rule Creation**: Rules can be generated from training data by identifying patterns and relationships between features and target labels. Common algorithms for generating rules include:
    - **Decision Trees**: Each path from the root to a leaf node represents a rule.
    - **Association Rule Learning**: Such as Apriori or FP-Growth algorithms, which identify relationships between variables.

2. **Rule Evaluation**: Rules can be evaluated based on metrics such as accuracy, precision, recall, and F1-score.

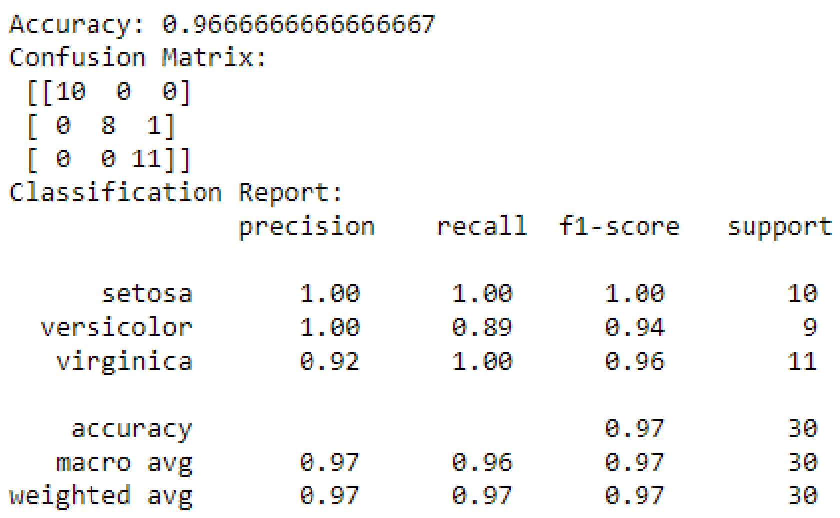
3. **Prediction**: For a new instance, the model checks which rule(s) apply and makes predictions based on those rules.

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: A Bayesian network is a probabilistic graphical model that represents a set of variables and their conditional dependencies using a directed acyclic graph (DAG). In the context of medical diagnosis, a Bayesian network can be used to model the relationships between symptoms, risk factors, and diseases, allowing for probabilistic inference and reasoning.

Key Concepts

1. **Bayesian Network Structure**:
   - Nodes represent random variables (e.g., symptoms, diseases).
   - Directed edges represent conditional dependencies between variables.

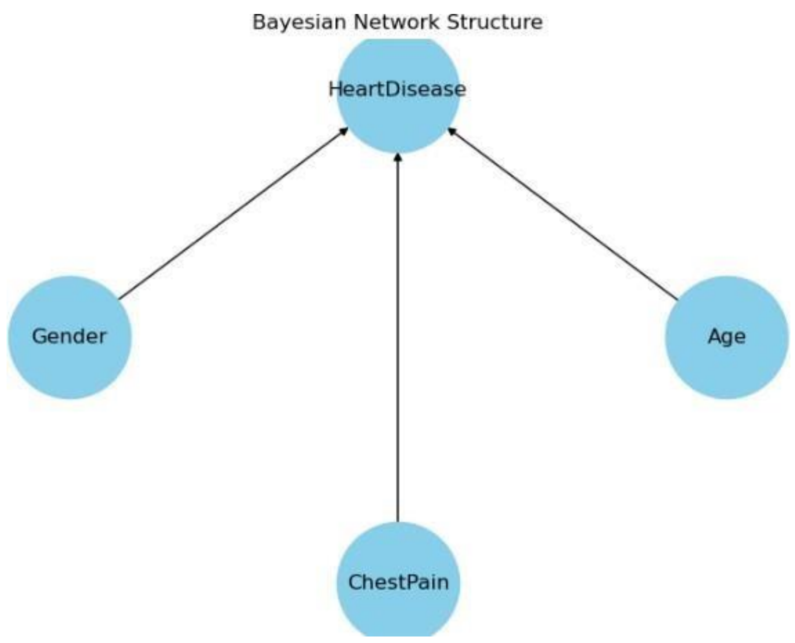
2. **Conditional Probability Tables (CPTs)**: Each node has a CPT that quantifies the effect of its parents on the node.

3. **Inference**: Given evidence (e.g., symptoms), the network can compute the probability of various diseases.

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) is a non-parametric regression technique that fits a model to a subset of the data points that are close to the query point. This method is particularly useful when the relationship between variables is not global but varies in different regions of the input space.

Key Concepts

1. **Non-parametric Nature**: Unlike parametric models that assume a specific form (e.g., linear regression), LWR does not assume a global form for the function being estimated.
2. **Weighting Scheme**: LWR assigns weights to the training examples based on their distance to the query point. Typically, a Gaussian kernel is used for weighting:

   $$w_i = e^{-\frac{(x_i - x)^2}{2\tau^2}}$$

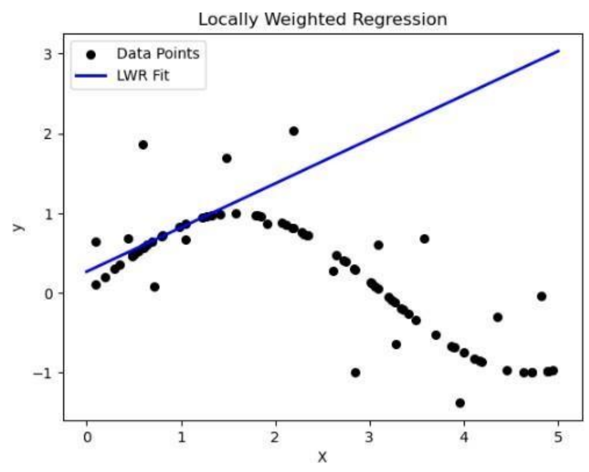
   where Ä\tauÄ controls the bandwidth of the kernel.

3. **Local Model Fitting**: For a query point xxx, a weighted linear regression model is fitted using the nearby points, allowing for different fits in different regions of the input space.

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:-



Practical 13

Aim: Implement ANN to solve the XOR problem using forward/ backward propagation and sigmoid activation function.

Theory: The XOR (exclusive OR) problem is a classic example used to demonstrate the capabilities of neural networks, particularly their ability to model non-linear relationships. The XOR function outputs true (1) if exactly one of the inputs is true (1), and false (0) otherwise.

Key Concepts

1. **Neural Network Structure**:
   - **Input Layer**: Takes in two binary inputs.
   - **Hidden Layer**: Typically contains multiple neurons to capture non-linear patterns.
   - **Output Layer**: Produces a single output indicating the XOR result.

2. **Activation Function**:
   - **Sigmoid Activation Function**: Ã(x)=11+e−x\sigma(x) = \frac{1}{1 + e^{-x}}Ã(x)=1+e−x1 This function squashes output values to a range between 0 and 1, making it suitable for binary classification.

3. **Forward Propagation**: Calculates the output of the network based on the current weights and biases.

4. **Backward Propagation**: Adjusts the weights based on the error between predicted and actual outputs using gradient descent.

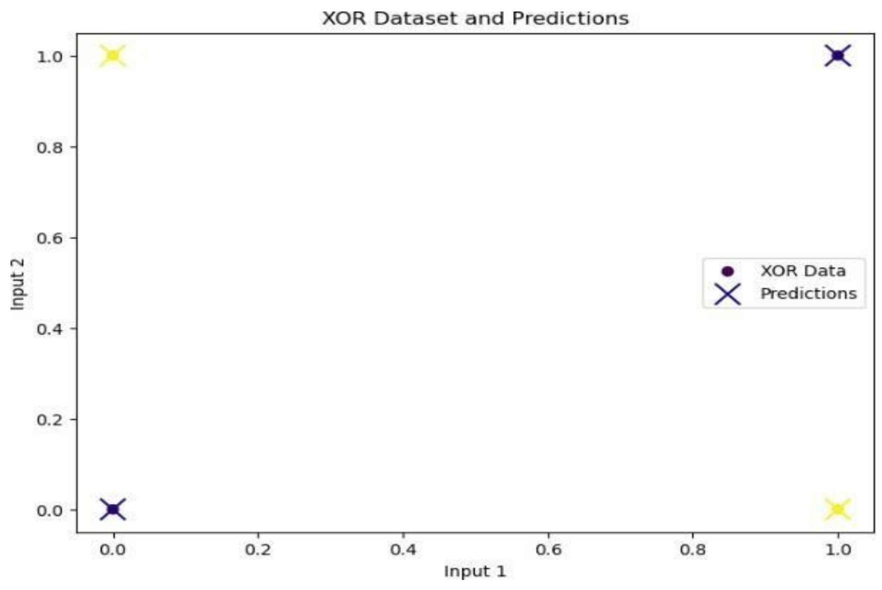
Code:
```
import numpy as np

import matplotlib.pyplot as plt

def sigmoid(x):

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


def sigmoid_derivative(x):

    return x*(1-x)
```

```
class NeuralNetwork:

    def_init_(self, input_size, hidden_size, output_size):

        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):

        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):

        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 = []

        for _ in range(epochs):

            output = self.forward(X)

            error = y-output

            self.loss_history.append(np.mean(error**2))

            self.backward(X,y,learning_rate)

    def predict(self, X):

        return self.forward(X)

X = np.array([[0,0],[0,1],[1,0],[1,1]])

y = np.array([[0], [1], [1], [0]])

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)

predictions = nn.predict(X)

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()

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.]
```


XOR Dataset and Predictions
Input 2
Input 1
● XOR Data
× Predictions