**EXPERIMENT – 1**

**AIM:** To build and evaluate models using Linear Regression with metrics.

**DESCRIPTION:** Linear regression is one of the easiest and most popular Machine Learning algorithms. It is a statistical method that is used for predictive analysis. Linear regression algorithm shows a linear relationship between a dependent (y) and one or more independent (y) variables, hence called as linear regression. Since linear regression shows the linear relationship, which means it finds how the value of the dependent variable is changing according to the value of the independent variable.

Mathematically, we can represent a linear regression as:

y= a0+a1x+ ε

Y= Dependent Variable (Target Variable)

X= Independent Variable (predictor Variable)

a0= intercept of the line (Gives an additional degree of freedom)

a1 = Linear regression coefficient (scale factor to each input value).

ε = random error

The values for x and y variables are training datasets for Linear Regression model representation.

Types of Linear Regression

Linear regression can be further divided into two types of the algorithm:

• **Simple Linear Regression:** If a single independent variable is used to predict the value of a numerical dependent variable, then such a Linear Regression algorithm is called Simple Linear Regression.

• **Multiple Linear regression:** If more than one independent variable is used to predict the value of a numerical dependent variable, then such a Linear Regression algorithm is called Multiple Linear Regression.

Evaluation metrics for a linear regression model

Evaluation metrics are a measure of how good a model performs and how well it approximates the relationship. There are three error metrics that are commonly used for evaluating and reporting the performance of a regression model; they are:

• Mean Squared Error (MSE).

• Root Mean Squared Error (RMSE).

• Mean Absolute Error (MAE)

Mean Squared Error (MSE) It is the average of the squared difference between the predicted and actual value. Mean Absolute Error (MAE) This is simply the average of the absolute difference between the target value and the value predicted by the model. Root Mean Squared Error (RMSE) This is the square root of the average of the squared difference of the predicted and actual value.

**6 Steps to build a Linear Regression model**

Step 1: Importing the dataset

Step 2: Data pre-processing

Step 3: Splitting the test and train sets

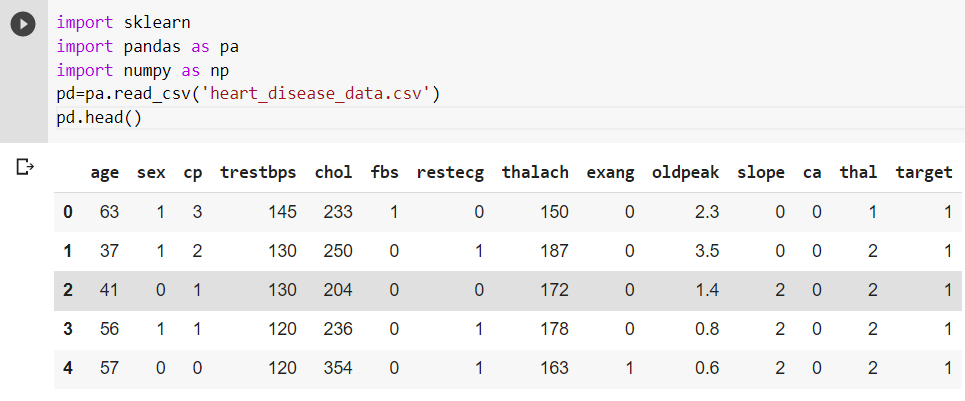
Step 4: Fitting the linear regression model to the training set

Step 5: Predicting test results

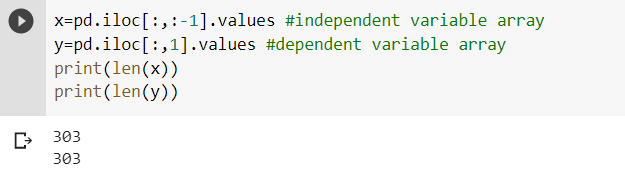
Step 6: Visualizing the test results

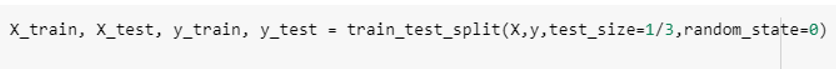
**PROGRAM :**

1. Load the dataset :



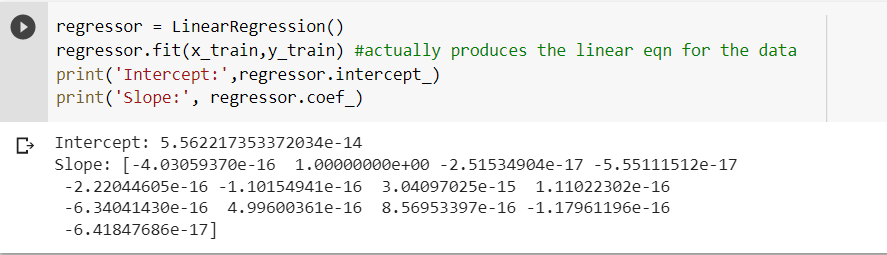
1. Linear Regression :

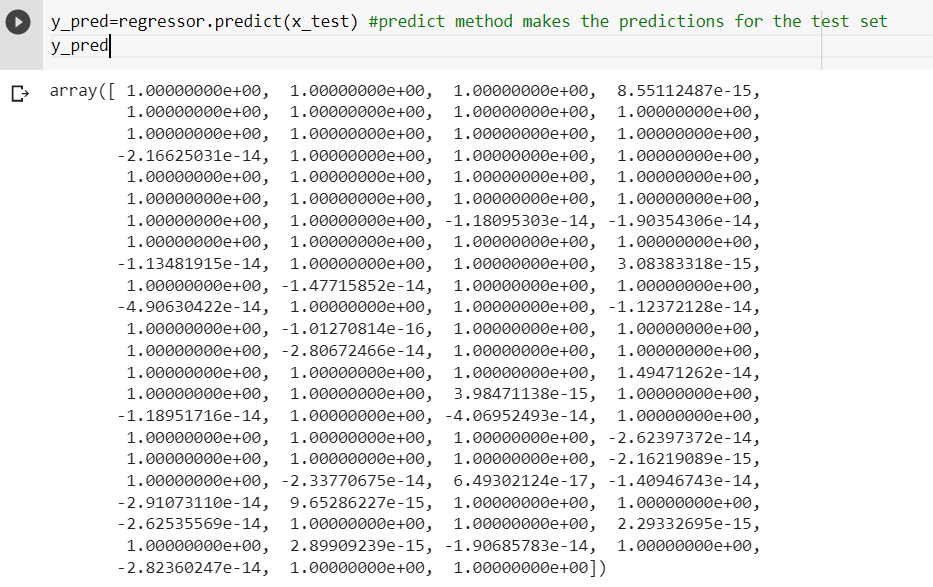


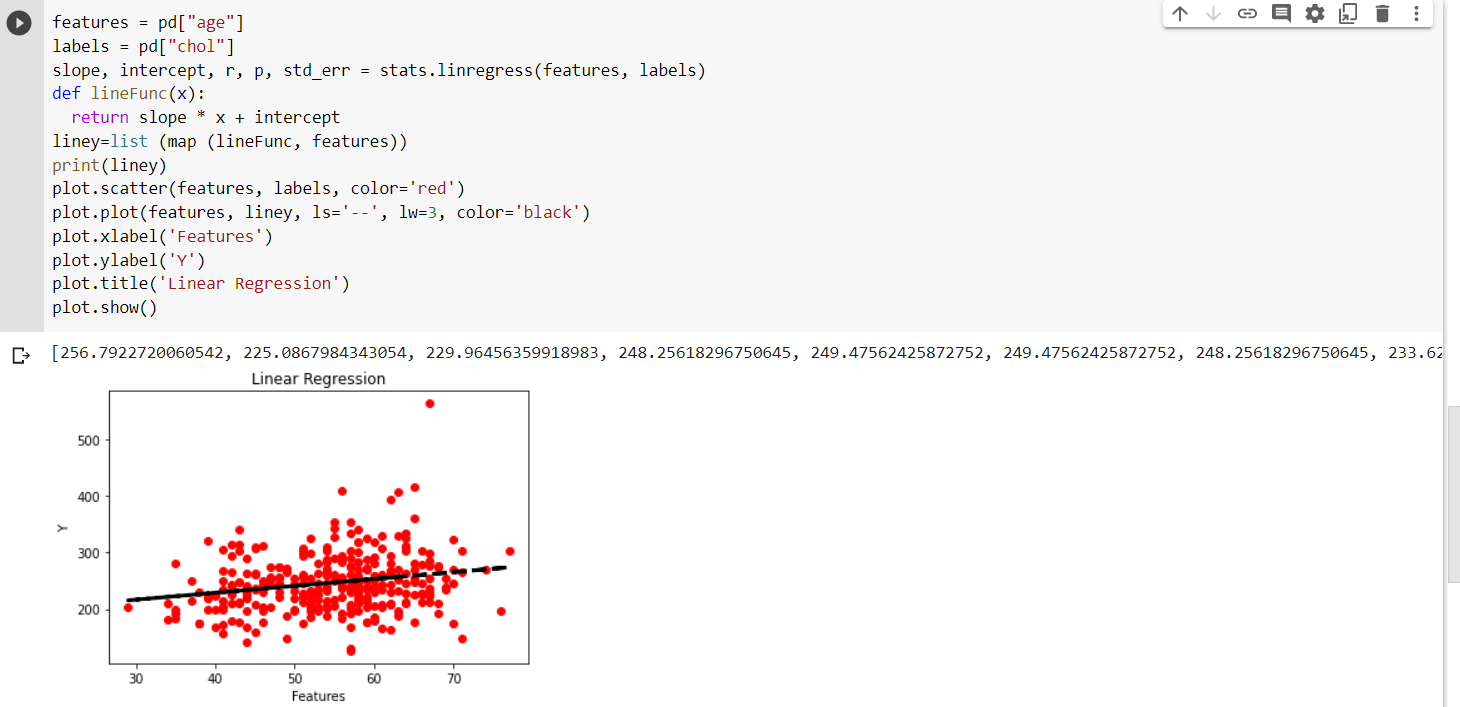


Fitting linear regression model into the training set

To fit the regressor into the training set, we will call the fit method – function to fit the regressor into the training set. We need to fit X\_train (training data of matrix of features) into the target values y\_train. Thus the model learns the correlation and learns how to predict the dependent variables based on the independent variable.

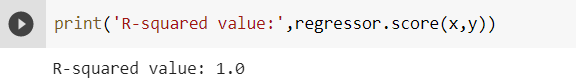






**Evaluation Metrics**

To evaluate the overall fit of a linear model, we use the R-squared value R-squared is the proportion of To evaluate the overall fit of a linear model, we use the R-squared value R-squared is the proportion of variance explained It is the proportion of variance in the observed data that is explained by the model, or the reduction in error over the null model The null model just predicts the mean of the observed response, and thus it has an intercept and no slope R-squared is between 0 and 1 Higher values are better because it means that more variance is explained by the model



**CONCLUSION:**

Therefore, we have build a Linear regression for the heart disease dataset.

**EXPERIMENT – 2**

**AIM:** To implement different plotings for Data Visualization.

**DESCRIPTION:** In today’s world, a lot of data is being generated on a daily basis. And sometimes to analyze this data for certain trends, patterns may become difficult if the data is in its raw format. To overcome this data visualization comes into play. Data visualization provides a good, organized pictorial representation of the data which makes it easier to understand, observe, analyze

**PROGRAM:**

**Dataset used:** name\_gender\_dataset

import matplotlib.pyplot as plt

import numpy

import pandas

#On one axis number from 1 to 15

a = pandas.read\_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/00591/name\_gender\_dataset.csv')

#On other axis generate random integers with mean and sigma

mean = 50

sigma = 10

b = numpy.random.normal(mean, sigma, 15).astype(int)

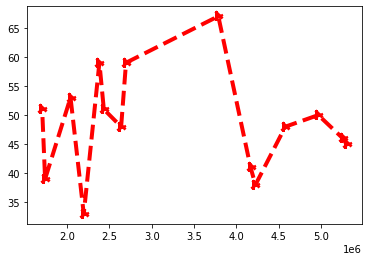
z1=a['Count'][0:15]

z2=a['Probability'][0:15]

z3=a['Gender'][0:15]

plt.plot(z1,b,color='Red',ls='--',lw=4,marker='3',mew=10)

**OUTPUT:**

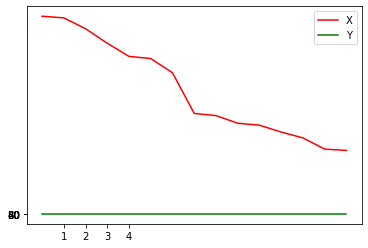
****

sales = pandas.DataFrame({'X':z1,'Y':z2,'Z': z3})

colors = ['Red','Green','Black']

sales.plot(xticks=range(1,5),yticks=range(0,100,20),color=colors)

**OUTPUT:**

****

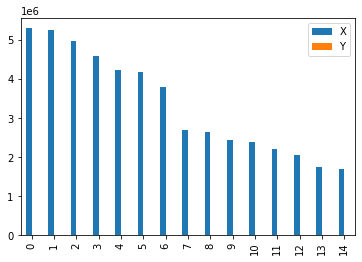
plt.bar(z2,z3)

**OUTPUT:**

****

sales.plot(kind='bar')

**OUTPUT:**

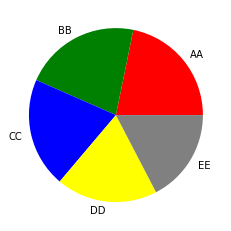
****

z11=a['Count'][0:5]

color\_list = ['Red','Green','Blue','Yellow','Grey']

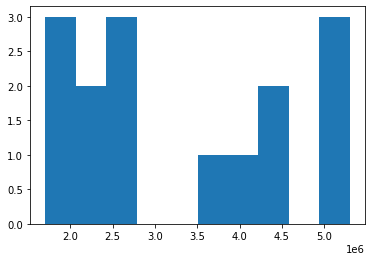
plt.pie(z11,labels=['AA','BB','CC','DD','EE'],colors=color\_list)

**OUTPUT:**

****

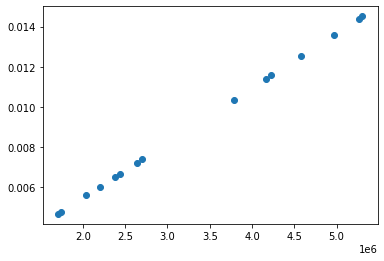
plt.hist(z1)

**OUTPUT:**

****

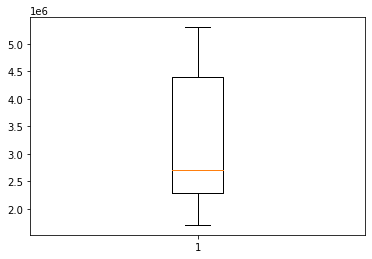
plt.scatter(z1,z2)

**OUTPUT:**

****

plt.boxplot(z1)

**OUTPUT:**

****

mean = 20

sigma = 5

c = numpy.random.normal(mean, sigma, 15).astype(int)

#Create figure object

fig\_sub\_object = plt.figure()

#Two axes inside figure object.

number\_of\_rows= 1

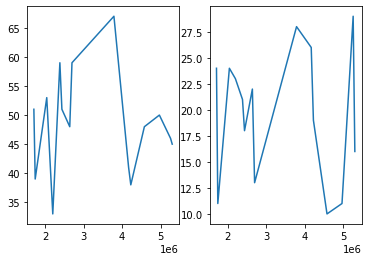
number\_of\_cols = 2

fig\_sub\_object, (axes1,axes2) = plt.subplots(number\_of\_rows,number\_of\_cols)

axes1.plot(z1,b)

axes2.plot(z1,c)

**OUTPUT:**

****

import seaborn

from os import name

#Generate random integers with mean and sigma

mean = 25

sigma = 10

dist\_data\_1 = numpy.random.normal(mean, sigma, 500).astype(int)

dist\_data\_2 = numpy.random.normal(mean+5, sigma-4, 500).astype(int)

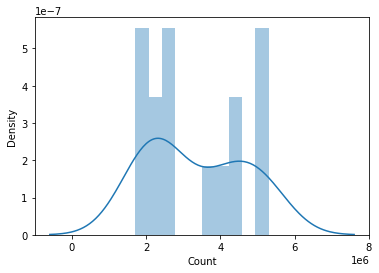
dist\_data\_3 = numpy.random.normal(mean-5, sigma+2, 500).astype(int)

dist\_data = pandas.DataFrame({"A" :dist\_data\_1,"B":dist\_data\_2,"C":dist\_data\_3})

z4=a['Name'][0:15]

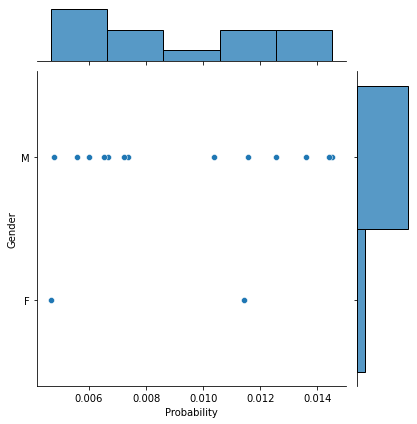
seaborn.distplot(z1,bins=10)

**OUTPUT:**



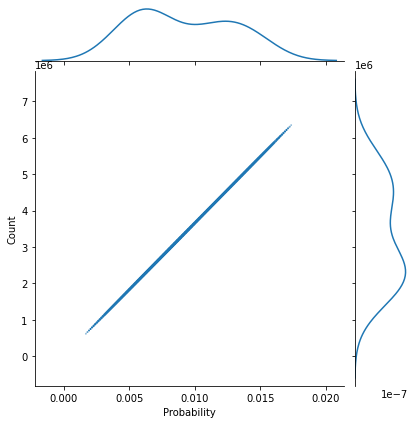
seaborn.jointplot(x=z2, y=z3);

**OUTPUT:**



seaborn.jointplot(x=z2, y=z1,kind="kde")

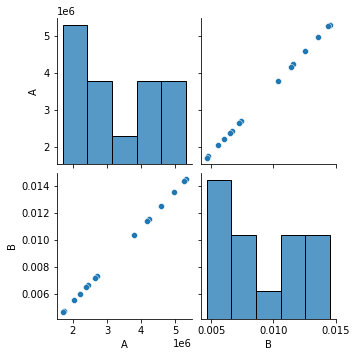
**OUTPUT:**

****

dist\_data = pandas.DataFrame({"A" :z1,"B":z2,"C":z3})

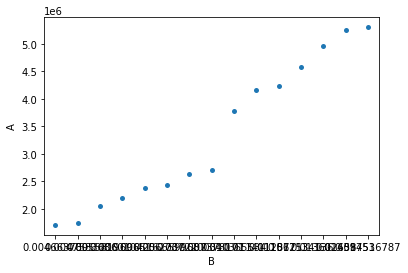
seaborn.pairplot(dist\_data)

**OUTPUT:**



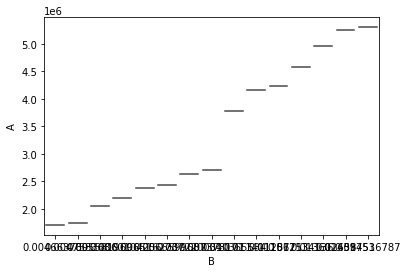
seaborn.stripplot(x="B", y="A", data=dist\_data, jitter=True)

**OUTPUT:**



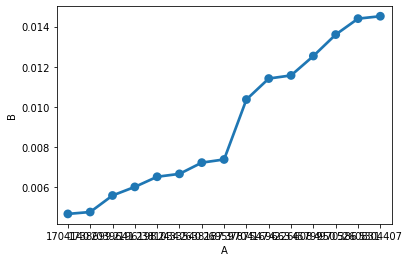
seaborn.violinplot(x="B", y="A", data=dist\_data)

**OUTPUT:**

****

seaborn.pointplot(x="A", y="B",data=dist\_data)

**OUTPUT:**

****

**CONCLUSION:**

We have successfully implemented different plotings for Data Visualization

**EXPERIMENT – 3**

**AIM:** To implement Ridge and lasso regression techniques

**DESCRIPTION:**

Ridge Regression and Lasso Regression are two popular regularization techniques used in linear regression to prevent overfitting and improve model generalization.

Ridge Regression adds a penalty term to the cost function, which is the sum of squared differences between the actual and predicted values of the target variable, along with the sum of squared coefficients multiplied by a hyperparameter called alpha. The penalty term shrinks the coefficients towards zero, thereby reducing their impact on the predictions. The higher the value of alpha, the stronger the penalty and the more the coefficients are shrunk towards zero.

Lasso Regression, on the other hand, adds a penalty term that is the sum of absolute values of the coefficients multiplied by alpha. Lasso Regression has a tendency to shrink some of the coefficients all the way to zero, thereby effectively performing feature selection. The higher the value of alpha, the more the coefficients are shrunk towards zero, and the more features are eliminated.

**ALGORITHM:**

1)Import the dataset

2)Perform pre-processing

3)Splitting the data into training and testing datasets

4)Apply ridge and lasso regression on the datasets

5)Perform the evaluation metrics and analyse the output

**PROGRAM:**

from sklearn.linear\_model import Ridge, Lasso

from sklearn.datasets import load\_iris

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import mean\_squared\_error

import numpy as np

import sklearn

import pandas as pa

import numpy as np

pd=pa.read\_csv('heart\_disease\_data.csv')

pd.head()

X = pd.drop('age',axis=1)

y = pd['age']

X\_train, X\_test, y\_train, y\_test = train\_test\_split (X, y, test\_size=0.30)

# Ridge Regression

import matplotlib.pyplot as plot

from sklearn.linear\_model import Ridge

## training the model

ridgeReg = Ridge(alpha=0.05)

ridgeReg.fit(x\_train,y\_train)

pred = ridgeReg.predict(x\_test)

#calculating mse

lreg = LinearRegression()

lreg.fit(x\_train,y\_train)

pred\_cv = lreg.predict(x\_test)

mse = np.mean((pred\_cv - y\_test)\*\*2)

#mse 1348171.96 ## calculating score

ridgeReg.score(x\_test,y\_test) #0.5691

plot.plot(pred\_cv,pred)

plot.xlabel('variables')

plot.ylabel('coefficients')

plot.show()

# Lasso Regression

import matplotlib.pyplot as plot

lassoReg = Lasso(alpha=0.3)

lassoReg.fit(x\_train,y\_train)

pred = lassoReg.predict(x\_test)

# calculating mse

lreg = LinearRegression()

lreg.fit(x\_train,y\_train)

pred\_cv = lreg.predict(x\_test)

mse = np.mean((pred\_cv - y\_test)\*\*2)

# mse

# 1346205.82

lassoReg.score(x\_test,y\_test)

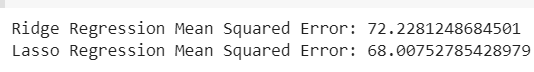
# 0.5720

print()

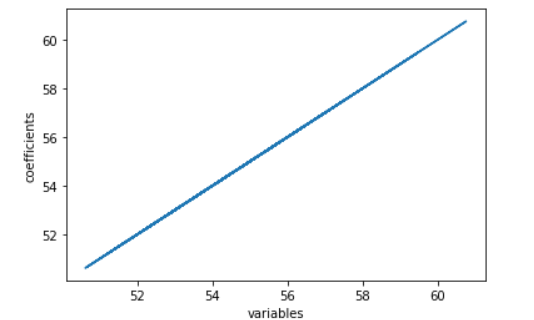
plot.plot(pred\_cv,pred)

plot.show()

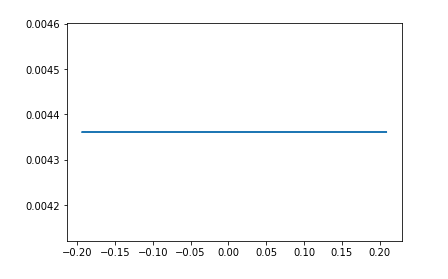
**OUTPUT:**



**Plot of Ridge Regression :**



**Plot of Lasso Regression :**



**CONCLUSION:**

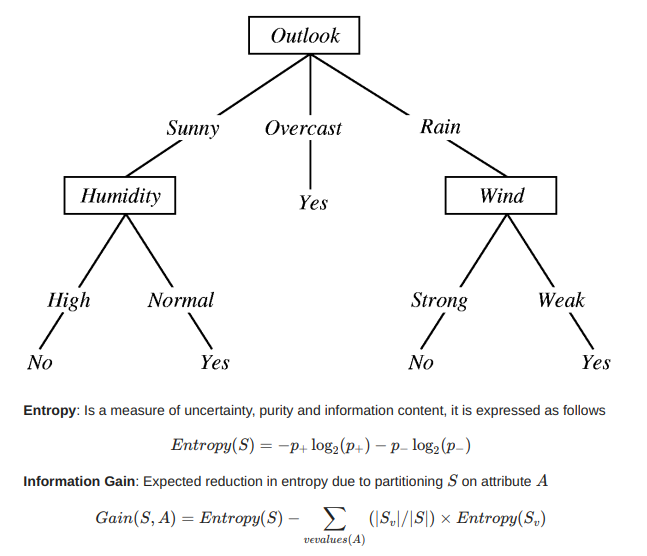
Therefore Ridge and lasso regressions techniques are successfully demonstrated.

**EXPERIMENT – 4**

**AIM:** Program to implement Decision Tree Classifier on iris dataset

**Description :**

A decision tree is a flowchart-like structure in which each internal node represents a test on a feature (e.g. whether a coin flip comes up heads or tails) , each leaf node represents a class label (decision taken after computing all features) and branches represent conjunctions of features that lead to those class labels. The paths from root to leaf represent classification rules. Below diagram illustrate the basic flow of decision tree for decision making with labels (Rain(Yes), No Rain(No)). Example Decision Tree for Outlook, Humidity and Wind Dataset for playing an Outdoor game



**PROGRAM :**

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.tree import DecisionTreeClassifier

from sklearn.metrics import classification\_report,confusion\_matrix

import sklearn.metrics as metrics

from IPython.display import Image

from six import StringIO

from sklearn.tree import export\_graphviz

import pydot

df = pd.read\_csv('heart\_disease\_data.csv')

df.shape

df.head()

X = df.drop('chol',axis=1)

y = df['chol']

print(X)

print(y)

X\_train, X\_test, y\_train, y\_test = train\_test\_split (X, y, test\_size=0.30)

dtree = DecisionTreeClassifier()

dtree.fit(X\_train,y\_train)

y\_pred = dtree.predict(X\_test)

accuracy = metrics.accuracy\_score(y\_test,y\_pred)

print('Accuracy Score:',accuracy )

mat = confusion\_matrix(y\_test,y\_pred)

rep = classification\_report(y\_test,y\_pred)

print(mat)

print(rep)

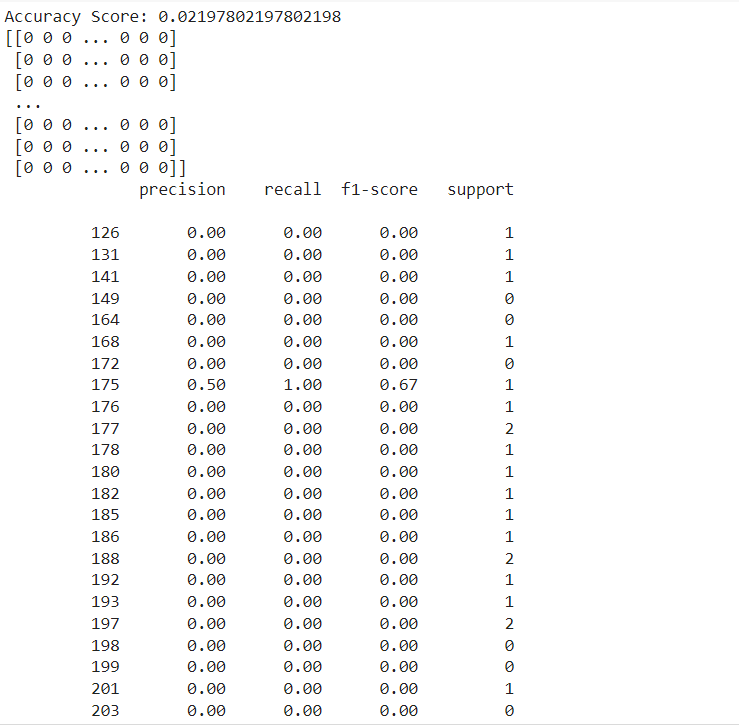
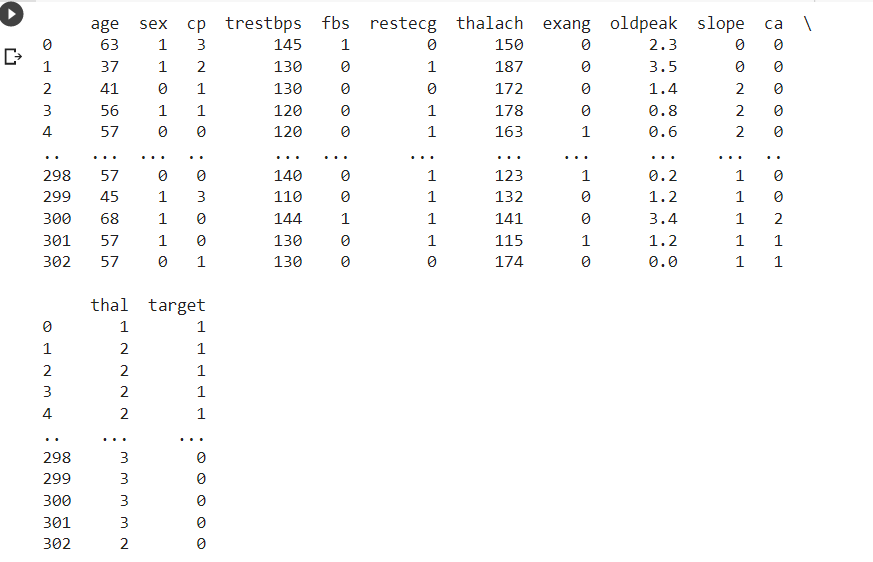
features = list(df.columns[1:])

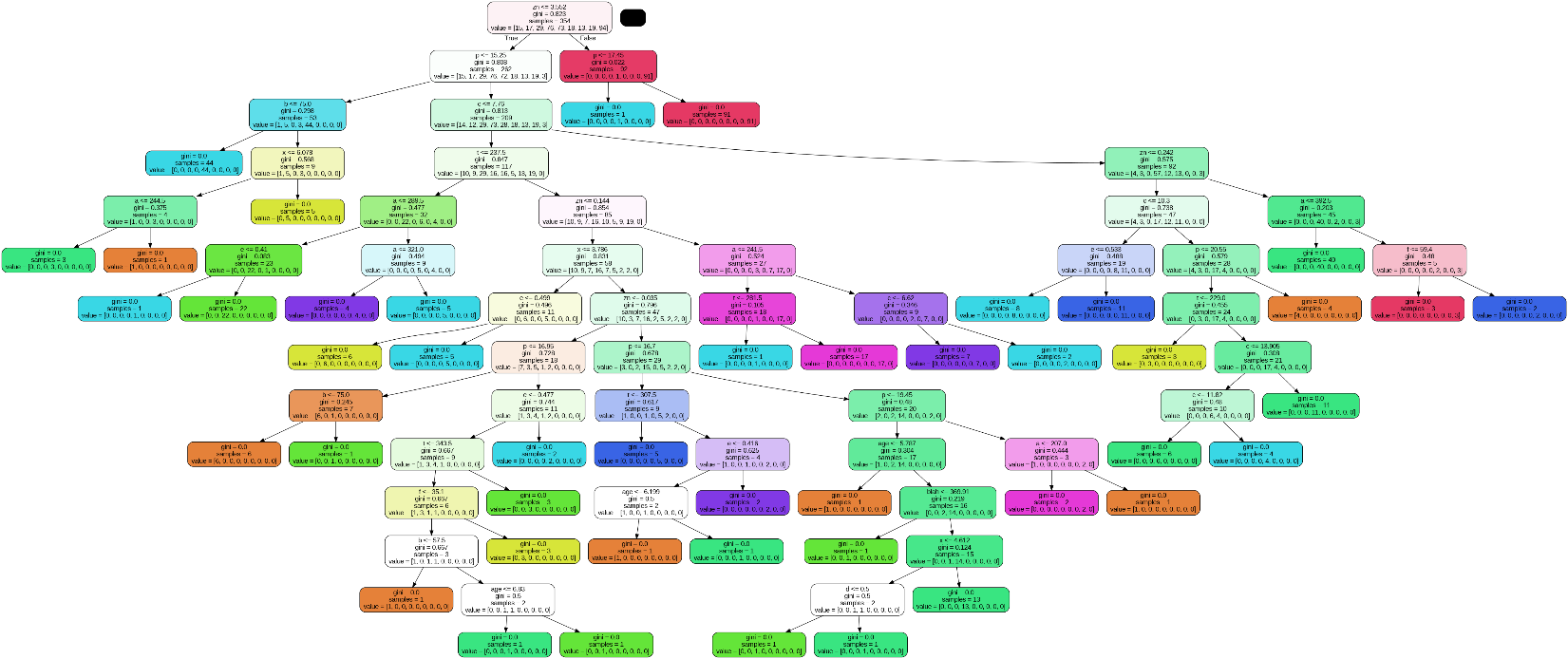
dot\_data = StringIO()

export\_graphviz(dtree, out\_file=dot\_data,feature\_names=features,filled=True,rounded=True)

graph = pydot.graph\_from\_dot\_data(dot\_data.getvalue())

Image(graph[0].create\_png())

**OUTPUT:**

**CONCLUSION :**

Hence, The Decision Tree Classifier has been applied for iris dataset and different matrices has been computed successfully.

**EXPERIMENT - 5**

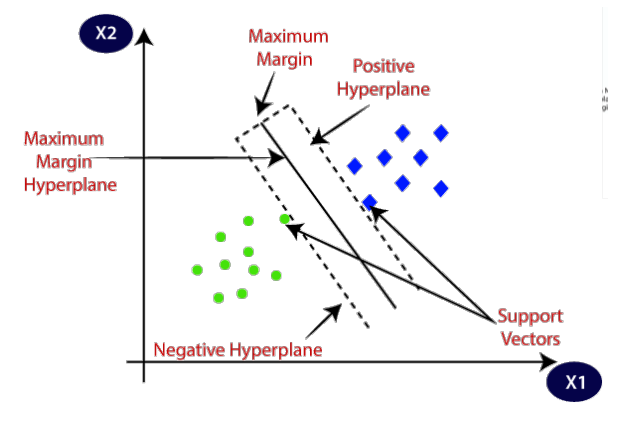
**AIM:** Program to implement Support Vector Machine (SVM) Classifier on iris dataset

**Description :**

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.



**PROGRAM:**

1. **SVM WITH KERNAL=’’LINEAR”**

from sklearn.model\_selection import train\_test\_split

from sklearn import svm

from sklearn import metrics

import numpy as np

import matplotlib.pyplot  as plot

import pandas as pa

pd = pa.read\_csv("heart\_disease\_data.csv")

X=pd.drop(['chol'], axis = 'columns')

y=pd['chol']

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size = 0.3)

X\_train.shape, X\_test.shape, y\_train.shape, y\_test.shape

clf = svm.SVC(kernel='linear')

clf.fit(X\_train, y\_train)

y\_pred = clf.predict(X\_test)

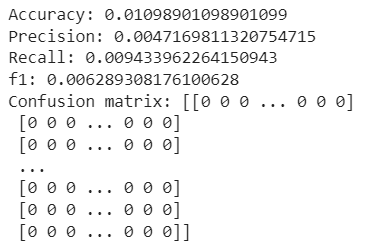
print("Accuracy:",metrics.accuracy\_score(y\_test, y\_pred))

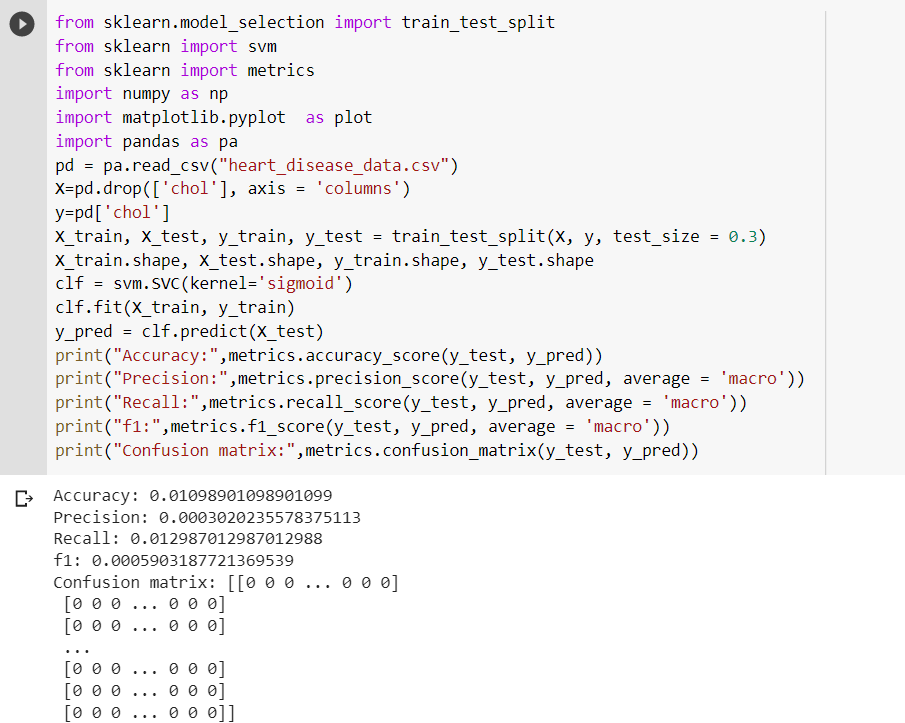
print("Precision:",metrics.precision\_score(y\_test, y\_pred, average = 'macro'))

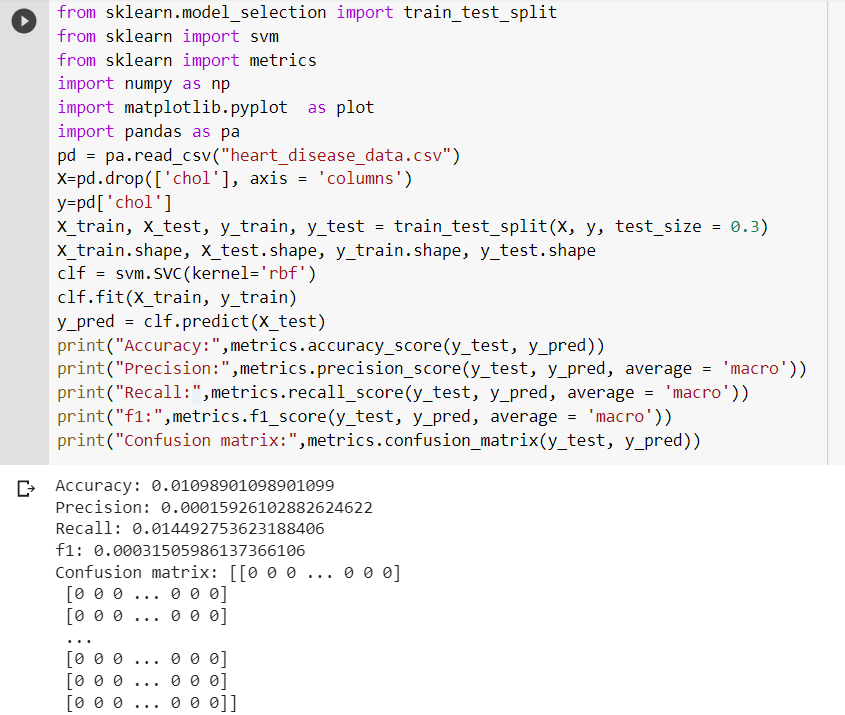
print("Recall:",metrics.recall\_score(y\_test, y\_pred, average = 'macro'))

print("f1:",metrics.f1\_score(y\_test, y\_pred, average = 'macro'))

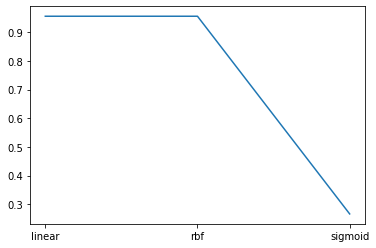
print("Confusion matrix:",metrics.confusion\_matrix(y\_test, y\_pred))



1. **SVM WITH KERNAL=’’SIGMOID’’**
2. **SVM WITH KERNAL=’’RBF”**



**GRAPH:**



**CONCLUSION :**

Hence, The SVM Classifier has been applied for iris dataset and different matrices has been computed successfully.

**EXPERIMENT - 6**

**AIM:** Program to implement K means Clustering

**Description:** K-means clustering is a popular unsupervised machine learning algorithm used for partitioning data into clusters based on their similarity. The algorithm aims to group data points that are similar to each other while maximizing the dissimilarity between different clusters.

K-means clustering is an iterative algorithm that aims to minimize the within-cluster sum of squares, also known as inertia or distortion. The algorithm is efficient and can handle large datasets. However, the choice of the number of clusters (K) is crucial and can impact the clustering results. Different initialization methods, such as the k-means++ algorithm, can be used to improve the convergence and avoid poor local optima.

K-means clustering has various applications, including customer segmentation, image segmentation, document clustering, anomaly detection, and data compression. It is a widely used technique for exploratory data analysis and pattern recognition, providing insights into the underlying structure of the data.

**CODE:**

from sklearn import datasets

import matplotlib.pyplot  as plot

from sklearn.cluster import KMeans

from sklearn.model\_selection import train\_test\_split

from sklearn import metrics

import numpy as np

import matplotlib.pyplot  as plot

import pandas as pa

pd = datasets.make\_blobs()

X, y = make\_blobs(n\_samples=300,centers=4,cluster\_std=0.60,random\_state=0)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size = 0.3)

X\_train.shape, X\_test.shape, y\_train.shape, y\_test.shape

clf = KMeans(n\_clusters=3)

clf.fit(X\_train, y\_train)

y\_pred = clf.fit\_predict(X\_test)

print("Accuracy:",metrics.accuracy\_score(y\_test, y\_pred))

print("Precision:",metrics.precision\_score(y\_test, y\_pred, average = 'macro'))

print("Recall:",metrics.recall\_score(y\_test, y\_pred, average = 'macro'))

print("f1:",metrics.f1\_score(y\_test, y\_pred, average = 'macro'))

print("Confusion matrix:",metrics.confusion\_matrix(y\_test, y\_pred))

plot.scatter(X[:, 0], X[:, 1], s=50)

from sklearn.cluster import KMeans

kmeans = KMeans(n\_clusters=4)

kmeans.fit(X)

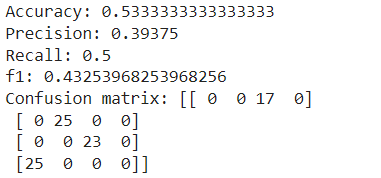
y\_kmeans = kmeans.predict(X)

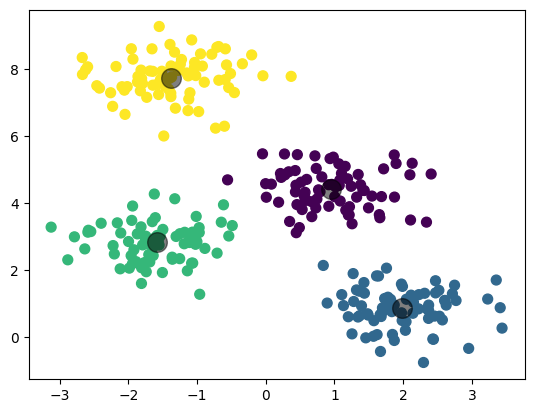
plot.scatter(X[:, 0], X[:, 1], c=y\_kmeans, s=50, cmap='viridis')

centers = kmeans.cluster\_centers\_

plot.scatter(centers[:, 0], centers[:, 1], c='black', s=200, alpha=0.5);

**OUTPUT:**





**CONCLUSION:**

We have successfully implemented K means Clustering.

**EXPERIMENT - 7**

**AIM:** Program to implement DBSCAN Clustering

**Description**: DBSCAN (Density-Based Spatial Clustering of Applications with Noise) is a density-based clustering algorithm commonly used in unsupervised machine learning. It is designed to discover clusters of arbitrary shape within a dataset, while also being able to identify noise points. Unlike some other clustering algorithms, DBSCAN does not require the number of clusters to be predefined.

DBSCAN has a few key advantages. It can discover clusters of various shapes and sizes, including clusters with irregular boundaries. It can handle outliers effectively by classifying them as noise points. DBSCAN also does not require the number of clusters to be specified in advance, making it more flexible than algorithms like K-means.

However, DBSCAN has some considerations. The algorithm's performance can be affected by the choice of parameters, such as the radius and minimum number of points required to form a cluster. Determining suitable values for these parameters can be challenging and may require experimentation or domain knowledge. The algorithm may also struggle with datasets where the density of points varies significantly or when dealing with high-dimensional data.

DBSCAN is widely used in various applications, including image segmentation, spatial data analysis, anomaly detection, and outlier detection. It provides a powerful approach to cluster analysis, especially when dealing with datasets that exhibit varying densities and complex structures.

**CODE:**

from sklearn import datasets

import matplotlib.pyplot  as plot

from sklearn.cluster import DBSCAN

from sklearn.model\_selection import train\_test\_split

from sklearn import metrics

import numpy as np

import matplotlib.pyplot  as plot

import pandas as pa

pd = datasets.make\_blobs()

X, y = make\_blobs(n\_samples=12000,centers=4,cluster\_std=0.50,random\_state=0)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size = 0.3)

X\_train.shape, X\_test.shape, y\_train.shape, y\_test.shape

clf = DBSCAN(eps=0.1)

clf.fit(X\_train, y\_train)

y\_pred = clf.fit\_predict(X\_test)

print("Accuracy:",metrics.accuracy\_score(y\_test, y\_pred))

print("Precision:",metrics.precision\_score(y\_test, y\_pred, average = 'macro'))

print("Recall:",metrics.recall\_score(y\_test, y\_pred, average = 'macro'))

print("f1:",metrics.f1\_score(y\_test, y\_pred, average = 'macro'))

print("Confusion matrix:",metrics.confusion\_matrix(y\_test, y\_pred))

plot.scatter(X[:, 0], X[:, 1], s=25)

dbscan = DBSCAN(eps= 0.5, min\_samples = 4).fit(X)

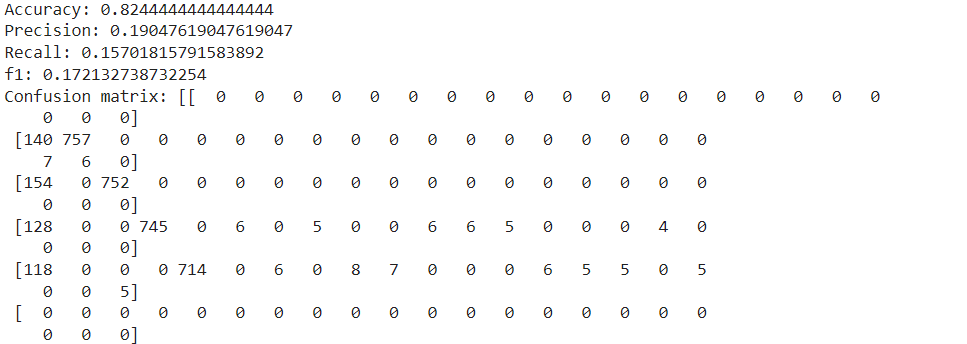
dbscan.fit(X)

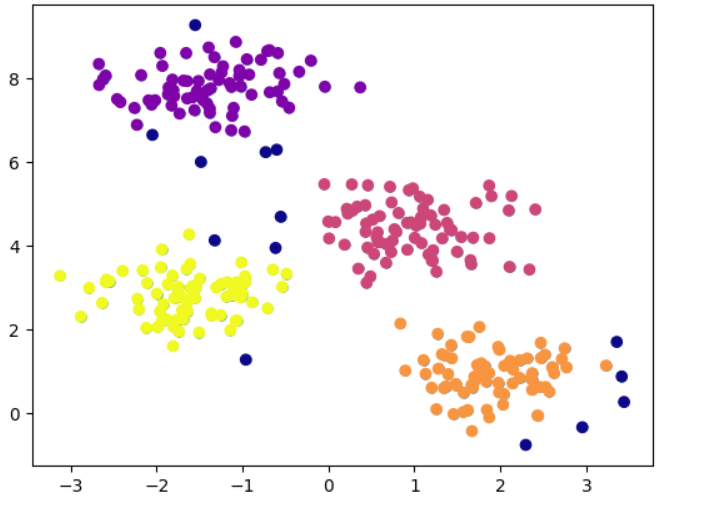
y\_scan = dbscan.fit\_predict(X)

centers = dbscan.labels\_

plot.scatter(X[:, 0], X[:,1], c = centers, cmap= "plasma");

**OUTPUT:**





**CONCLUSION:**

We have successfully implemented DBSCAN Clustering.

**EXPERIMENT - 8**

**AIM:** Program to implement Logistic Regression

**Description**: Logistic regression is a popular statistical modeling technique used for binary classification problems. It is widely employed in machine learning and statistics to predict the probability of an event or the likelihood of a certain outcome based on a set of independent variables.

Logistic regression has several advantages. It is relatively simple to interpret, as the coefficients provide insights into the relationship between the independent variables and the log-odds of the event occurring. It can handle both continuous and categorical independent variables. Logistic regression also allows for probabilistic predictions, making it useful for risk assessment and decision-making.

However, logistic regression also has certain assumptions and limitations. It assumes a linear relationship between the independent variables and the log-odds of the event. It assumes independence of observations and that the error terms are independent and identically distributed. Logistic regression may struggle with non-linear relationships and may not perform well with highly imbalanced datasets or when there are multicollinearity issues.

Logistic regression is widely applied in various domains, including healthcare, marketing, finance, and social sciences, for tasks such as predicting disease outcomes, customer churn, credit risk, and sentiment analysis. It serves as a fundamental algorithm in the field of binary classification and provides valuable insights into the relationships between variables and binary outcomes.

**CODE:**

from sklearn.linear\_model import LogisticRegression

from sklearn.model\_selection import train\_test\_split

pd = datasets.load\_iris()

X, y = make\_blobs(n\_samples=300,centers=4,cluster\_std=0.60,random\_state=0)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

logreg = LogisticRegression()

logreg.fit(X\_train, y\_train)

y\_pred = logreg.predict(X\_test)

acc = metrics.accuracy\_score(y\_test, y\_pred)

pre = metrics.precision\_score(y\_test, y\_pred,average='macro')

r = metrics.recall\_score(y\_test, y\_pred,average='macro')

f1 =  metrics.f1\_score(y\_test, y\_pred,average='macro')

from sklearn import metrics

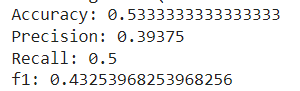
print("Accuracy:",acc )

print("Precision(in %):", pre)

print("Recall(in %):", r)

print("F1 score(in %):",f1)

**OUTPUT:**



**CONCLUSION:**

We have successfully implemented Logistic Regression.

**EXPERIMENT - 9**

**AIM:** Program to implement Naïve Bayes

**Description**:

Naive Bayes is a popular machine learning algorithm that is commonly used for classification tasks. It is based on the principles of Bayes' theorem and assumes that the features (independent variables) are conditionally independent of each other given the class labels.

Here's Naive Bayes works:

1. Bayes' theorem: Naive Bayes is built upon Bayes' theorem, which describes the relationship between the conditional probability of an event given prior knowledge and the prior probability of the event. The theorem is expressed as:

P(Y|X) = (P(X|Y) \* P(Y)) / P(X)

where:

- P(Y|X) is the posterior probability of class Y given the features X.

- P(X|Y) is the likelihood of the features X given the class Y.

- P(Y) is the prior probability of class Y.

- P(X) is the probability of the features X.

2. Assumption of conditional independence: Naive Bayes assumes that the features are conditionally independent of each other given the class label. This simplifying assumption allows the algorithm to calculate the likelihood of each feature independently.

3. Model training: In the training phase, Naive Bayes estimates the prior probabilities of each class label by counting the occurrences of each class in the training dataset. It also estimates the likelihood probabilities of the features given each class label. Depending on the type of features, different variants of Naive Bayes, such as Gaussian Naive Bayes, Bernoulli Naive Bayes, or Multinomial Naive Bayes, can be used.

4. Model testing and prediction: Once the model is trained, it can be used to predict the class labels of new, unseen instances. Naive Bayes calculates the posterior probability of each class given the features of the instance using Bayes' theorem. The class with the highest posterior probability is predicted as the output class.

5. Evaluation and performance: The performance of the Naive Bayes model can be evaluated using various metrics such as accuracy, precision, recall, and F1 score. It is important to note that Naive Bayes assumes the independence of features, which may not hold true in some real-world datasets. However, despite this simplifying assumption, Naive Bayes often performs well in practice, especially when the independence assumption is approximately satisfied or when dealing with high-dimensional datasets.

Naive Bayes is known for its simplicity, speed, and ability to handle large feature spaces. It is widely used in text classification, spam filtering, sentiment analysis, recommendation systems, and various other applications where fast and efficient classification is required. Although it may not always be the most accurate algorithm, Naive Bayes can provide reliable results in many practical scenarios.

**CODE:**

from sklearn.model\_selection import train\_test\_split

from sklearn import svm

from sklearn import metrics

import numpy as np

import matplotlib.pyplot  as plot

import pandas as pa

pd = pa.read\_csv("heart.csv")

X = pd['age']

y = pd['thal']

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.4, random\_state=1)

X\_train.shape, X\_test.shape, y\_train.shape, y\_test.shape

from sklearn.naive\_bayes import GaussianNB

gnb = GaussianNB()

X\_train= X\_train.array.reshape(-1, 1)

X\_test = X\_test.array.reshape(-1, 1)

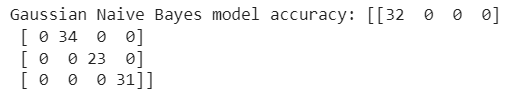
gnb.fit(X\_train, y\_train)

y\_pred = gnb.predict(X\_test)

from sklearn import metrics

print("Gaussian Naive Bayes model accuracy:", metrics.confusion\_matrix(y\_test, y\_pred))

**OUTPUT:**



**CONCLUSION:**

We have successfully implemented Naïve Bayes.

**Experiment No.: 10**

**AIM:** Program to implement Gradient Descent Algorithm using tensorflow

**Description** :

TensorFlow is an open-source software framework created by Google for building and training machine learning models. It provides a simple way to create and execute mathematical operations on large-scale datasets. TensorFlow supports a variety of deep learning algorithms and can be used for many different applications, including image recognition, natural language processing, and speech recognition. It also provides a variety of visualization tools to help users understand and debug their models. TensorFlow is one of the most widely used machine learning frameworks in the world and is used by researchers, businesses, and developers to build and train sophisticated AI models.

Gradient descent is an iterative optimization algorithm used to minimize a cost function by updating the parameters of a model in the direction of steepest descent or negative gradient.

The formula for gradient descent is as follows:

For a cost function J(theta), where theta is a vector of parameters:

repeat until convergence { theta\_j = theta\_j - alpha \* partial derivative of J(theta) with respect to theta\_j }

where alpha is the learning rate, which controls the size of the steps taken during each iteration.

The partial derivative of J(theta) with respect to thetaj is the rate of J(theta) change of with respect to thetaj, or how much J(theta) changes for a small change in . This rate of change determines the direction of steepest descent, which is the direction that reduces the value of J(theta) the most. Gradient descent works by iteratively updating the parameters of the model in the direction of negative gradient, which is the opposite of the direction of steepest ascent. By repeating these updates and gradually reducing the value of J(theta), gradient descent converges to a minimum value of J(theta) that corresponds to the optimal set of parameters for the model.

**PROGRAM:**

import numpy as np

import tensorflow as tf

x\_train = np.random.rand(100, 1) \* 10

y\_train = 2 \* x\_train - 3 + np.random.randn(100, 1)

model = tf.keras.Sequential([tf.keras.layers.Dense(units=1, input\_shape=[1])])

loss\_fn = tf.keras.losses.MeanSquaredError()

optimizer = tf.keras.optimizers.SGD(learning\_rate=0.01)

train\_dataset = tf.data.Dataset.from\_tensor\_slices(

        (x\_train, y\_train)).batch(32)

for epoch in range(1000):

    for x\_batch, y\_batch in train\_dataset:

        with tf.GradientTape() as tape:

            y\_pred = model(x\_batch)

            loss = loss\_fn(y\_batch, y\_pred)

        gradients = tape.gradient(loss, model.trainable\_variables)

        optimizer.apply\_gradients(zip(gradients, model.trainable\_variables))

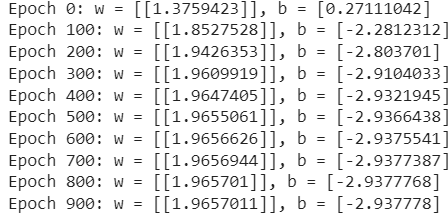
    if epoch % 100 == 0:

        # Print the weights every 100 epochs

        print("Epoch {}: w = {}, b = {}".format(

            epoch, model.get\_weights()[0], model.get\_weights()[1]))

        model.get\_weights()

**OUTPUT:  
**

****

**CONCLUSION:**

We have successfully implemented Gradient Descent Algorithm using tensorflow.

**Experiment No.: 11**

**AIM:** Program to implement KNN on a dataset

**Description**: K-nearest neighbors algorithm (KNN) is a type of supervised machine learning algorithm that is used for classification and regression tasks. It determines the class of an unknown sample data point by looking at the K number of nearest neighbors in the training set. The distance between the data points is calculated based on various metrics like Euclidean distance, Manhattan distance, etc. The algorithm classifies the new data point based on the majority class of the K nearest neighbors. K represents the number of neighbors we consider in the model.

The k-Nearest Neighbors algorithm can be formulated as follows:

Given a set of labelled training data , where each is a feature vector and each is its corresponding class label, and a new unlabeled sample :

1. Compute the distance or similarity between and each using a distance or similarity metric, such as Euclidean distance or cosine similarity.
2. Select the k training samples with the smallest distances/similarities to , forming a set .
3. Assign the class label that is most frequent among the k nearest neighbors .

Mathematically, this can be expressed as: S = {xi1 , xi2 ,…, xik },

y = arg max j{1,…,C} k ∑ l=1 [yil = j]

where S is the set of k nearest neighbors, is the number of distinct class labels, and is the predicted class label of x.

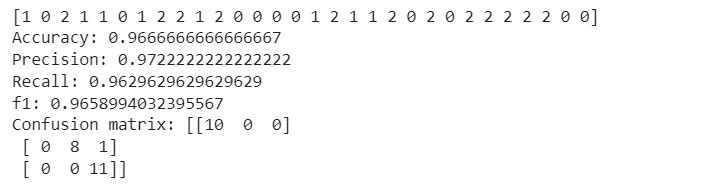
**CODE:**

|  |
| --- |
| from sklearn import datasets from sklearn import metrics  from sklearn.neighbors import KNeighborsClassifier from sklearn.model\_selection import train\_test\_split iris = datasets.load\_iris() X = iris['data'] y = iris['target']  X\_train, X\_test, y\_train, y\_test = train\_test\_split( X, y, test\_size = 0.2, random\_state=42)  knn = KNeighborsClassifier(n\_neighbors=7) model=knn.fit(X\_train, y\_train) y\_pred = knn.predict(X\_test) y\_pred.shape print(y\_pred)  print("Accuracy:",metrics.accuracy\_score(y\_test, y\_pred)) print("Precision:",metrics.precision\_score(y\_test, y\_pred, average = 'macro')) |

print("Recall:",metrics.recall\_score(y\_test, y\_pred, average =

'macro')) print("f1:",metrics.f1\_score(y\_test, y\_pred, average = 'macro')) print("Confusion matrix:",metrics.confusion\_matrix(y\_test, y\_pred))

**OUTPUT:**



**Conclusion:**

Hence, The KNN has been implemented for iris dataset, different metrics calculated successfully

**Experiment No.: 12**

**AIM:** Program to implement Agglomerative Clustering on a dataset

**Description:**

Agglomerative is a type of hierarchical clustering algorithm used in machine learning and data mining. It starts with considering each data point as a separate cluster and merges similar clusters to form larger clusters until all the data points are in a single cluster. It involves calculating the similarity or distance between two data points or clusters and forming the new cluster by combining the two most similar clusters. The output of the agglomerative clustering algorithm is a tree-like structure called a dendrogram, representing how the clusters are merged. The agglomerative clustering algorithm can be represented using the following mathematical:

**Input:** {x1, x2,…, xn}

**Output:** Dendrogram of clusters

**Step 1:** Initialize the clusters as C = {{x1}, {x2},…, {xn}}

**Step 2:** Compute the pairwise distances between all clusters using a suitable distance metric d(ci, cj)

**Step 3:** Find the two closest clusters ci ∗ , cj∗ ∈ C according to the distance metric

**Step 4:** Merge the two closest clusters into a single cluster ci ∗ U cj∗

**Step 5:** Update the set of clusters C = C ∖ { ci ∗ , cj∗ } ∪ {Ck } where Ck = ci ∗ , cj∗

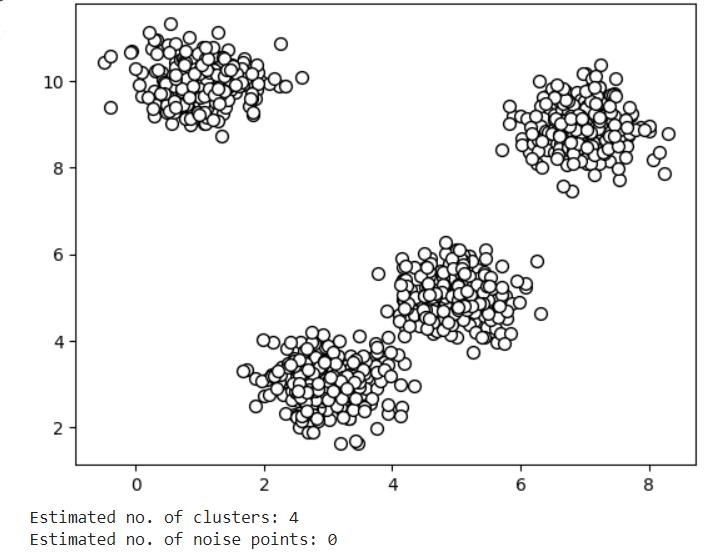
**Step 6:** Repeat steps 2-5 until all points are in a single cluster

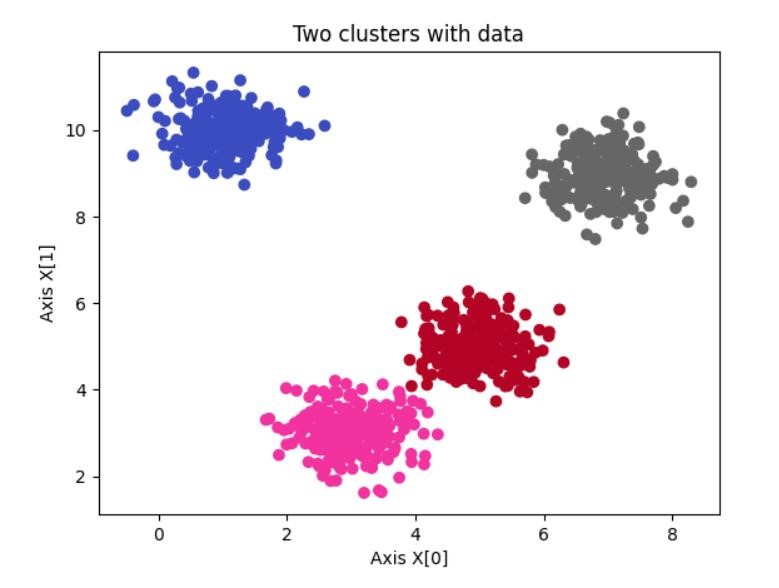
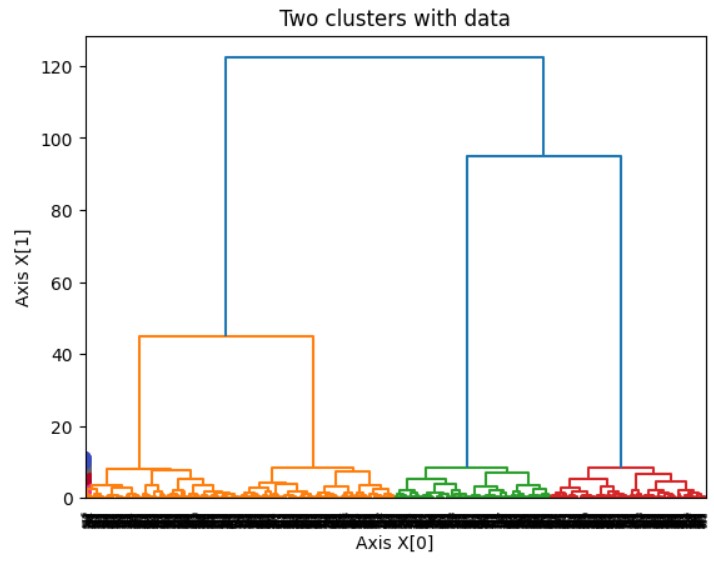
After applying this algorithm, we obtain a dendrogram of clusters that represents the hierarchical structure of the data.

**CODE:**

|  |
| --- |
| import numpy as np import matplotlib.pyplot as plt from sklearn.datasets import make\_blobs  from sklearn.cluster import AgglomerativeClustering import scipy.cluster.hierarchy as sch num\_samples\_total = 1000  cluster\_centers = [(3,3), (7,9), (1, 10), (5, 5)] num\_classes = len(cluster\_centers) epsilon = 1.0 min\_samples = 13 X, y = make\_blobs( n\_samples=num\_samples\_total, n\_features=num\_classes, centers=cluster\_centers, cluster\_std=.5, shuffle=True, random\_state=0, center\_box=(0, 1),  ) |
| plt.scatter( X[:, 0], X[:, 1], c='white', marker='o', edgecolor='black', s=50  ) plt.show() dendrogram = sch.dendrogram(sch.linkage(X, method='ward')) m = AgglomerativeClustering( n\_clusters=num\_classes, metric='euclidean', linkage='ward',  ) y\_m = m.fit(X) labels = y\_m.labels\_ no\_clusters = len(np.unique(labels) ) no\_noise = np.sum(np.array(labels) == -1, axis=0) print('Estimated no. of clusters: %d' % no\_clusters) print('Estimated no. of noise points: %d' % no\_noise) colors = list(map(lambda x: ['#3b4cc0', '#b40426', '#666666',  '#f1329f'][x], labels)) plt.scatter(X[:,0], X[:,1], c=colors, marker="o", picker=True) plt.title('Two clusters with data') plt.xlabel('Axis X[0]') plt.ylabel('Axis X[1]') plt.show() |

**OUTPUT:**





**Conclusion:**

Hence, The Agglomerative Clustering has been implemented for Blobs dataset, different cluster were identified successfully