Exercise 4:

IMPLEMENT REGRESSION MODEL WITHOUT USING ML LIBRARIES

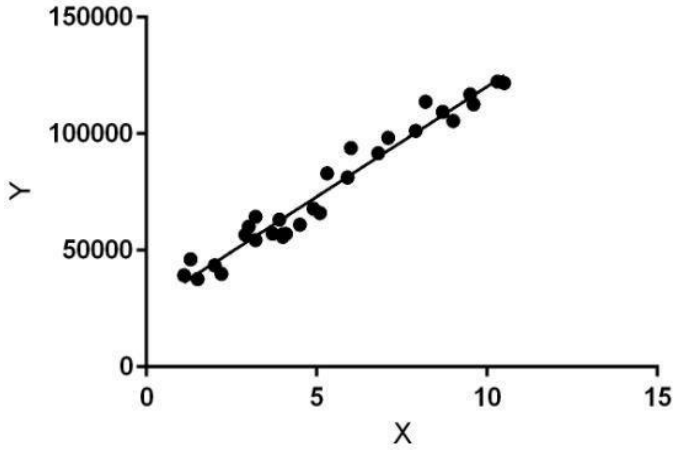
Aim: Implement regression model without using ML libraries.

Software Required: Google Co Lab, Jupyter notebook, Kaggle.

Dataset: https://www.kaggle.com/spscientist/students-performance-in-exams

Theory:

Linear Regression is a machine learning algorithm based on supervised learning. It performs a regression task. Regression models a target prediction value based on independent variables. It is mostly used for finding out the relationship between variables and forecasting. Different regression models differ based on – the kind of relationship between dependent and independent variables they are considering, and the number of independent variables getting used.



Linear regression performs the task to predict a dependent variable value (y) based on a given independent variable (x). So, this regression technique finds out a linear relationship between x (input) and y(output). Hence, the name is Linear Regression.
In the figure above, X (input) is the work experience and Y (output) is the salary of a person. The regression line is the best fit line for our model.

Hypothesis function for Linear Regression:

$$y = \theta_1 + \theta_2.x$$

While training the model we are given:
x: input training data (univariate – one input variable(parameter))
y: labels to data (supervised learning)

When training the model – it fits the best line to predict the value of y for a given value of x. The model gets the best regression fit line by finding the best θ1 and θ2 values.
θ1: intercept
θ2: coefficient of x

Once we find the best θ1 and θ2 values, we get the best fit line. So when we are finally using our model for prediction, it will predict the value of y for the input value of x.

Program :

```
# Import required libraries
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import os


# Read dataset into pandas dataframe
import pandas as pd
data=pd.read_csv('/kaggle/input/students-performance-in-
exams/StudentsPerformance.csv')
data
```

Checking for Null/Missing Values

```
data.isnull().sum()
```

Histogram to check numerical data

```
data.hist()
```

Checking Covariance

```
data.cov()

data.cov()['math  score']

data.cov()['math score']['writing score']

data.corr()

data.corr()['math score']['writing score']
```

Visualizing relationship between reading, writing and math scores

```
import matplotlib.pyplot as plt
plt.scatter(data['math score'],data['writing score'])
plt.xlabel('Maths Score')
plt.ylabel('Writing Score')
plt.title('Covariance='+str(data.cov()['math score']
                                        ['writing score']))
plt.show()
```

```
plt.scatter(data['reading score'],data['writing score'])
plt.xlabel('Reading Score')
plt.ylabel('Writing Score')
plt.title('Covariance='+str(data.cov()['reading score']['writing score']))
plt.show()
```

```
plt.scatter(data['reading score'],data['writing score'])
plt.xlabel('Reading Score')
plt.ylabel('Writing Score')
plt.title('Covariance='+str(data.cov()['reading score']['writing score']))
plt.show()
```

```
plt.scatter(data['reading score'],data['writing score'])
plt.xlabel('Reading Score')
plt.ylabel('Writing Score')
plt.title('Corrlation='+str(data.corr()['reading score']['writing score']))
plt.show()
```

Selecting read score as predictor(x) and writing score(y) as target

```
x=data['reading score'].values
y=data['writing score'].values
```

Splitting Data into Train and Test sets

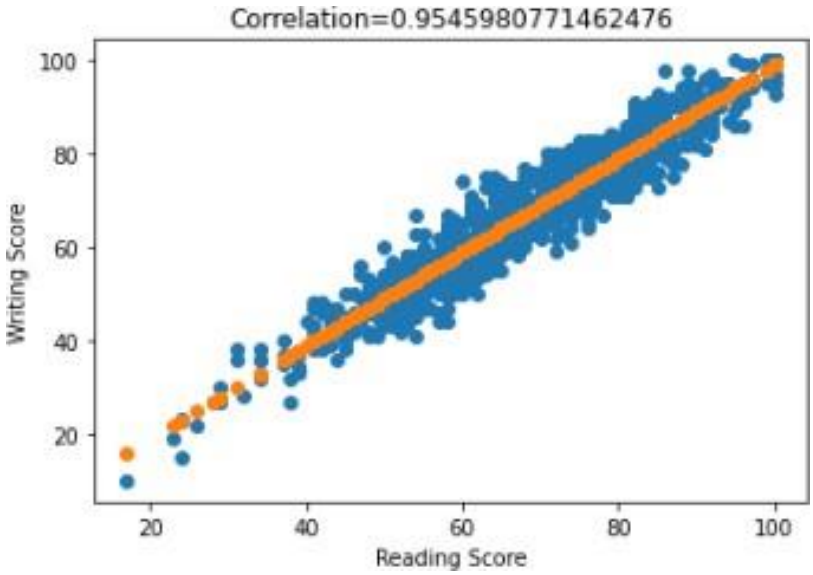
```
from sklearn.model_selection import train_test_split
x_train,x_test,y_train,y_test=train_test_split(x,y,test_size=0.2)
x_train.shape,x_test.shape,y_train.shape,y_test.shape
```

Finding the slope of the line and intercept using traditional statistical formula

```
m=np.cov(x_train,y_train)[0,1]/x_train.var()
c=y_train.mean()-(x_train.mean()*m)
print(m,c)

y_pred=(x_train*m)+c

plt.scatter(x,y)
plt.scatter(x_train,y_pred)
plt.xlabel('Reading Score')
plt.ylabel('Writing Score')
plt.title('Correlation='+str(data.corr()['reading score']
                                        ['writing score']))
plt.show()
```




```
#y_pred-y_train
#np.abs(y_pred-y_train)

np.sum(np.abs(y_pred-y_train))/(x_train.size)
```

```
# Prediction for Test data
y_pred=(x_test*m)+c
np.sum(np.abs(y_pred-y_test))/(x_test.size)

Using Least Squares method

l=x_train.size
x2=x_train**2
y2=y_train**2
xy=x_train*y_train

m= ((l * sum(xy)) - (sum(x_train)*sum(y_train))) / ((l * sum(x2)) -
 sum(x_train)**2)
m
c= y_train.mean()-m*x_train.mean()
c

y_pred=(x_train*m)+c

plt.scatter(x,y)
plt.scatter(x_train,y_pred)
plt.xlabel('Reading Score')
plt.ylabel('Writing Score')
plt.title('Covariance='+str(data.corr()['reading score']['writing score']))
plt.show()

np.sum(np.abs(y_pred-y_train))/(x_train.size)
```

Result: The Mean absolute error for a given dataset using various regression are

MAE in Linear Regression:

MAE in Polynomial Regression:

MAE in Multiple Regression:

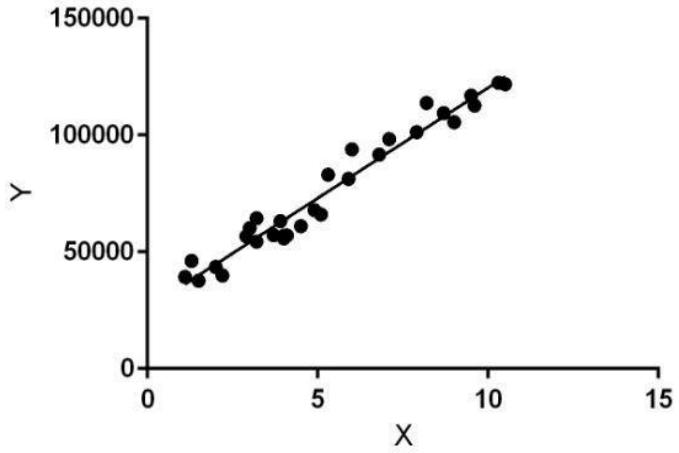
Exercise 5:

VARIOUS REGRESSION MODELS ON GIVEN DATASET

Aim: Apply various regression models on given dataset and find a proper model for prediction with minimal errors.

Software Required: Google Co Lab, Jupyter notebook, Kaggle .

Theory:
Linear Regression is a machine learning algorithm based on supervised learning. It performs a regression task. Regression models a target prediction value based on independent variables. It is mostly used for finding out the relationship between variables and forecasting. Different regression models differ based on – the kind of relationship between dependent and independent variables they are considering, and the number of independent variables getting used.



Linear regression performs the task to predict a dependent variable value (y) based on a given independent variable (x). So, this regression technique finds out a linear relationship between x (input) and y(output). Hence, the name is Linear Regression.
In the figure above, X (input) is the work experience and Y (output) is the salary of a person. The regression line is the best fit line for our model.

Hypothesis function for Linear Regression :

$$y = \theta_1 + \theta_2.x$$

While training the model we are given :
x: input training data (univariate – one input variable(parameter))
y: labels to data (supervised learning)

When training the model – it fits the best line to predict the value of y for a given value of x. The model gets the best regression fit line by finding the best θ1 and θ2 values.
θ1: intercept

θ2: coefficient of x

Once we find the best θ1 and θ2 values, we get the best fit line. So when we are finally using our model for prediction, it will predict the value of y for the input value of x.

Dataset: https://www.kaggle.com/camnugent/california-housing-prices

Program :

```
# Import libraries
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O
import os

#Read dataset into pandas dataframe

data = pd.read_csv('/kaggle/input/housing/housing.csv')
data

data.corr()

x=data.median_income.values.reshape(-1,1)
x

y=data.median_house_value.values.reshape(-1,1)
y

x.max(),x.min(),y.max(),y.min()

Splitting train and test datasets
from sklearn.model_selection import train_test_split
x_train,x_test,y_train,y_test=train_test_split(x,y,test_size=0.2)
x_train.shape,x_test.shape,y_train.shape,y_test.shape


 Building model using sklearn

from sklearn.linear_model import LinearRegression

b. linear Regression
```

```
lm=LinearRegression()

lm.fit(x_train,y_train)

lm.coef_,lm.intercept_

#Prediction for train dataset
y_pred=lm.predict(x_test)
y_pred

print('MAE of regression is:',mean_absolute_error(y_test,y_pred))

import matplotlib.pyplot as plt
plt.scatter(x,y)
plt.scatter(x_test,y_pred)

b. Polynomial Regression

from sklearn.preprocessing import PolynomialFeatures

trans = PolynomialFeatures(degree=2)
x = trans.fit_transform(x)
x.shape

from sklearn.model_selection import train_test_split
x_train,x_test,y_train,y_test=train_test_split(x,y,test_size=0.2)
x_train.shape,y_train.shape, x_test.shape,y_test.shape

from sklearn.linear_model import LinearRegression
lm=LinearRegression()

lm.fit(x_train,y_train)

# Prediction for train data
y_train_pred=lm.predict(x_train)
mean_absolute_error(y_train,y_train_pred)

# Prediction for test dataset
y_test_pred=lm.predict(x_test)
```

```
print('MAE of Polynomial regression is ',mean_absolute_error(
                                                y_test,y_test_pred))
```

c. Multiple Regression

```
#Select features for multiple regression
x=data[['median_income','households','housing_median_age']]
y=data.median_house_value.values.reshape(-1,1)

x.max(),x.min(),y.max(),y.min()

from sklearn.model_selection import train_test_split
x_train,x_test,y_train,y_test=train_test_split(x,y,test_size=0.2)
x_train.shape,y_train.shape, x_test.shape,y_test.shape

x_train.max(),y_train.max(), x_test.max(),y_test.max()

from sklearn.linear_model import LinearRegression
lm=LinearRegression()
lm.fit(x_train,y_train)

#Prediction for train data
y_train_pred=lm.predict(x_train)
mean_absolute_error(y_train,y_train_pred)

#prediction for test dataset
y_test_pred=lm.predict(x_test)
print('MAE of Multiple regressions is ',mean_absolute_error(
                                                y_test,y_test_pred))
```

Result: The Mean absolute error for a given dataset using various regression are

MAE in Linear Regression:

MAE in Polynomial Regression:

MAE in Multiple Regression:

Exercise 6:

IMPLEMENTATION LOGISTIC REGRESSION MODEL

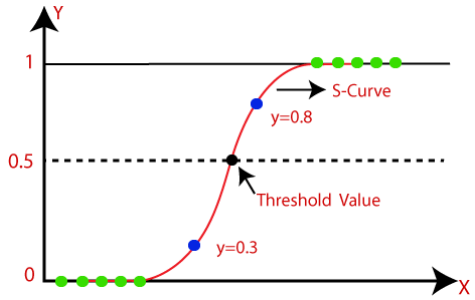
Aim: Implement a logistic regression model on given dataset and check the accuracy for test dataset.

Software Required: Google Co Lab, Jupyter notebook, Kaggle .

Dataset: https://www.kaggle.com/datasets/uciml/iris

Theory:

- Logistic regression is one of the most popular Machine Learning algorithms, which comes under the Supervised Learning technique. It is used for predicting the categorical dependent variable using a given set of independent variables.
- Logistic regression predicts the output of a categorical dependent variable. Therefore the outcome must be a categorical or discrete value. It can be either Yes or No, 0 or 1, true or False, etc. but instead of giving the exact value as 0 and 1, **it gives the probabilistic values which lie between 0 and 1**.
- Logistic Regression is much similar to the Linear Regression except that how they are used. Linear Regression is used for solving Regression problems, whereas **Logistic regression is used for solving the classification problems**.
- In Logistic regression, instead of fitting a regression line, we fit an "S" shaped logistic function, which predicts two maximum values (0 or 1).
- The curve from the logistic function indicates the likelihood of something such as whether the cells are cancerous or not, a mouse is obese or not based on its weight, etc.
- Logistic Regression is a significant machine learning algorithm because it has the ability to provide probabilities and classify new data using continuous and discrete datasets.
- Logistic Regression can be used to classify the observations using different types of data and can easily determine the most effective variables used for the classification. The below image is showing the logistic function:



Logistic Function (Sigmoid Function):

- The sigmoid function is a mathematical function used to map the predicted values to probabilities.
- It maps any real value into another value within a range of 0 and 1.
- The value of the logistic regression must be between 0 and 1, which cannot go beyond this limit, so it forms a curve like the "S" form. The S-form curve is called the Sigmoid function or the logistic function.
- In logistic regression, we use the concept of the threshold value, which defines the probability of either 0 or 1. Such as values above the threshold value tends to 1, and a value below the threshold values tends to 0.
- $y = b_0 + b_1x_1 + b_2x_2 + b_3x_3 + \cdots + b_nx_n$
- $\frac{y}{1-y}$; 0 for y= 0, and infinity for y=1
- $log\left[\frac{y}{1-y}\right] = b_0 + b_1x_1 + b_2x_2 + b_3x_3 + \cdots + b_nx_n$

Type of Logistic Regression:

On the basis of the categories, Logistic Regression can be classified into three types:

- **Binomial:** In binomial Logistic regression, there can be only two possible types of the dependent variables, such as 0 or 1, Pass or Fail, etc.
- **Multinomial:** In multinomial Logistic regression, there can be 3 or more possible unordered types of the dependent variable, such as "cat", "dogs", or "sheep"
- **Ordinal:** In ordinal Logistic regression, there can be 3 or more possible ordered types of dependent variables, such as "low", "Medium", or "High".

Program :

```
#Import required libraries
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O
import os

#Read dataset into pandas dataframe.
data=pd.read_csv('/kaggle/input/iris/Iris.csv')

data.head()

# Display the datatype of each column
data.dtypes

# Create a column cat_code which represents the numerical values for Species
column.
data['cat_code']=data.Species.astype('category').cat.codes

data.head()

data.columns

#Select the input features as x
x=data[['SepalLengthCm', 'SepalWidthCm', 'PetalLengthCm','PetalWidthCm']]

#Select the target
y=data['cat_code']

# Visualize the 3 categories of iris flower with respective to
Sepal Length and Sepal Width
import matplotlib.pyplot as plt
plt.scatter(x['SepalLengthCm'],x['SepalWidthCm'],c=y)


# Visualize the 3 categories of iris flower with respective to
Petal Length and Petal Width

plt.scatter(x['PetalLengthCm'],x['PetalWidthCm'],c=y)

x.shape,y.shape
# Splitting train and test datasets
```

```
from sklearn.model_selection import train_test_split
x_train,x_test,y_train,y_test=train_test_split(x,y,test_size=0.2,shuffle=True
)
x_train.shape,x_test.shape,y_train.shape,y_test.shape

# Fit the logistic regression model with SKlearn library
from sklearn.linear_model import LogisticRegression
lm=LogisticRegression()
lm.fit(x_train,y_train)

# Prediction for train set
y_pred = lm.predict(x_train)
y_train.values
y_pred

# Accuracy for train dataset
from sklearn.metrics import accuracy_score
# Prediction for test dataset and Accuracy
y_pred = lm.predict(x_test)
accuracy_score(y_test,y_pred)
```

Result: The actual train values, predicted values and accuracy score are

Accuracy_score:

Exercise 7:

BUILD A DECISION TREE MODEL

Aim: Build a decision tree model for given dataset to predict the target with best accuracy.

Software Required: Google Co Lab, Jupyter notebook, Kaggle .

Theory:

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

For instance, in the example below, decision trees learn from data to approximate a sine curve with a set of if-then-else decision rules. The deeper the tree, the more complex the decision rules and the fitter the model.

Some advantages of decision trees are:

- Simple to understand and to interpret. Trees can be visualized.
- Requires little data preparation. Other techniques often require data normalization, dummy variables need to be created and blank values to be removed. Note however that this module does not support missing values.

- The cost of using the tree (i.e., predicting data) is logarithmic in the number of data points used to train the tree.
- Able to handle both numerical and categorical data. However scikit-learn implementation does not support categorical variables for now. Other techniques are usually specialized in analyzing datasets that have only one type of variable. See algorithms for more information.
- Able to handle multi-output problems.
- Uses a white box model. If a given situation is observable in a model, the explanation for the condition is easily explained by boolean logic. By contrast, in a black box model (e.g., in an artificial neural network), results may be more difficult to interpret.
- Possible to validate a model using statistical tests. That makes it possible to account for the reliability of the model.
- Performs well even if its assumptions are somewhat violated by the true model from which the data were generated.
-

The disadvantages of decision trees include:

- Decision-tree learners can create over-complex trees that do not generalize the data well. This is called overfitting. Mechanisms such as pruning, setting the minimum number of samples required at a leaf node or setting the maximum depth of the tree are necessary to avoid this problem.
- Decision trees can be unstable because small variations in the data might result in a completely different tree being generated. This problem is mitigated by using decision trees within an ensemble.
- Predictions of decision trees are neither smooth nor continuous, but piecewise constant approximations as seen in the above figure. Therefore, they are not good at extrapolation.
- The problem of learning an optimal decision tree is known to be NP-complete under several aspects of optimality and even for simple concepts. Consequently, practical decision-tree learning algorithms are based on heuristic algorithms such as the greedy algorithm where locally optimal decisions are made at each node. Such algorithms cannot guarantee to return the globally optimal decision tree. This can be mitigated by training multiple trees in an ensemble learner, where the features and samples are randomly sampled with replacement.
- There are concepts that are hard to learn because decision trees do not express them easily, such as XOR, parity or multiplexer problems.
- Decision tree learners create biased trees if some classes dominate. It is therefore recommended to balance the dataset prior to fitting with the decision tree.

Program :

```
        import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
#numpy and pandas initialization
import numpy as np
import pandas as pd
PlayTennis = pd.read_csv("tennis.csv")
#numerical values into numerical values using LabelEncoder
from sklearn.preprocessing import LabelEncoder
Le = LabelEncoder()
PlayTennis['outlook'] = Le.fit_transform(PlayTennis['outlook'])
PlayTennis['temp'] = Le.fit_transform(PlayTennis['temp'])
PlayTennis['humidity'] = Le.fit_transform(PlayTennis['humidity'])
PlayTennis['windy'] = Le.fit_transform(PlayTennis['windy'])
PlayTennis['play'] = Le.fit_transform(PlayTennis['play'])
```

```
features_cols=['outlook','temp','humidity','windy']
x=PlayTennis[features_cols]
y=PlayTennis.play
from sklearn.model_selection import train_test_split
x_train, x_test, y_train,y_test=train_test_split(x,y,test_size=0.2)
from sklearn.tree import DecisionTreeClassifier
#classifier=DecisionTreeClassifier(criterion='gini')
classifier=DecisionTreeClassifier(criterion='entropy')
classifier.fit(x_train,y_train)
DecisionTreeClassifier(criterion='entropy')
classifier.predict(x_test)
classifier.score(x_test,y_test)
from sklearn import tree
clf = tree.DecisionTreeClassifier(criterion = 'entropy')
clf = clf.fit(x, y)
tree.plot_tree(clf)
```

Result:

Exercise 8:

KNN MODEL FOR CLASSIFICATION

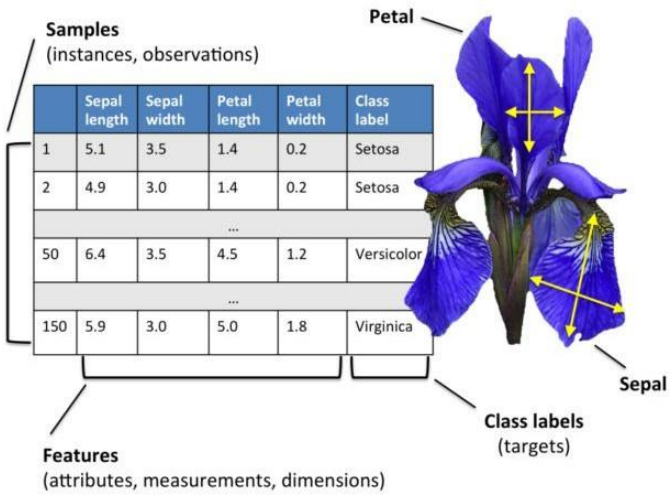
Aim: Implement KNN model to classify the target in given dataset. calculate the Accuracy & confusion matrix

Software Required: Google Co Lab, Jupyter notebook, Kaggle .

DataSet: https://www.kaggle.com/datasets/uciml/iris

Iris Plants Dataset: Dataset contains 150 instances (50 in each of three classes) Number of Attributes: 4 numeric, predictive attributes and the Class.

The Iris dataset was used in R.A. Fisher's classic 1936 paper, The Use of Multiple Measurements in Taxonomic Problems, and can also be found on the UCI Machine Learning Repository.
It includes three iris species with 50 samples each as well as some properties about each flower.
One flower species is linearly separable from the other two, but the other two are not linearly separable from each other.
The columns in this dataset are:
Id SepalLengthCm SepalWidthCm PetalLengthCm PetalWidthCm Species



	Sepal length	Sepal width	Petal length	Petal width	Class label
1	5.1	3.5	1.4	0.2	Setosa
2	4.9	3.0	1.4	0.2	Setosa
...					
50	6.4	3.5	4.5	1.2	Versicolor
...					
150	5.9	3.0	5.0	1.8	Virginica

Theory:

Training algorithm:

For each training example (x, f (x)), add the example to the list training examples Classification algorithm:

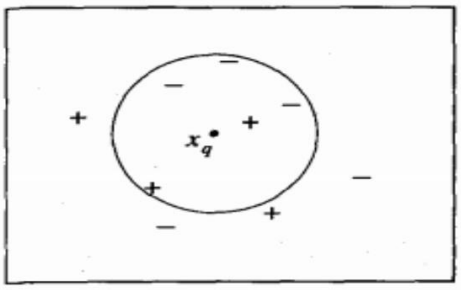
Given a query instance xq to be classified,

Let x1 . . .xk denote the k instances from training examples that are nearest to xq

Return Where, f(xi) function to calculate the mean value of the k nearest training examples.

$$\hat{f}(x_q) \leftarrow \frac{\sum_{i=1}^{k} f(x_i)}{k}$$

k-NEAREST NEIGHBOR.
A set of positive and negative training examples is shown on the right, along with a query instance x q, to be classified.



Program :

```
#Import required libraries
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O
import os
#Read dataset into pandas dataframe.
data=pd.read_csv('Iris.csv')
data.head()
#Display the datatype of each column
data.dtypes
data['cat_code']=data.Species.astype('category').cat.codes
data.cat_code.unique()
data.Species.unique()
#Select the input features as x
x=data[['SepalLengthCm', 'SepalWidthCm', 'PetalLengthCm', 'PetalWidthCm']]
#Select the target
y=data['cat_code']
# Visualize the iris flower with respective to Sepal Length and Sepal Width
import matplotlib.pyplot as plt
plt.scatter(x['SepalLengthCm'],x['SepalWidthCm'],c=y)
#Splitting train and test datasets
from sklearn.model_selection import train_test_split
x_train,x_test,y_train,y_test=train_test_split(x,y,test_size=0.2,shuffle=True)
x_train.shape,x_test.shape,y_train.shape,y_test.shape
from sklearn.neighbors import KNeighborsClassifier
from sklearn import metrics
KNN=KNeighborsClassifier(n_neighbors=3)
KNN.fit(x_train,y_train)
prediction=KNN.predict(x_test)
print('The accuracy of the KNN is',metrics.accuracy_score(prediction,y_test)*100,'percent')
#import confusion_matrix
from sklearn.metrics import confusion_matrix
prediction=KNN.predict(x_test)
confusion_matrix(y_test,prediction)
newval=pd.DataFrame({'SepalLengthCm':[5.1],'SepalWidthCm':[3.5],'PetalLengthCm':[1.4],'PetalWidthCm':[0.2]})
print("predicted flower is")
KNN.predict(newval)
```

Result:
The accuracy of the KNN is :
Confusion matrix is

Training-set accuracy score:

Testing-set accuracy score:

Exercise 9:

DEMONSTRATE REGRESSION AND CLASSIFICATION METRICS

Aim: Demonstrate regression and classification metrics using sample data.

Software Required: Google Co Lab, Jupyter notebook, Kaggle .

Theory:

The confusion matrix is a matrix used to determine the performance of the classification models for a given set of test data. It can only be determined if the true values for test data are known. The matrix itself can be easily understood, but the related terminologies may be confusing. Since it shows the errors in the model performance in the form of a matrix, hence also known as an **error matrix**. Some features of Confusion matrix are given below:

- For the 2 prediction classes of classifiers, the matrix is of 2*2 table, for 3 classes, it is 3*3 table, and so on.
- The matrix is divided into two dimensions, that are **predicted values** and **actual values** along with the total number of predictions.
- Predicted values are those values, which are predicted by the model, and actual values are the true values for the given observations.
- It looks like the below table:

n = total predictions	Actual: No	Actual: Yes
Predicted: No	True Negative	False Positive
Predicted: Yes	False Negative	True Positive

The above table has the following cases:

- **True Negative:** Model has given prediction No, and the real or actual value was also No.
- **True Positive:** The model has predicted yes, and the actual value was also true.
- **False Negative:** The model has predicted no, but the actual value was Yes, it is also called as **Type-II error**.
- **False Positive:** The model has predicted Yes, but the actual value was No. It is also called a **Type-I error.**

Need for Confusion Matrix in Machine learning

- It evaluates the performance of the classification models, when they make predictions on test data, and tells how good our classification model is.
- It not only tells the error made by the classifiers but also the type of errors such as it is either type-I or type-II error.
- With the help of the confusion matrix, we can calculate the different parameters for the model, such as accuracy, precision, etc.

Example: We can understand the confusion matrix using an example.

Suppose we are trying to create a model that can predict the result for the disease that is either a person has that disease or not. So, the confusion matrix for this is given as:

n = 100	Actual: No	Actual: Yes	
Predicted: No	TN: 65	FP: 3	68
Predicted: Yes	FN: 8	TP: 24	32
	73	27	

From the above example, we can conclude that:

- The table is given for the two-class classifier, which has two predictions "Yes" and "NO." Here, Yes defines that patient has the disease, and No defines that patient does not has that disease.
- The classifier has made a total of **100 predictions**. Out of 100 predictions, **89 are true predictions**, and **11 are incorrect predictions**.
- The model has given prediction "yes" for 32 times, and "No" for 68 times. Whereas the actual "Yes" was 27, and actual "No" was 73 times.

Calculations using Confusion Matrix:

We can perform various calculations for the model, such as the model's accuracy, using this matrix. These calculations are given below:

- **Classification Accuracy:** It is one of the important parameters to determine the accuracy of the classification problems. It defines how often the model predicts the correct output. It can be calculated as the ratio of the number of correct predictions made by the classifier

to all number of predictions made by the classifiers. The formula is given below:

$$\text{Accuracy}=\frac{TP+TN}{TP+FP+FN+TN}$$

- **Misclassification rate:** It is also termed as Error rate, and it defines how often the model gives the wrong predictions. The value of error rate can be calculated as the number of incorrect predictions to all number of the predictions made by the classifier. The formula is given below:

$$\text{Error rate}=\frac{FP+FN}{TP+FP+FN+TN}$$

- **Precision:** It can be defined as the number of correct outputs provided by the model or out of all positive classes that have predicted correctly by the model, how many of them were actually true. It can be calculated using the below formula:

$$\text{Precision}=\frac{TP}{TP+FP}$$

- **Recall:** It is defined as the out of total positive classes, how our model predicted correctly. The recall must be as high as possible.

$$\text{Recall}=\frac{TP}{TP+FN}$$

- **F-measure:** If two models have low precision and high recall or vice versa, it is difficult to compare these models. So, for this purpose, we can use F-score. This score helps us to evaluate the recall and precision at the same time. The F-score is maximum if the recall is equal to the precision. It can be calculated using the below formula:

$$\text{F-measure}=\frac{2*Recall*Precision}{Recall+Precision}$$

Other important terms used in Confusion Matrix:

- **Null Error rate:** It defines how often our model would be incorrect if it always predicted the majority class. As per the accuracy paradox, it is said that "*the best classifier has a higher error rate than the null error rate.*"
- **ROC Curve:** The ROC is a graph displaying a classifier's performance for all possible thresholds. The graph is plotted between the true positive rate (on the Y-axis) and the false Positive rate (on the x-axis).

Program :

```
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score,precision_score,recall_score,preci
sion_recall_curve,roc_curve,r2_score,f1_score
import matplotlib.pyplot as plt

y=[1,1,0,0,1,1,1,1,0,0,1,0,0,0,1,1,0,1]
y1=[1,1,0,0,1,1,1,1,0,0,1,0,0,0,1,1,0,1]
confusion_matrix(y,y1)
print('Accuracy Score: ', accuracy_score(y,y1))
print('Precision: ', precision_score(y,y1))
print('Recall:  ',  recall_score(y,y1))
print('F1 Score: ',f1_score(y,y1))
print('R2 Score: ',r2_score(y,y1))
print('Precision-Recall Curve: ',precision_recall_curve(y,y1))
print('ROC Curve: ',roc_curve(y,y1))
roc=roc_curve(y,y1)
plt.plot(roc[0],roc[1])

pr=precision_recall_curve(y,y1)
plt.plot(pr[0],pr[1])

y=[1,1,0,0,1,1,1,1,0,0,1,0,0,0,1,1,0,1]
y1=[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
confusion_matrix(y,y1)
print('Accuracy Score: ', accuracy_score(y,y1))
print('Precision: ', precision_score(y,y1))
print('Recall:  ',  recall_score(y,y1))
print('F1 Score: ',f1_score(y,y1))
print('R2 Score: ',r2_score(y,y1))
print('Precision-Recall Curve: ',precision_recall_curve(y,y1))
print('ROC Curve: ',roc_curve(y,y1))
roc=roc_curve(y,y1)
plt.plot(roc[0],roc[1])

pr=precision_recall_curve(y,y1)
plt.plot(pr[0],pr[1])

y=[1,1,0,0,1,1,1,1,0,0,1,0,0,0,1,1,0,1]
y1=[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]
confusion_matrix(y,y1)
print('Accuracy Score: ', accuracy_score(y,y1))
print('Precision: ', precision_score(y,y1))
print('Recall:  ',  recall_score(y,y1))
print('F1 Score: ',f1_score(y,y1))
print('R2 Score: ',r2_score(y,y1))
print('Precision-Recall Curve: ',precision_recall_curve(y,y1))
print('ROC Curve: ',roc_curve(y,y1))
```

```
roc=roc_curve(y,y1)
plt.plot(roc[0],roc[1])

pr=precision_recall_curve(y,y1)
plt.plot(pr[0],pr[1])

y=[1,1,0,0,1,1,1,1,0,0,1,0,0,0,1,1,0,1]
y1=[0,0,0,0,1,1,1,1,1,0,1,0,0,0,1,1,1,1]
confusion_matrix(y,y1)
print('Accuracy Score: ', accuracy_score(y,y1))
print('Precision: ', precision_score(y,y1))
print('Recall:  ',  recall_score(y,y1))
print('F1 Score: ',f1_score(y,y1))
print('R2 Score: ',r2_score(y,y1))
print('Precision-Recall Curve: ',precision_recall_curve(y,y1))
print('ROC Curve: ',roc_curve(y,y1))
roc=roc_curve(y,y1)
plt.plot(roc[0],roc[1])

pr=precision_recall_curve(y,y1)
plt.plot(pr[0],pr[1])

y=[1,1,0,0,1,1,1,1,0,0,1,0,0,0,1,1,0,1]
y1=[0,0,1,1,0,0,0,0,1,1,0,1,1,1,0,0,1,0]
confusion_matrix(y,y1)
print('Accuracy Score: ', accuracy_score(y,y1))
print('Precision: ', precision_score(y,y1))
print('Recall:  ',  recall_score(y,y1))
print('F1 Score: ',f1_score(y,y1))
print('R2 Score: ',r2_score(y,y1))
print('Precision-Recall Curve: ',precision_recall_curve(y,y1))
print('ROC Curve: ',roc_curve(y,y1))
roc=roc_curve(y,y1)
plt.plot(roc[0],roc[1])

pr=precision_recall_curve(y,y1)
plt.plot(pr[0],pr[1])
```

RESULT: The Performance Metrics for a classifier are

Accuracy=
Precision
Recall=
F1 score=
R2 score=

Exercise 10:

SUPPORT VECTOR MACHINE MODEL

Aim: Build SVM model with various kernels and select best kernel for given dataset.

Software Required: Google Co Lab, Jupyter notebook, Kaggle.

Dataset: https://www.kaggle.com/datasets/uciml/human-activity-recognition-with-smartphones

About Dataset: The Human Activity Recognition database was built from the recordings of 30 study participants performing activities of daily living (ADL) while carrying a waist-mounted smartphone with embedded inertial sensors. The objective is to classify activities into one of the six activities performed.

Description of experiment The experiments have been carried out with a group of 30 volunteers within an age bracket of 19-48 years. Each person performed six activities (WALKING, WALKINGUPSTAIRS, WALKINGDOWNSTAIRS, SITTING, STANDING, LAYING) wearing a smartphone (Samsung Galaxy S II) on the waist. Using its embedded accelerometer and gyroscope, we captured 3-axial linear acceleration and 3-axial angular velocity at a constant rate of 50Hz. The experiments have been video-recorded to label the data manually. The obtained dataset has been randomly partitioned into two sets, where 70% of the volunteers was selected for generating the training data and 30% the test data.

The sensor signals (accelerometer and gyroscope) were pre-processed by applying noise filters and then sampled in fixed-width sliding windows of 2.56 sec and 50% overlap (128 readings/window). The sensor acceleration signal, which has gravitational and body motion components, was separated using a Butterworth low-pass filter into body acceleration and gravity. The gravitational force is assumed to have only low frequency components, therefore a filter with 0.3 Hz cutoff frequency was used. From each window, a vector of features was obtained by calculating variables from the time and frequency domain.

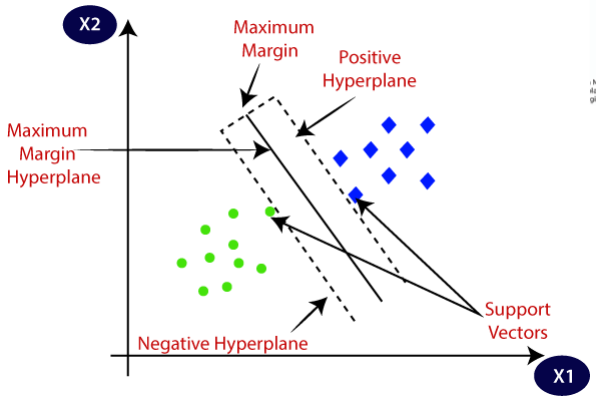
Attribute information
For each record in the dataset the following is provided:
Triaxial acceleration from the accelerometer (total acceleration) and the estimated body acceleration.
Triaxial Angular velocity from the gyroscope.
A 561-feature vector with time and frequency domain variables.
Its activity label.
An identifier of the subject who carried out the experiment.

Theory:

Support Vector Machine or SVM is one of the most popular Supervised Learning algorithms, which is used for Classification as well as Regression problems. However, primarily, it is used for Classification problems in Machine Learning.

The goal of the SVM algorithm is to create the best line or decision boundary that can segregate n-dimensional space into classes so that we can easily put the new data point in the correct category in the future. This best decision boundary is called a hyperplane.

SVM chooses the extreme points/vectors that help in creating the hyperplane. These extreme cases are called as support vectors, and hence algorithm is termed as Support Vector Machine. Consider the below diagram in which there are two different categories that are classified using a decision boundary or hyperplane:



Example: SVM can be understood with the example that we have used in the KNN classifier. Suppose we see a strange cat that also has some features of dogs, so if we want a model that can accurately identify whether it is a cat or dog, so such a model can be created by using the SVM algorithm. We will first train our model with lots of images of cats and dogs so that it can learn about different features of cats and dogs, and then we test it with this strange creature. So as support vector creates a decision boundary between these two data (cat and dog) and choose extreme cases (support vectors), it will see the extreme case of cat and dog. On the basis of the support vectors, it will classify it as a cat.
Consider the below diagram:
Types of SVM
SVM can be of two types:

- Linear SVM: Linear SVM is used for linearly separable data, which means if a dataset can be classified into two classes by using a single straight line, then such data is termed as linearly separable data, and classifier is used called as Linear SVM classifier.
- Non-linear SVM: Non-Linear SVM is used for non-linearly separated data, which means if a dataset cannot be classified by using a straight line, then such data is termed as non-linear data and classifier used is called as Non-linear SVM classifier.

Hyperplane and Support Vectors in the SVM algorithm:

Hyperplane: There can be multiple lines/decision boundaries to segregate the classes in n-dimensional space, but we need to find out the best decision boundary that helps to classify the data points. This best boundary is known as the hyperplane of SVM.

The dimensions of the hyperplane depend on the features present in the dataset, which means if there are 2 features (as shown in image), then hyperplane will be a straight line.

And if there are 3 features, then hyperplane will be a 2-dimension plane.

We always create a hyperplane that has a maximum margin, which means the maximum distance between the data points.

Support Vectors:

The data points or vectors that are the closest to the hyperplane and which affect the position of the hyperplane are termed as Support Vector. Since these vectors support the hyperplane, hence called a Support vector.

Program :

```
#Import required basic libraries
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import os

# Read dataset into dataframe
data=pd.read_csv('/kaggle/input/human-activity-recognition-with-
smartphones/train.csv')
data.describe()
data

data.info()

data.columns

data.Activity.value_counts()

data.shape

data['activity_code'] = data.Activity.astype('category').cat.codes

data.shape

data.activity_code

data1=data.drop('Activity',axis=1)
data1.shape

x_col=data1.columns.to_list()
x_col.pop(-1)
x_data=data1[x_col]
y_col='activity_code'
from sklearn.model_selection import train_test_split
train_x,test_x,train_y,test_y = train_test_split(data1[x_col],data1[y_col].values,te
st_size=0.1)

train_x.shape,test_x.shape,train_y.shape,test_y.shape
```

```
test_y
```

```
# Try to build Logistic regression
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
lg=LogisticRegression()#max_iter=800, solver='sag')
lg.fit(train_x,train_y)
train_y_pred=lg.predict(train_x)
test_y_pred= lg.predict(test_x)
#
print("Training Accuracy",accuracy_score(train_y,train_y_pred))
print("Testing Accuracy", accuracy_score(test_y,test_y_pred))

from sklearn.metrics import confusion_matrix
confusion_matrix(train_y,train_y_pred)

confusion_matrix(test_y,test_y_pred)
```

Support Vector Classifier:

SVC(*, C=1.0, kernel="rbf", degree=3, gamma="scale", coef0=0.0, shrinking=True, probability=False, tol=1e-3, cache_size=200, class_weight=None, verbose=False, max_iter=-1, decision_function_shape="ovr", break_ties=False, random_state=None)

Parameters

C : float, default=1.0 Regularization parameter. The strength of the regularization is inversely proportional to C. Must be strictly positive. The penalty is a squared l2 penalty.

kernel : {'linear', 'poly', 'rbf', 'sigmoid', 'precomputed'} or callable, default='rbf' Specifies the kernel type to be used in the algorithm. If none is given, 'rbf' will be used. If a callable is given it is used to pre-compute the kernel matrix from data matrices; that matrix should be an array of shape `(n_samples, n_samples)`.

degree : int, default=3 Degree of the polynomial kernel function ('poly'). Ignored by all other kernels.

gamma : {'scale', 'auto'} or float, default='scale' Kernel coefficient for 'rbf', 'poly' and 'sigmoid'.

coef0 : float, default=0.0 Independent term in kernel function. It is only significant in 'poly' and 'sigmoid'.

shrinking : bool, default=True Whether to use the shrinking heuristic. See the :ref:`User Guide <shrinking_svm>`.

probability : bool, default=False Whether to enable probability estimates. This must be enabled prior to calling `fit`, will slow down that method as it internally uses 5-fold cross-validation, and `predict_proba` may be inconsistent with `predict`. Read more in the :ref:`User Guide <scores_probabilities>`.

tol : float, default=1e-3 Tolerance for stopping criterion.

cache_size : float, default=200 Specify the size of the kernel cache (in MB).

class_weight : dict or 'balanced', default=None Set the parameter C of class i to class_weight[i]*C for SVC. If not given, all classes are supposed to have weight one. The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as `n_samples / (n_classes * np.bincount(y))`.

verbose : bool, default=False Enable verbose output. Note that this setting takes advantage of a per-process runtime setting in libsvm that, if enabled, may not work properly in a multithreaded context.

max_iter : int, default=-1 Hard limit on iterations within solver, or -1 for no limit.

decision_function_shape : {'ovo', 'ovr'}, default='ovr' Whether to return a one-vs-rest ('ovr') decision function of shape (n_samples, n_classes) as all other classifiers, or the original one-vs-one ('ovo') decision function of libsvm which has shape (n_samples, n_classes * (n_classes - 1) / 2). However, one-vs-one ('ovo') is always used as multi-class strategy. The parameter is ignored for binary classification.

break_ties : bool, default=False If true, `decision_function_shape='ovr'`, and number of classes > 2, :term:`predict` will break ties according to the confidence values of :term:`decision_function`; otherwise the first class among the tied classes is returned. Please note that breaking ties comes at a relatively high computational cost compared to a simple predict.

random_state : int, RandomState instance or None, default=None Controls the pseudo random number generation for shuffling the data for probability estimates. Ignored when `probability` is False. Pass an int for reproducible output across multiple function calls. See :term:`Glossary <random_state>`.

```
# Build SVM model using svm.SVC with default parameters. Default ker
nel is rbl
from sklearn.svm import SVC
sv=SVC() #default kernel is 'rbl'
sv.fit(train_x,train_y)
train_y_pred=sv.predict(train_x)
test_y_pred= sv.predict(test_x)
#
print("Training Accuracy",accuracy_score(train_y,train_y_pred))
print("Testing Accuracy", accuracy_score(test_y,test_y_pred))


# Build SVM model using svm.SVC with Linear Kernel
from sklearn.svm import SVC
sv=SVC(kernel='linear')
sv.fit(train_x,train_y)
train_y_pred=sv.predict(train_x)
test_y_pred= sv.predict(test_x)
#
print("Training Accuracy",accuracy_score(train_y,train_y_pred))
print("Testing Accuracy", accuracy_score(test_y,test_y_pred))

# Build SVM model using svm.SVC with Polynomial kernel
from sklearn.svm import SVC
sv=SVC(kernel='poly')
sv.fit(train_x,train_y)
train_y_pred=sv.predict(train_x)
test_y_pred= sv.predict(test_x)
#
print("Training Accuracy",accuracy_score(train_y,train_y_pred))
print("Testing Accuracy", accuracy_score(test_y,test_y_pred))



# Build SVM model using svm.SVC with Sigmoid kernel
from sklearn.svm import SVC
sv=SVC(kernel='sigmoid')
sv.fit(train_x,train_y)
train_y_pred=sv.predict(train_x)
test_y_pred= sv.predict(test_x)
```

```
#
print("Training
Accuracy",accuracy_score(train_y,train_y_pred))
print("Testing Accuracy",
accuracy_score(test_y,test_y_pred))

# Build SVM model using svm.SVC with Sigmoid kernel
and gamma as auto
from sklearn.svm import SVC
sv=SVC(kernel='sigmoid',gamm
a='auto')
sv.fit(train_x,train_y)
train_y_pred=sv.predict(tra
in_x) test_y_pred=
sv.predict(test_x)
#
print("Training
Accuracy",accuracy_score(train_y,train_y_pred))
print("Testing Accuracy",
accuracy_score(test_y,test_y_pred))
```

RESULT:

Training Accuracy of SVC without Kernel is :
Testing Accuracy of RFC without Kernel is:

Training Accuracy of SVC with linear Kernel is
Testing Accuracy of SCV with linear Kernel is

Training Accuracy of SVC with poly Kernel is:
Testing Accuracy of SVC with Poly Kernel is:

Training Accuracy of SVC with sigmoid Kernel is:
Testing Accuracy of SVC with sigmoid Kernel :

Training Accuracy of SVC with sigmoid Kernel and
gamma auto is:

Testing Accuracy of SVC with sigmoid Kernel and gamma auto is:

Exercise 11:

BUILD RANDOM FOREST MODEL AND APPLY ON GIVEN DATASET. EVALUATE THE MODEL WITH SUITABLE METRICS

Aim: Build Random Forest model and apply on given dataset. Evaluate the model with suitablemetrics.

Software Required: Google Co Lab, Jupyter notebook, Kaggle .

Dataset: https://www.kaggle.com/datasets/uciml/human-activity-recognition-with-smartphones

About Dataset The Human Activity Recognition database was built from the recordings of 30 study participants performing activities of daily living (ADL) while carrying a waist-mounted smartphone with embedded inertial sensors. The objective is to classify activities into one of the six activities performed.

Description of experiment The experiments have been carried out with a group of 30 volunteers within an age bracket of 19-48 years. Each person performed six activities (WALKING, WALKINGUPSTAIRS, WALKINGDOWNSTAIRS, SITTING, STANDING, LAYING) wearing a
smartphone (Samsung Galaxy S II) on the waist. Using its embedded accelerometer and gyroscope, we captured 3-axial linear acceleration and 3-axial angular velocity at a constant rate of 50Hz. The experiments have been video-recorded to label the data manually. The obtained dataset has been randomly partitioned into two sets, where 70% of the volunteers was selected for generating the training data and 30% the test data.

The sensor signals (accelerometer and gyroscope) were pre-processed by applying noise filters and then sampled in fixed-width sliding windows of 2.56 sec and 50% overlap (128 readings/window). The sensor acceleration signal, which has gravitational and body motion components, was separated using a Butterworth low-pass filter into body acceleration and gravity. The gravitational force is assumed to have only low frequency components, therefore a filter with 0.3 Hz cutoff frequency was used. From each window, a vector of features was obtained by calculating variables from the time and frequency domain.

Attribute information
For each record in the dataset the following is provided:
Triaxial acceleration from the accelerometer (total acceleration) and the estimated body acceleration.Triaxial Angular velocity from the gyroscope.
A 561-feature vector with time and frequency domain variables. Its activity label.
An identifier of the subject who carried out the experiment.

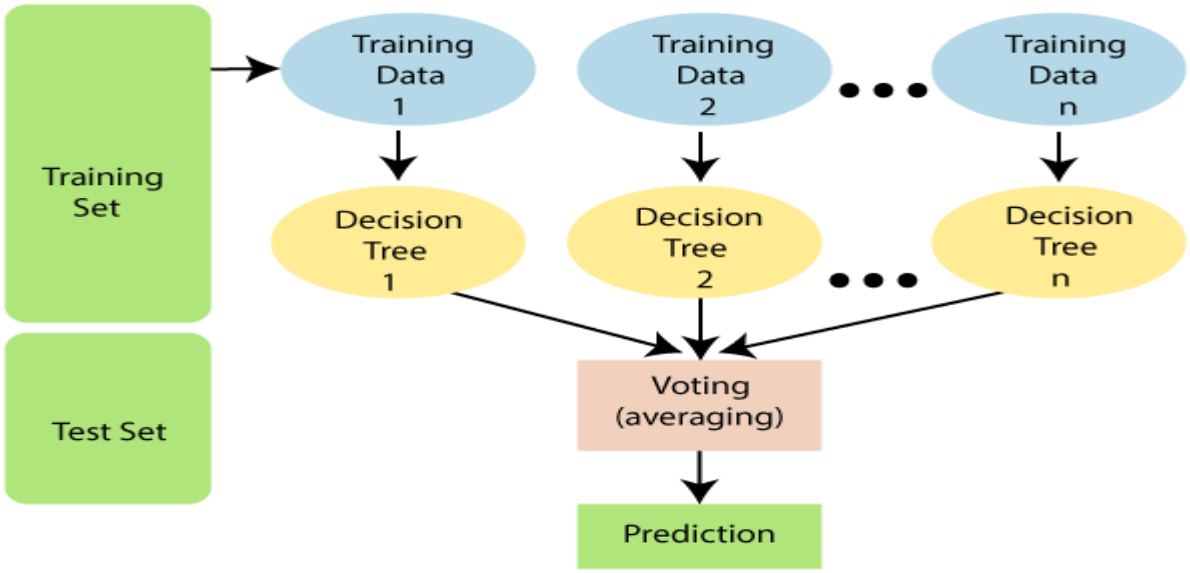
Theory:

Random Forest is a popular machine learning algorithm that belongs to the supervised learning technique. It can be used for both Classification and Regression problems in ML. It is based on the concept of ensemble learning, which is a process of *combining multiple classifiers to solve a complex problem and to improve the performance of the model.*

As the name suggests, "Random Forest is a classifier that contains a number of decision trees on various subsets of the given dataset and takes the average to improve the predictive accuracy of that dataset." Instead of relying on one decision tree, the random forest takes the prediction from each tree and based on the majority votes of predictions, and it predicts the final output.

The greater number of trees in the forest leads to higher accuracy and prevents the problem of overfitting.

The below diagram explains the working of the Random Forest algorithm:



Program :

```
#Import required
basic libraries
import numpy as np
# linear algebra
import pandas as pd # data processing, CSV file I/O
(e.g. pd.read_csv)import os
```

```
# Read dataset into dataframe
data=pd.read_csv('/kaggle/input/human-activity-recognition-with-
smartphones/train.csv')
data.describe()

data

data.info()

data.columns

data.Activity.value_counts()

data.shape

data['activity_code'] = data.Activity.astype('category').cat.codes


data.shape

data.activity_code

data1=data.drop('Activity',axis=1)

data.info()

data1.shape

x_col=data1.columns.to_list()
x_col.pop(-1)
x_data=data1[x_col]
y_col='activity_code'
from sklearn.model_selection import train_test_split
train_x,test_x,train_y,test_y=train_test_split(data1[x_col],data1[y_col].values,test
_size=0.1)

train_x.shape,test_x.shape,train_y.shape,test_y.shape

test_y

# Try to build Logistic regression
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
lg=LogisticRegression()#max_iter=800, solver='sag')
lg.fit(train_x,train_y)
train_y_pred=lg.predict(train_x)
test_y_pred= lg.predict(test_x)
#
print("Training Accuracy",accuracy_score(train_y,train_y_pred))
print("Testing Accuracy", accuracy_score(test_y,test_y_pred))

from sklearn.metrics import confusion_matrix
confusion_matrix(train_y,train_y_pred)

confusion_matrix(test_y,test_y_pred)
```

Random Forest Classifier:

RandomForestClassifier(n_estimators=100, *, criterion="gini", max_depth=None, min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features="auto", max_leaf_nodes=None, min_impurity_decrease=0.0, bootstrap=True, oob_score=False, n_jobs=None, random_state=None, verbose=0, warm_start=False, class_weight=None, ccp_alpha=0.0, max_samples=None)

A random forest classifier.

A random forest is a meta estimator that fits a number of decision tree classifiers on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting. The sub-sample size is controlled with the `max_samples` parameter if `bootstrap=True` (default), otherwise the whole dataset is used to build each tree.

Parameters

n_estimators : int, default=100 The number of trees in the forest.

criterion : {"gini", "entropy"}, default="gini" The function to measure the quality of a split. Supported criteria are "gini" for the Gini impurity and "entropy" for the information gain. Note: this parameter is tree-specific.

max_depth : int, default=None The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.

min_samples_split : int or float, default=2 The minimum number of samples required to split an internal node:

min_samples_leaf : int or float, default=1 The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least `min_samples_leaf` training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression.

min_weight_fraction_leaf : float, default=0.0 The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when sample_weight is not provided.

max_features : {"auto", "sqrt", "log2"}, int or float, default="auto" The number of features to consider when looking for the best split:

max_leaf_nodes : int, default=None Grow trees with `max_leaf_nodes` in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes.

min_impurity_decrease : float, default=0.0 A node will be split if this split induces a decrease of the impurity greater than or equal to this value.

bootstrap : bool, default=True Whether bootstrap samples are used when building trees. If False, the whole dataset is used to build each tree.

oob_score : bool, default=False Whether to use out-of-bag samples to estimate the generalization score. Only available if bootstrap=True.

n_jobs : int, default=None The number of jobs to run in parallel. :meth:`fit`, :meth:`predict`, :meth:`decision_path` and :meth:`apply` are all parallelized over the trees. `None` means 1 unless in a :obj:`joblib.parallel_backend` context. `-1` means using all processors. See :term:`Glossary <n_jobs>` for more details.

random_state : int, RandomState instance or None, default=None Controls both the randomness of the bootstrapping of the samples used when building trees (if `bootstrap=True`) and the

sampling of the features to consider when looking for the best split at each node (if `max_features < n_features`). See :term:`Glossary <random_state>` for details.

verbose : int, default=0 Controls the verbosity when fitting and predicting.

warm_start : bool, default=False When set to `True`, reuse the solution of the previous call to fit and add more estimators to the ensemble, otherwise, just fit a whole new forest. See :term:`the Glossary <warm_start>`.

class_weight : {"balanced", "balanced_subsample"}, dict or list of dicts, default=None Weights associated with classes in the form `{class_label: weight}`. If not given, all classes are supposed to have weight one. For multi-output problems, a list of dicts can be provided in the same order as the columns of y.

ccp_alpha : non-negative float, default=0.0 Complexity parameter used for Minimal Cost-Complexity Pruning. The subtree with the largest cost complexity that is smaller than `ccp_alpha` will be chosen. By default, no pruning is performed.

max_samples : int or float, default=None If bootstrap is True, the number of samples to draw from X to train each base estimator.

Program:

```
# Build Random Forest model using ensemble.RandomForestClassifier
from sklearn.ensemble import RandomForestClassifier
rf=RandomForestClassifier()
rf.fit(train_x,train_y)
train_y_pred=rf.predict(train_x)
test_y_pred= rf.predict(test_x)
#
print("Training Accuracy",accuracy_score(train_y,train_y_pred))
print("Testing Accuracy", accuracy_score(test_y,test_y_pred))


# Build Random Forest model using ensemble.RandomForestClassifier
from sklearn.ensemble import RandomForestClassifier
rf=RandomForestClassifier(criterion='entropy')
rf.fit(train_x,train_y)
train_y_pred=rf.predict(train_x)
test_y_pred= rf.predict(test_x)
#
print("Training Accuracy",accuracy_score(train_y,train_y_pred))
print("Testing Accuracy", accuracy_score(test_y,test_y_pred))

# Build Random Forest model using ensemble.RandomForestClassifier
from sklearn.ensemble import RandomForestClassifier
rf=RandomForestClassifier(max_depth=5)
rf.fit(train_x,train_y)
train_y_pred=rf.predict(train_x)
test_y_pred= rf.predict(test_x)
#
print("Training Accuracy",accuracy_score(train_y,train_y_pred))
print("Testing Accuracy", accuracy_score(test_y,test_y_pred))

# Build Random Forest model using ensemble.RandomForestClassifier
from sklearn.ensemble import RandomForestClassifier
rf=RandomForestClassifier(max_depth=10)
```

```
rf.fit(train_x,train_y)


train_y_pred=rf.predict(train_x)
test_y_pred= rf.predict(test_x)
#
print("Training Accuracy",accuracy_score(train_y,train_y_pred))
print("Testing Accuracy", accuracy_score(test_y,test_y_pred))
# Build Random Forest model using ensemble.RandomForestClassifier
from sklearn.ensemble import RandomForestClassifier
rf=RandomForestClassifier(max_features='sqrt')
rf.fit(train_x,train_y)
train_y_pred=rf.predict(train_x)
test_y_pred= rf.predict(test_x)
#
print("Training Accuracy",accuracy_score(train_y,train_y_pred))
print("Testing Accuracy", accuracy_score(test_y,test_y_pred))
```

RESULT:

The training and Testing accuracy are:

Training Accuracy of RFC without entropy criteria is
Testing Accuracy of RFC without entropy criteria is

Training Accuracy of RFC with entropy criteria is
Testing Accuracy of RFC with entropy criteria is

Training Accuracy of RFC with max depth of 5 is
Testing Accuracy of RFC with max depth of 5 is

Training Accuracy of RFC with max depth of 10 is
Testing Accuracy of RFC with max depth of 10 is

Training Accuracy of RFC with max feature of sqrt is
Testing Accuracy of RFC with max feature of sqrt is