Experiment No.: 01

AIM:

Program to implement Linear Regression on a dataset

Description

Linear regression is a statistical method used to examine the relationship between a dependent variable and one or more independent variables. It is used to predict the value of the dependent variable based on the given values of the independent variable(s). Linear regression model assumes that there is a linear relationship between the independent variable(s) and the dependent variable. The objective of linear regression is to find the best-fit line that represents the relationship between the variables. The line is obtained by minimizing the sum of squared differences between the observed values and the predicted values of the dependent variable. Linear regression is widely used in various fields, such as economics, finance, marketing, and social sciences, to make predictions and to identify patterns in large datasets.

$$f_{w,b}(x) = wx + b$$

where, w is weight, b is bias, x is feature or input variable and $f_{w,b}(x)$ is predicted value.

In [ ]:
```
# Imports
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score
```

In [ ]:
```
# Loading dataset
df = pd.read_csv("tvmarketing.csv")
df
```

Out[ ]:

	TV	Sales
0	230.1	22.1
1	44.5	10.4
2	17.2	9.3
3	151.5	18.5
4	180.8	12.9
...	...	...
195	38.2	7.6
196	94.2	9.7
197	177.0	12.8
198	283.6	25.5
199	232.1	13.4

200 rows × 2 columns

In [ ]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 200 entries, 0 to 199
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   TV      200 non-null    float64
 1   Sales   200 non-null    float64
dtypes: float64(2)
memory usage: 3.2 KB
```

In [ ]:
```
df.describe()
```

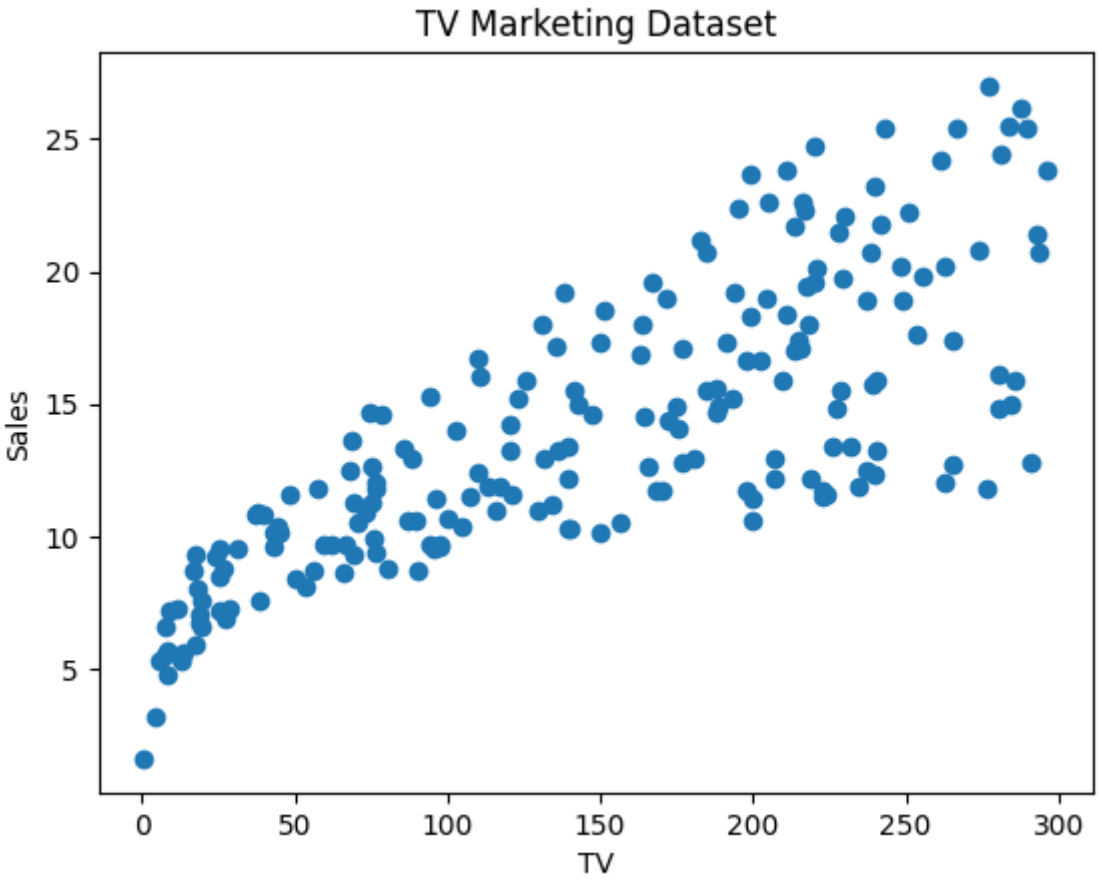
Out[ ]:

	TV	Sales
count	200.000000	200.000000
mean	147.042500	14.022500
std	85.854236	5.217457
min	0.700000	1.600000
25%	74.375000	10.375000
50%	149.750000	12.900000
75%	218.825000	17.400000
max	296.400000	27.000000

In [ ]:
```
# Identifying
x = df['TV']
y = df['Sales']
```

In [ ]:
```
# Plotting x & y
plt.scatter(x, y)
plt.title("TV Marketing Dataset")
plt.xlabel("TV")
plt.ylabel("Sales")
```

Out[ ]: Text(0, 0.5, 'Sales')



In [ ]:
```
# Splitting the dataset
x_train, x_test, y_train, y_test = train_test_split(x, y, train_size=.6, random
x_train = np.array(x_train)
x_test = np.array(x_test)
x_train = x_train.reshape((*x_train.shape, 1))
x_test = x_test.reshape((*x_test.shape, 1))

print(x_train.shape)
print(y_train.shape)
print(x_test.shape)
print(y_test.shape)
```

```
(120, 1)
(120,)
(80, 1)
(80,)
```

In [ ]:
```
# Training the dataset
m = LinearRegression()
m.fit(x_train, y_train)
```

Out[ ]:
```
▾ LinearRegression
LinearRegression()
```

In [ ]:
```
# Intercept and Coefficient
(m.intercept_, m.coef_)
```

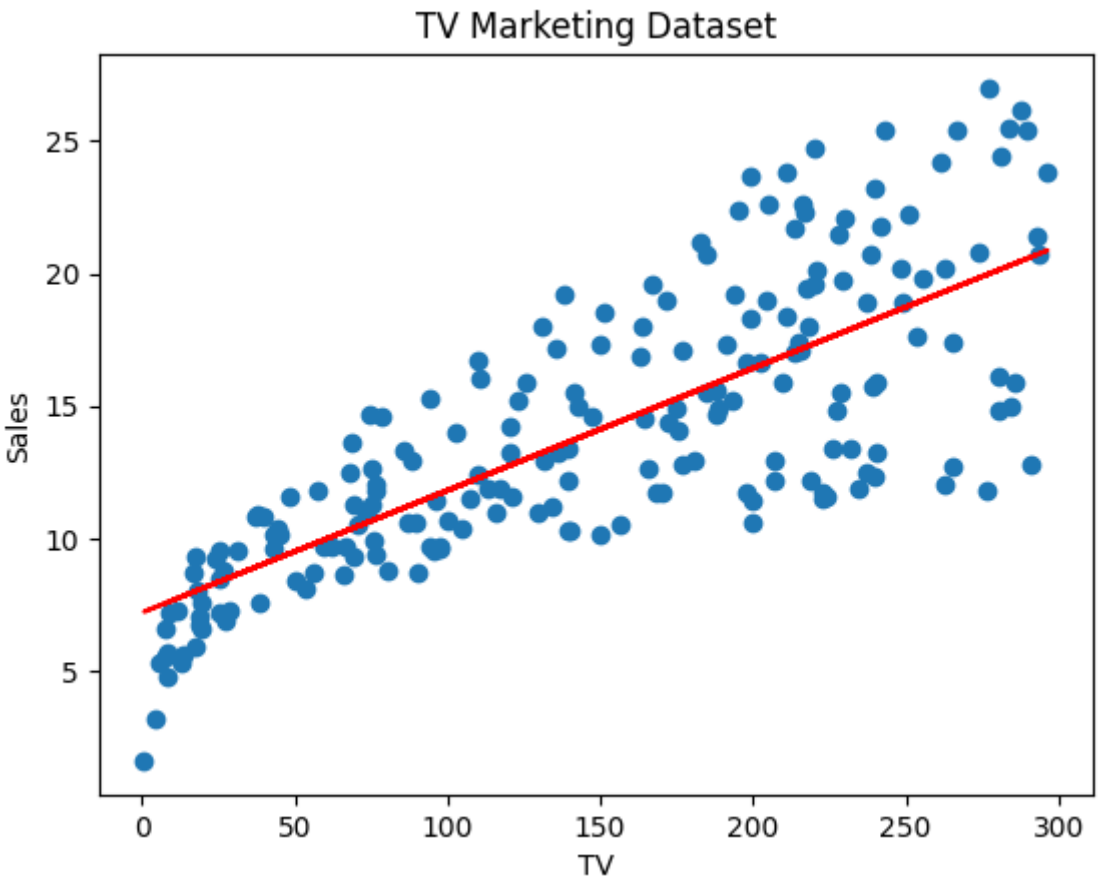
Out[ ]: (7.194488785595947, array([0.04613525]))

In [ ]:
```
# Predicting the test set
y_pred = m.predict(x_test)
y_pred[:5]
```

Out[ ]:
```
array([10.38704839,  9.50125151, 11.36511578, 20.55987199, 15.04670909])
```

In [ ]:
```
# Plotting x & y and line
plt.plot(x_test, y_pred, c='r')
plt.scatter(x, y)
plt.title("TV Marketing Dataset")
plt.xlabel("TV")
plt.ylabel("Sales")
```

Out[ ]:
```
Text(0, 0.5, 'Sales')
```



In [ ]:
```
# Error and R2 Score
mse = mean_squared_error(y_test, y_pred)
r_sq = r2_score(y_test, y_pred)

print("Mean Square: ", mse)
print("R2 Score(Accuracy): ", r_sq)
```

```
Mean Square:  8.18918415499061
R2 Score(Accuracy):  0.688901987584994
```

Conclusion:

Hence, The Linear Regression has been implemented for tv marketing dataset, Mean Squared Error (MSE) and R2 Score calculated successfully

Experiment No.: 02

AIM:

Preform Data Visualization on dataset using matplotlib and seaborn on winequality dataset

Description:

Matplotlib is a widely-used data visualization and plotting library in Python that helps produce publication-quality figures. It allows users to create various 2D and 3D plots, such as line graphs, scatter plots, bar plots, histograms, and more. Additionally, it allows for customizing various aspects of the plot including labels, titles, colors, axis scales etc. Matplotlib is integrated with the NumPy library and can be used in conjunction with other libraries such as Pandas and SciPy.

Seaborn is a Python data visualization library based on Matplotlib. It provides a high-level interface for creating informative and attractive statistical graphics. Seaborn includes built-in datasets, visualizations for regression analysis, multivariate data analysis, time series analysis, and categorical data analysis. The library applies sensible defaults to plots, and customizes the output with a few simple changes. Its functionality is based on statistical knowledge and conventions, making it more user-friendly to non-statisticians.

In [ ]:
```
# imports
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sn
%matplotlib inline
```

In [ ]:
```
# loading dataset
df = pd.read_csv('winequality-red.csv')
df
```

Out[ ]:

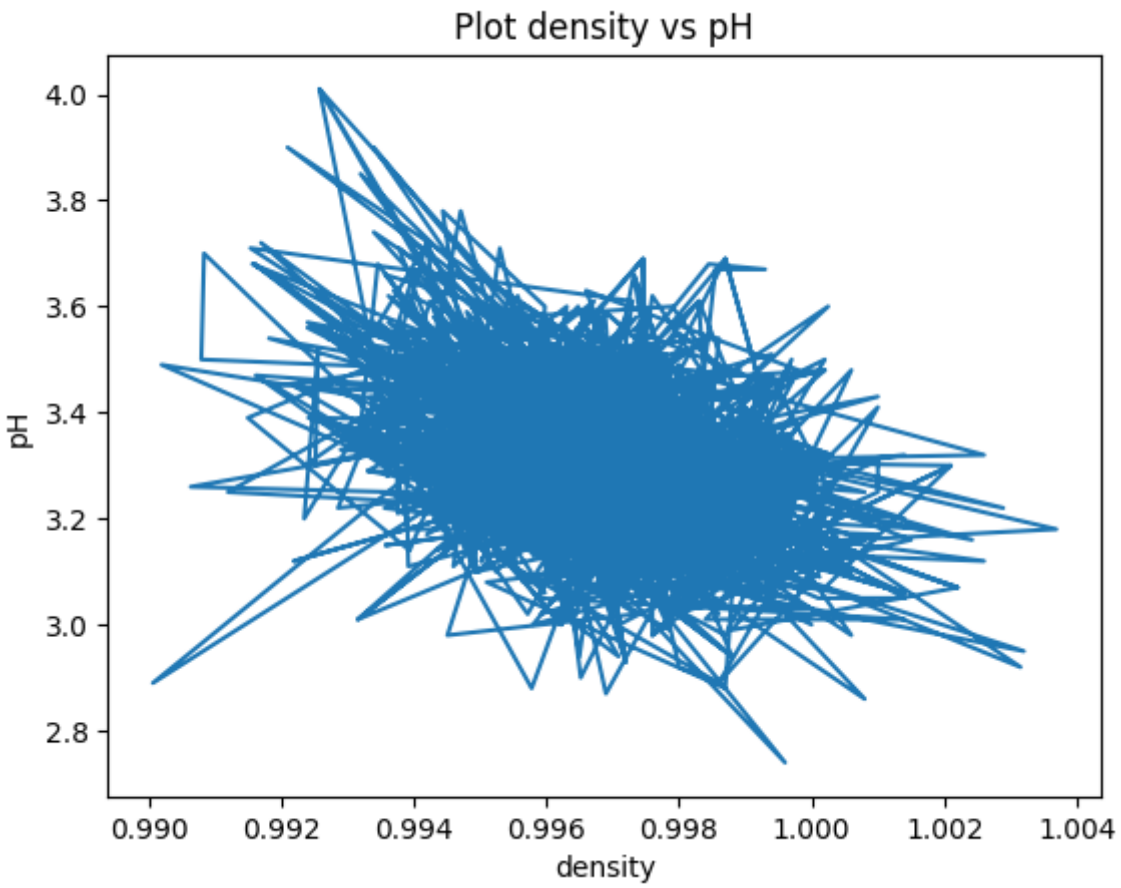
	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol
0	7.4	0.700	0.00	1.9	0.076	11.0	34.0	0.99780	3.51	0.56	9.4
1	7.8	0.880	0.00	2.6	0.098	25.0	67.0	0.99680	3.20	0.68	9.8
2	7.8	0.760	0.04	2.3	0.092	15.0	54.0	0.99700	3.26	0.65	9.8
3	11.2	0.280	0.56	1.9	0.075	17.0	60.0	0.99800	3.16	0.58	9.8
4	7.4	0.700	0.00	1.9	0.076	11.0	34.0	0.99780	3.51	0.56	9.4
...	...	...	...	...	...	...	...	...	...	...	...
1594	6.2	0.600	0.08	2.0	0.090	32.0	44.0	0.99490	3.45	0.58	10.5
1595	5.9	0.550	0.10	2.2	0.062	39.0	51.0	0.99512	3.52	0.76	11.2
1596	6.3	0.510	0.13	2.3	0.076	29.0	40.0	0.99574	3.42	0.75	11.0
1597	5.9	0.645	0.12	2.0	0.075	32.0	44.0	0.99547	3.57	0.71	10.2
1598	6.0	0.310	0.47	3.6	0.067	18.0	42.0	0.99549	3.39	0.66	11.0

1599 rows × 12 columns

Plots using `matplotlib`

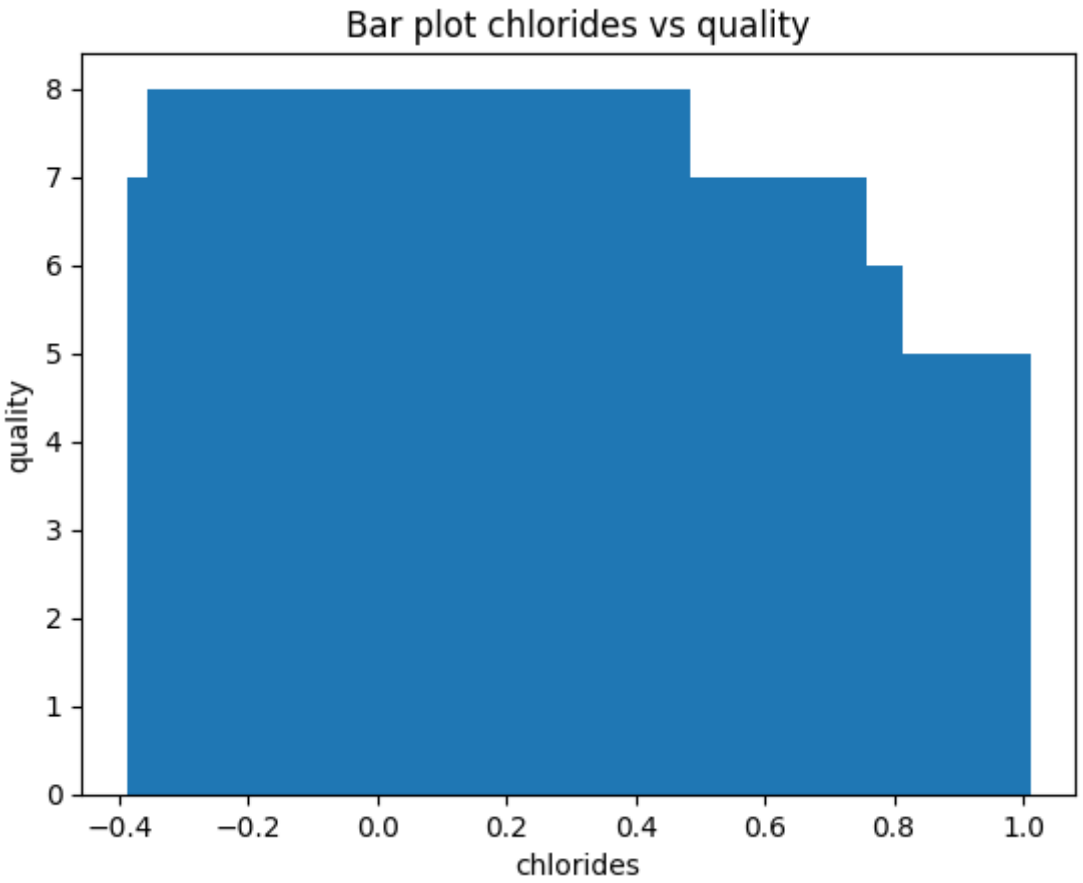
Plot

In [ ]:
```
plt.plot(df['density'], df['pH'])
plt.title('Plot density vs pH')
plt.xlabel('density')
plt.ylabel('pH')
plt.show()
```



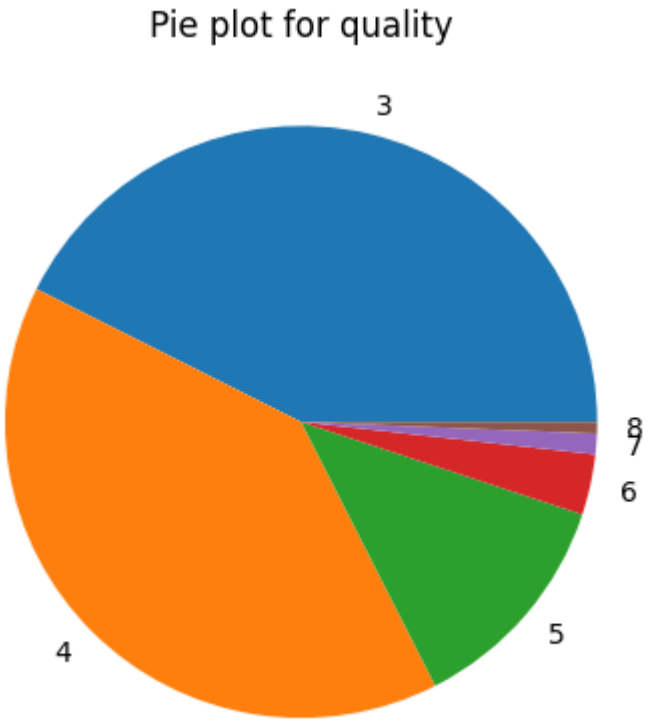
Bar Plot

In [ ]:
```
plt.bar(df['chlorides'], df['quality'])
plt.title('Bar plot chlorides vs quality')
plt.xlabel('chlorides')
plt.ylabel('quality')
plt.show()
```



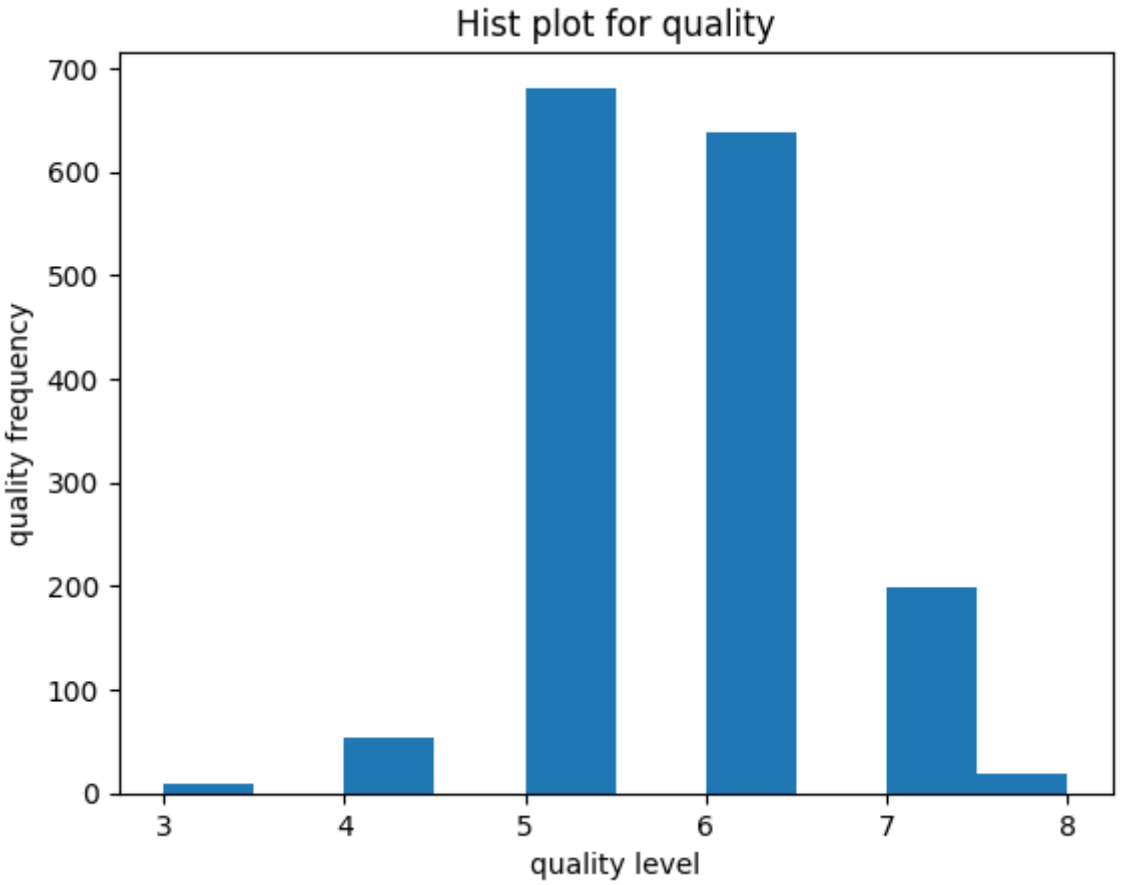
Pie Plot

In [ ]:
```
count = df['quality'].value_counts()
plt.pie(count, labels = np.array(list(set(df['quality']))))
plt.title('Pie plot for quality')
plt.show()
```



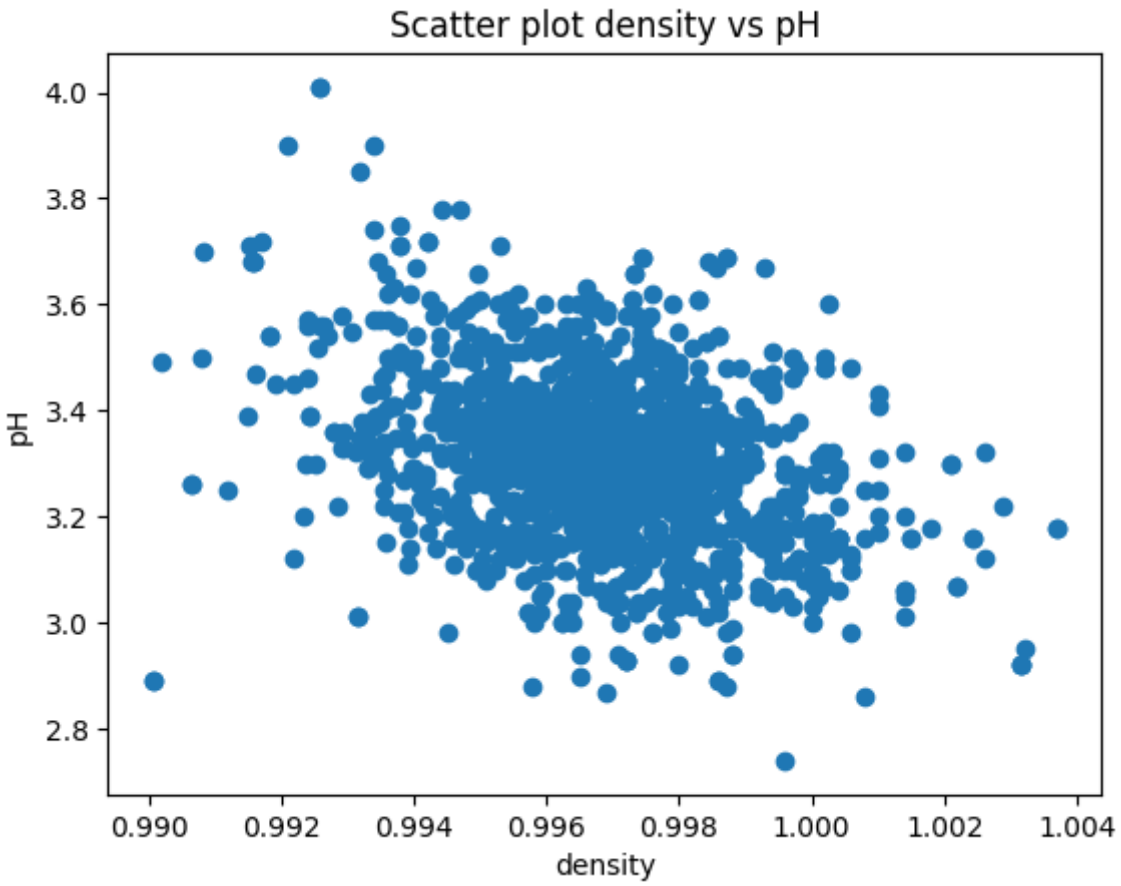
Hist Plot

In [ ]:
```
plt.hist(df['quality'])
plt.title('Hist plot for quality')
plt.xlabel('quality level')
plt.ylabel('quality frequency')
plt.show()
```



Scatter Plot

In [ ]:
```
plt.scatter(df['density'], df['pH'])
plt.title('Scatter plot density vs pH')
plt.xlabel('density')
plt.ylabel('pH')
plt.show()
```



BoxPlot

In [ ]:
```
plt.boxplot(df['citric acid'])
plt.title('Boxplot plot for citric acid')
plt.show()
```

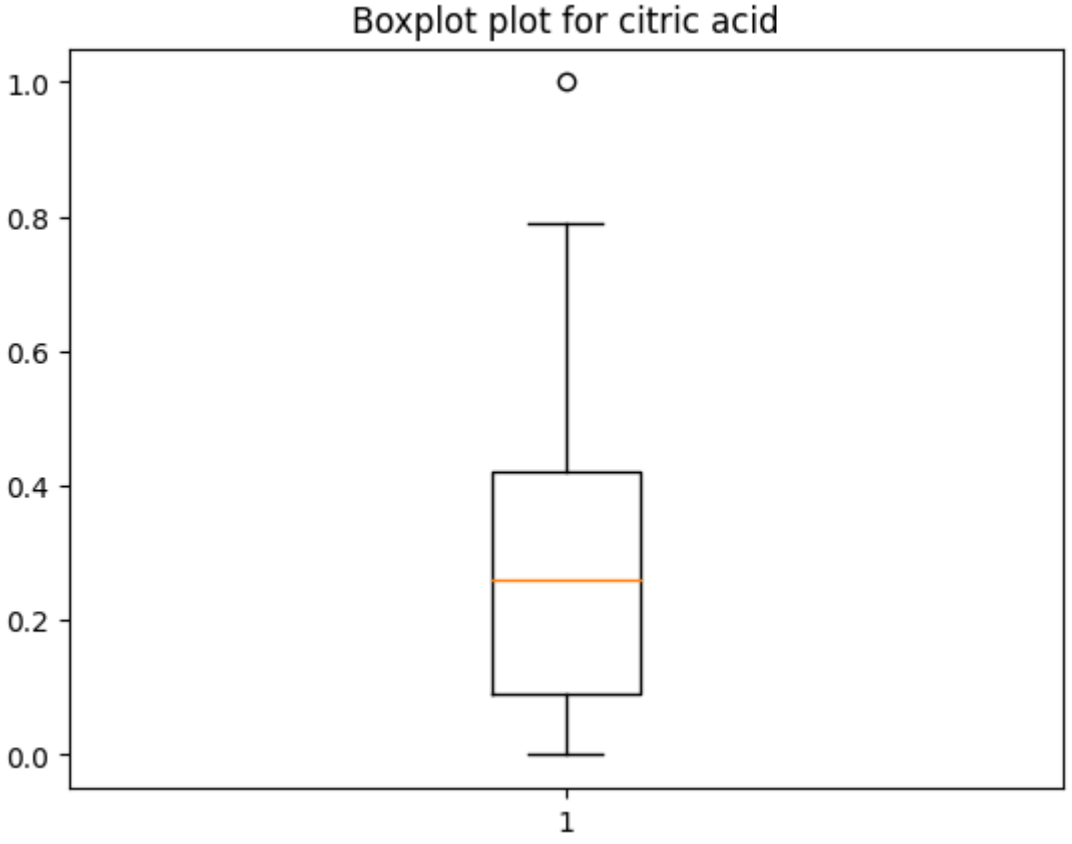
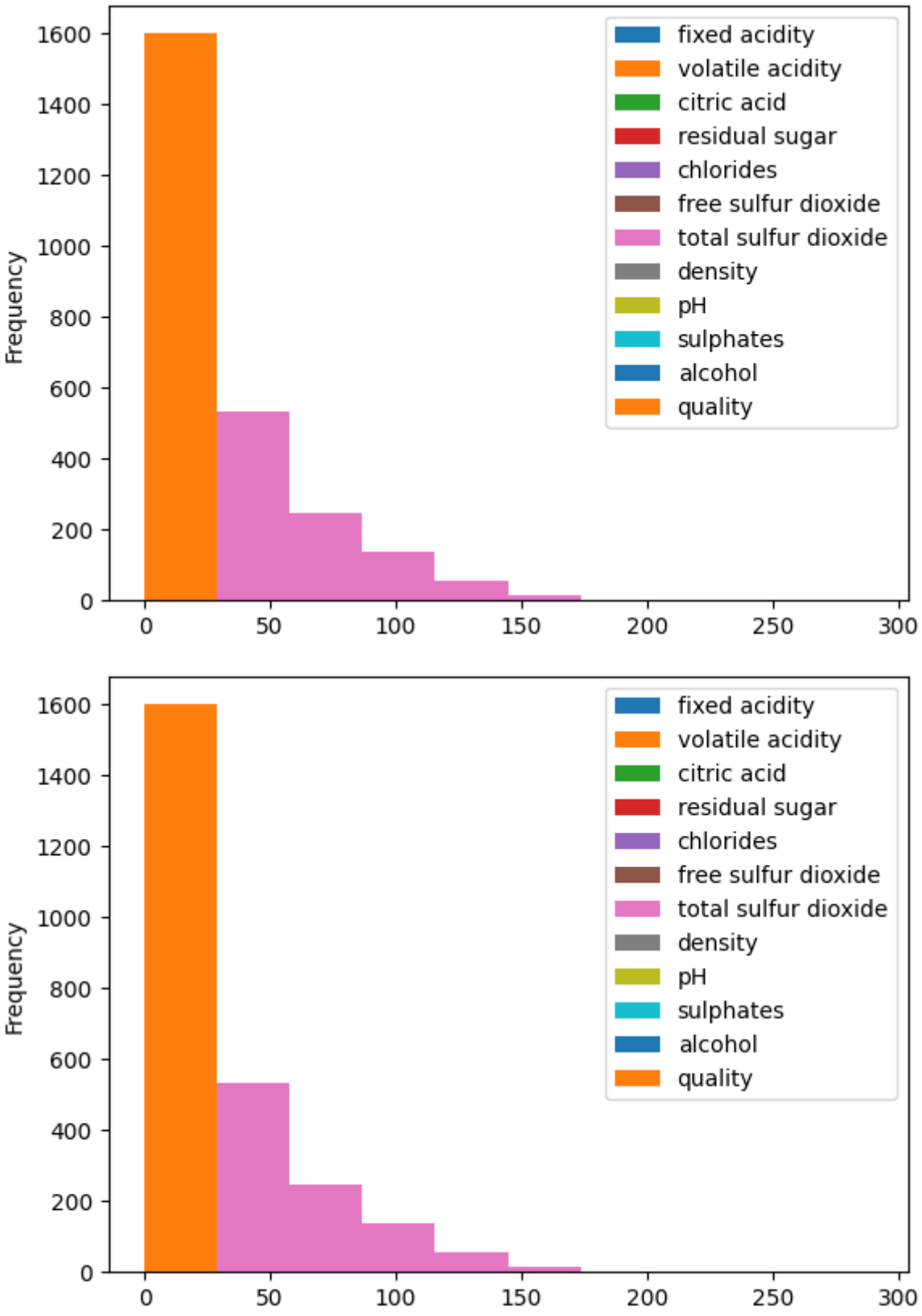


figure object

In [ ]:
```
fig = df.plot(kind="hist")
fig.get_figure()
```

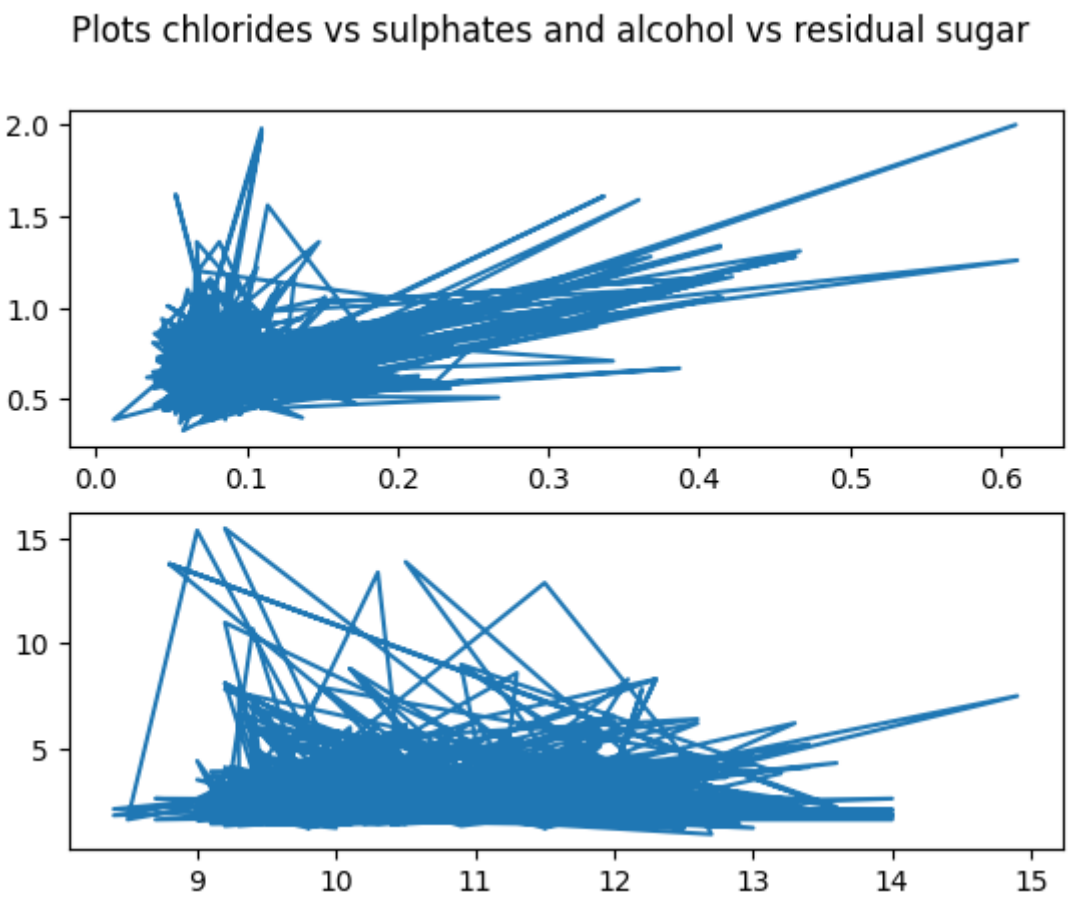
Out[ ]:



SubPlots

In [ ]:
```
fig, axs = plt.subplots(2)
fig.suptitle('Plots chlorides vs sulphates and alcohol vs residual sugar')
axs[0].plot(df['chlorides'], df['sulphates'])
axs[1].plot(df['alcohol'], df['residual sugar'])
```

Out[ ]:
```
[<matplotlib.lines.Line2D at 0x7f6fb3cbae60>]
```

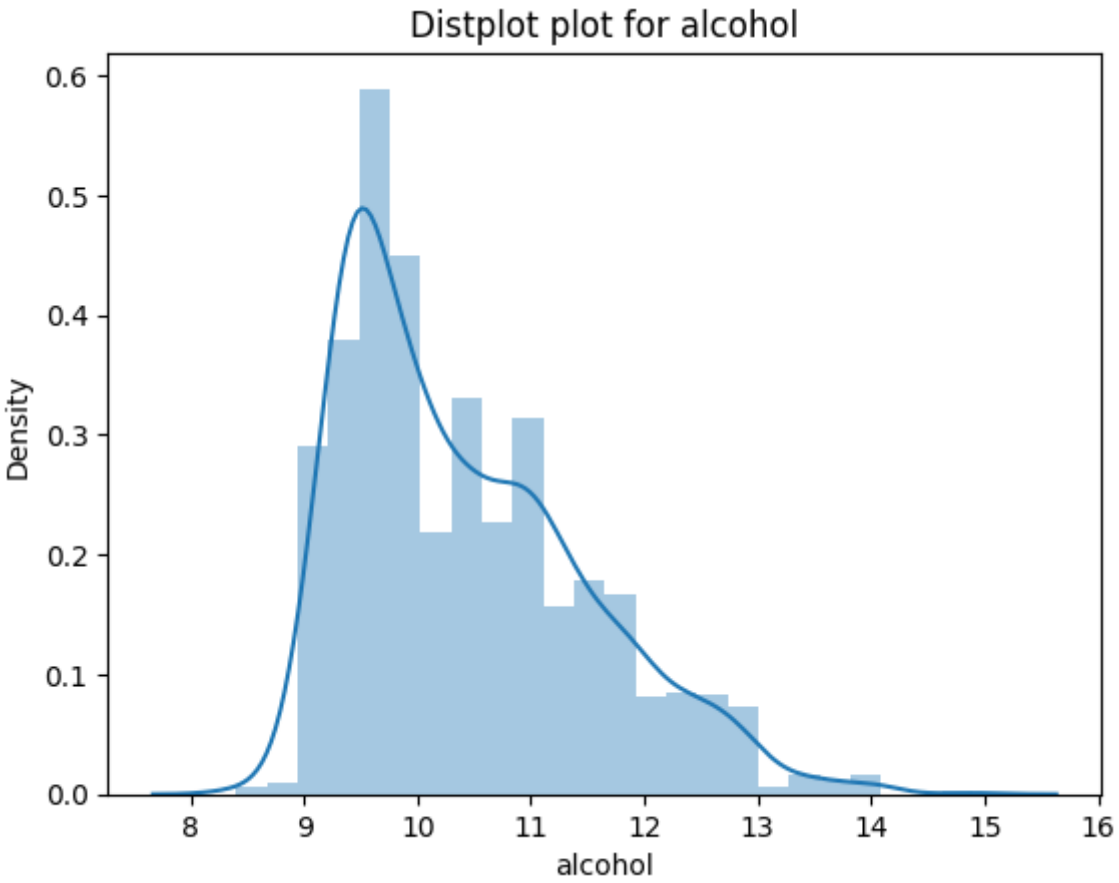


Plots using `seaborn`

Dist Plot

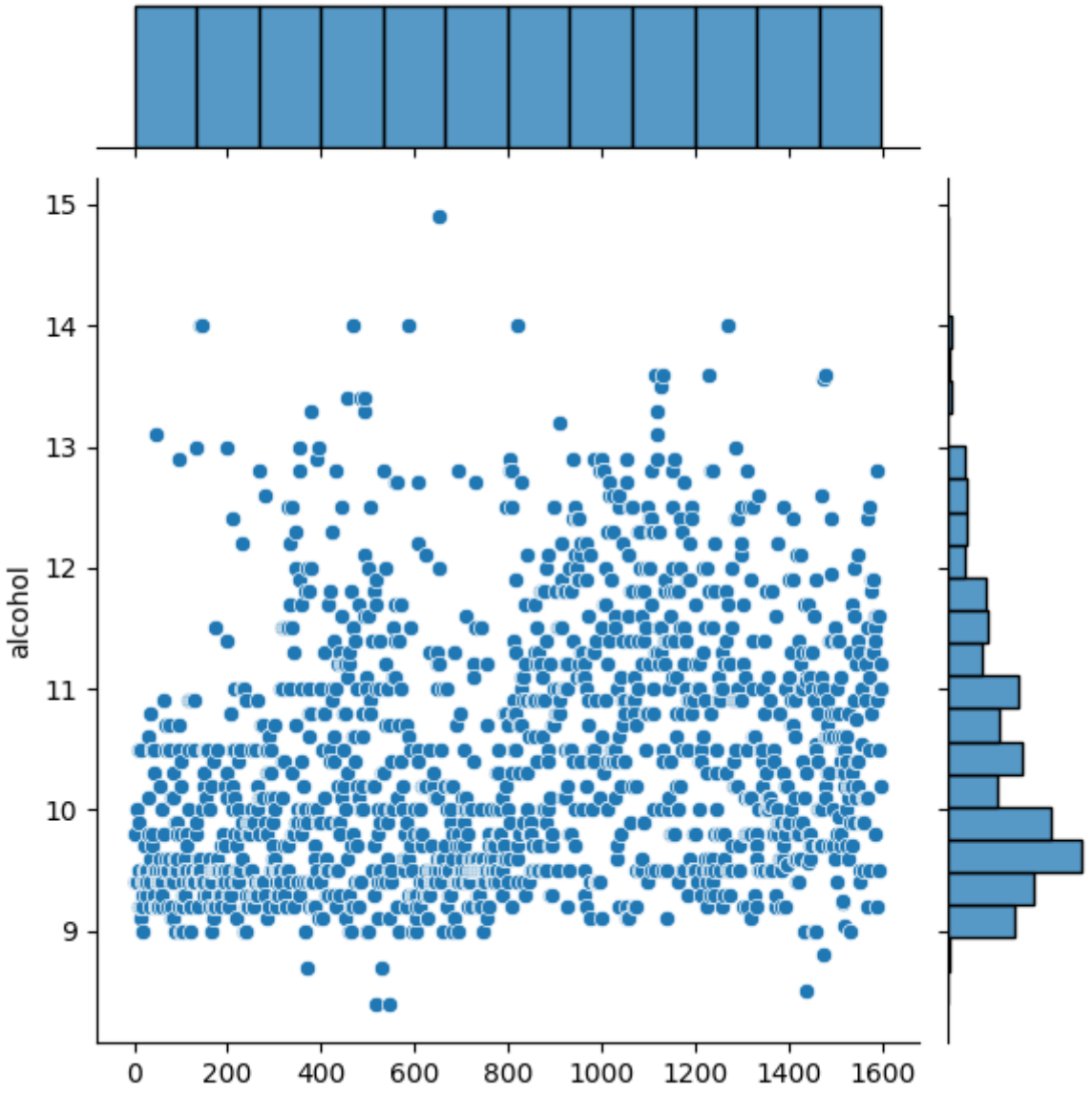
In [ ]:
```
sn.distplot(df['alcohol'])
plt.title('Distplot plot for alcohol')
plt.show()
```

```
/tmp/ipykernel_1638045/33422094.py:1: UserWarning:

`distplot` is a deprecated function and will be removed in seaborn v0.14.0.

Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `histplot` (an axes-level function for histograms).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

  sn.distplot(df['alcohol'])
```



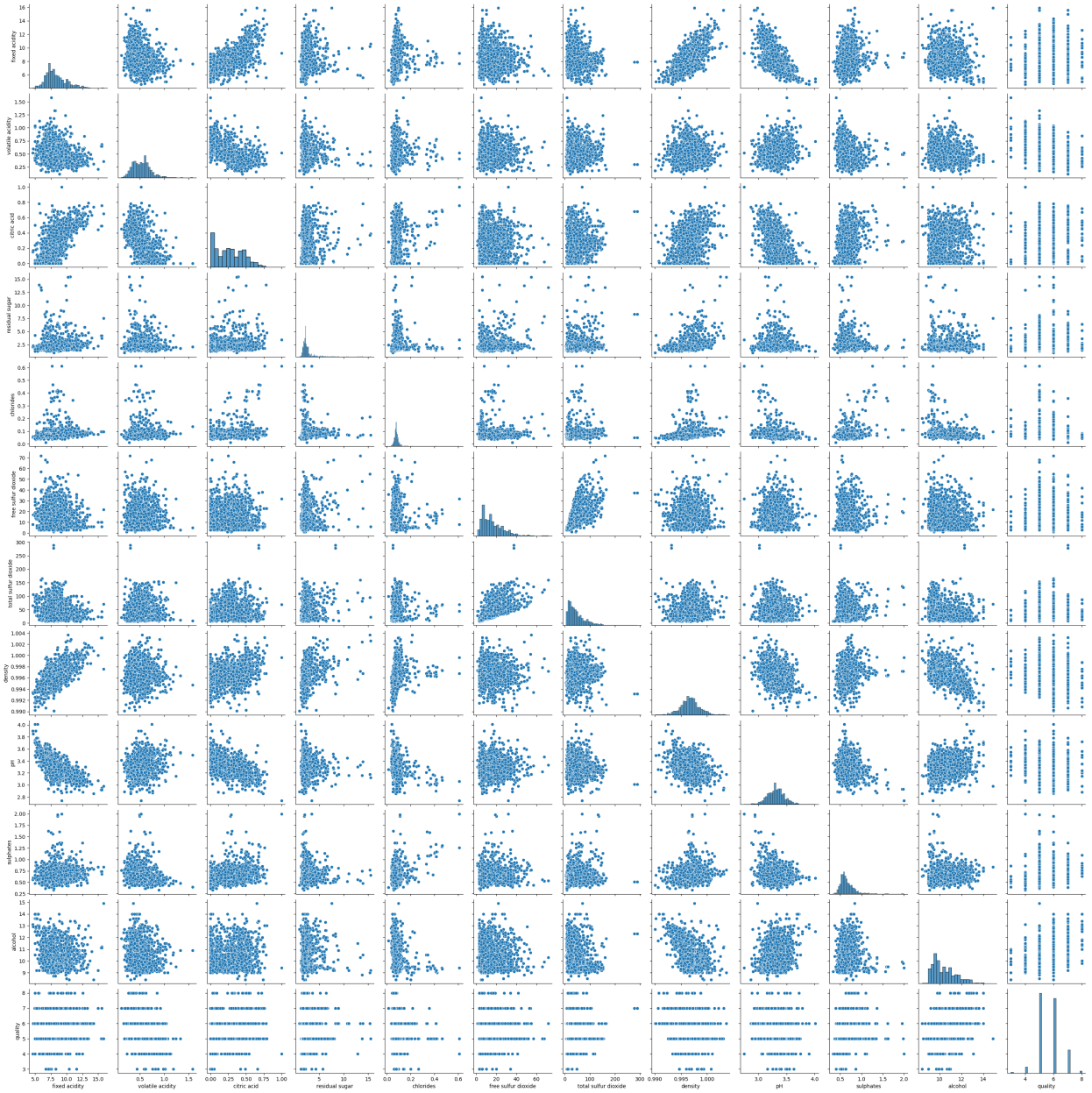
Joint Plot

In [ ]:
```
sn.jointplot(df['alcohol'])
plt.show()
```



Pair Plot

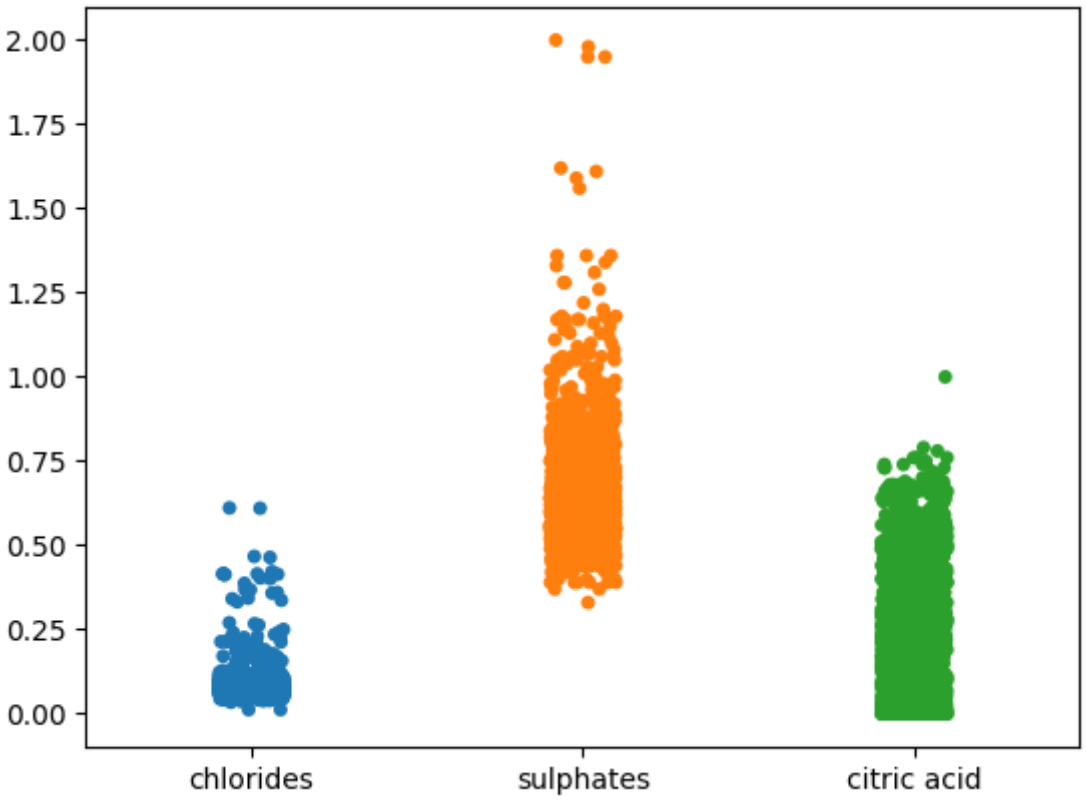
In [ ]:
```
sn.pairplot(df)
plt.show()
```



Strip Plot

In [ ]:
```
acids = pd.DataFrame(
    {'chlorides': df['chlorides'],
    'sulphates': df['sulphates'],
    'citric acid': df['citric acid']})
sn.stripplot(data=acids)
```

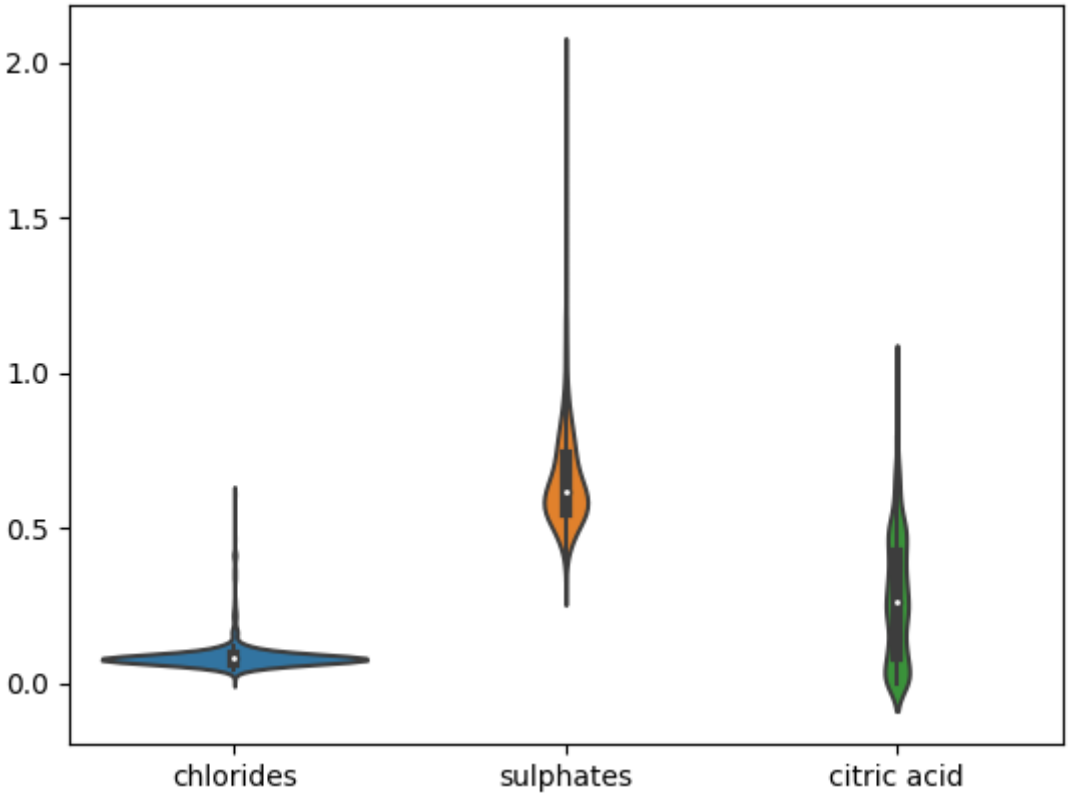
Out[ ]:
```
<Axes: >
```



Violin Plot

In [ ]:
```
sn.violinplot(data=acids)
```

Out[ ]:
```
<Axes: >
```



Conclusion:

Hence, The Data Visualization has been performed on wine quality dataset using matplotlib and seaborn successfully.

Experiment No.: 03

AIM:

To Perform Ridge and Lasso Regression on a dataset

Description:

Ridge and Lasso regression are some of the simple techniques to reduce model complexity and prevent over-fitting which may result from simple linear regression.

Let $\hat{y}$ be the prediction for the given n input features

$$\hat{y} = w_0 \times x_0 + w_1 \times x_1 + \ldots + w_n \times x_n + b \tag{1}$$

then, the cost function can be expressed as follows, assuming dataset has M instance of p features

$$\sum_{i=1}^{M}(y_i - \hat{y}_i)^2 = \sum_{i=1}^{M}\left(y_i - \sum_{j=0}^{p} w_j \times x_{ij}\right)^2 \tag{2}$$

Lasso Regression(L1 Normalization): The cost function for Lasso (least absolute shrinkage and selection operator) regression can be written as

$$\sum_{i=1}^{M}(y_i - \hat{y}_i)^2 = \sum_{i=1}^{M}\left(y_i - \sum_{j=0}^{p} w_j \times x_{ij}\right)^2 + \lambda \sum_{j=0}^{p} |w_j| \tag{3}$$

for some $t > 0, \sum_{j=0}^{p} |w_j| < t$, Lasso regression coefficients, subject to similar constrain as Ridge.

Ridge Regression(L2 Normalization): In ridge regression, the cost function is altered by adding a penalty equivalent to square of the magnitude of the coefficients.

$$\sum_{i=1}^{M}(y_i - \hat{y}_i)^2 = \sum_{i=1}^{M}\left(y_i - \sum_{j=0}^{p} w_j \times x_{ij}\right)^2 + \lambda \sum_{j=0}^{p} w_j^2 \tag{4}$$

for some $c > 0, \sum_{j=0}^{p} w_j^2 < c$, Constrain on Ridge regression coefficients

In [ ]:
```
# Imports
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression, Lasso, Ridge
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score
```

In [ ]:
```
# Loading dataset
df = pd.read_csv("tvmarketing.csv")
df
```

Out[ ]:

	TV	Sales
0	230.1	22.1
1	44.5	10.4
2	17.2	9.3
3	151.5	18.5
4	180.8	12.9
...	...	...
195	38.2	7.6
196	94.2	9.7
197	177.0	12.8
198	283.6	25.5
199	232.1	13.4

200 rows × 2 columns

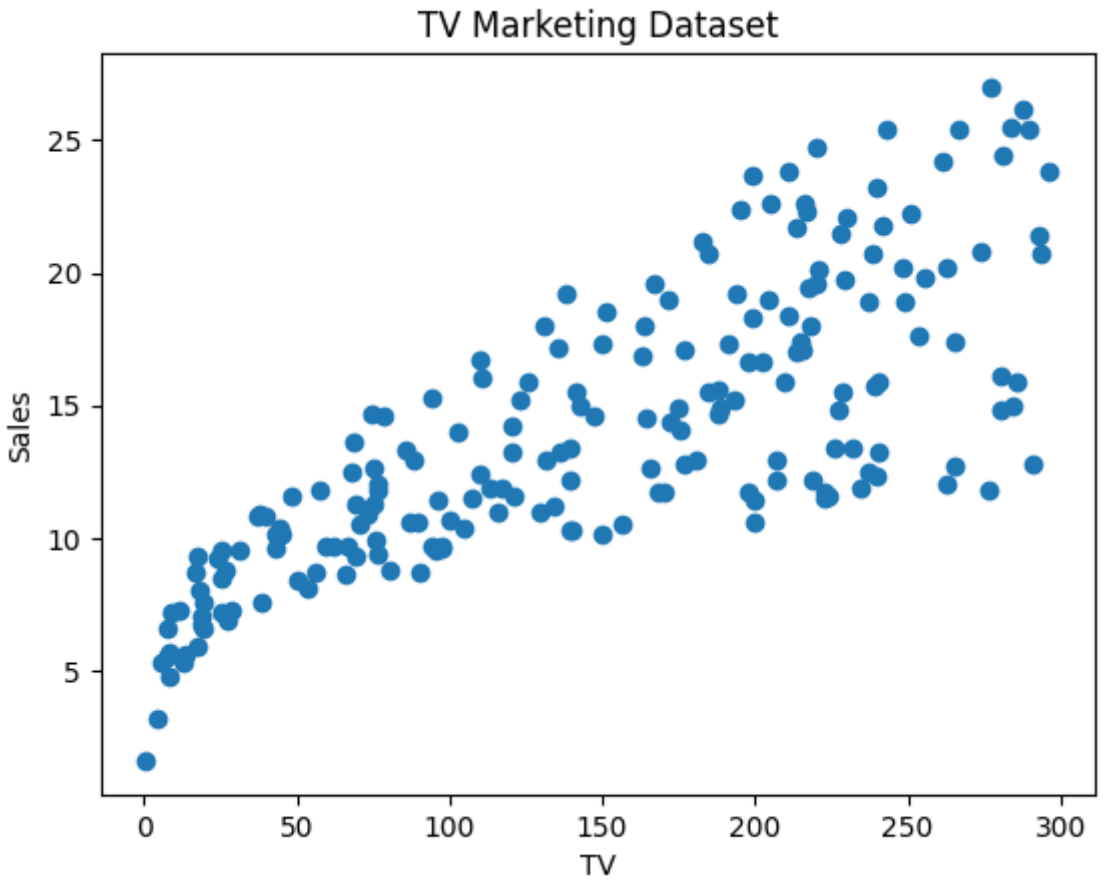
In [ ]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 200 entries, 0 to 199
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   TV      200 non-null    float64
 1   Sales   200 non-null    float64
dtypes: float64(2)
memory usage: 3.2 KB
```

In [ ]:
```
# Identifying
x = df['TV']
y = df['Sales']
```

In [ ]:
```
# Plotting x & y
plt.scatter(x, y)
plt.title("TV Marketing Dataset")
plt.xlabel("TV")
plt.ylabel("Sales")
```

Out[ ]:
```
Text(0, 0.5, 'Sales')
```



In [ ]:
```
# Splitting the dataset
x_train, x_test, y_train, y_test = train_test_split(x, y, train_size=.6, random
x_train = np.array(x_train)
x_test = np.array(x_test)
x_train = x_train.reshape((*x_train.shape, 1))
x_test = x_test.reshape((*x_test.shape, 1))

print(x_train.shape)
print(y_train.shape)
print(x_test.shape)
print(y_test.shape)
```

```
(120, 1)
(120,)
(80, 1)
(80,)
```

In [ ]:
```
# Training the dataset
linear_model = LinearRegression()
linear_model.fit(x_train, y_train)

# Intercept and Coefficient
(linear_model.intercept_, linear_model.coef_)
```

Out[ ]:
```
(7.194488785595947, array([0.04613525]))
```

In [ ]:
```
# Predicting the test set
y_pred_linear = linear_model.predict(x_test)

# Error and R2 Score
mse = mean_squared_error(y_test, y_pred_linear)
r_sq = r2_score(y_test, y_pred_linear)

print("Mean Square: ", mse)
print("R2 Score(Accuracy): ", r_sq)
```

```
Mean Square:  8.18918415499061
R2 Score(Accuracy):  0.688901987584994
```

Lasso Regression

In [ ]:
```
# Training the dataset
lasso_model = Lasso(alpha=0.001, max_iter=10 ** 3)
lasso_model.fit(x_train, y_train)

# Intercept and Coefficient
(lasso_model.intercept_, lasso_model.coef_)
```

Out[ ]:
```
(7.194509056071489, array([0.04613512]))
```

In [ ]:
```
# Predicting the test set
y_pred_lasso = lasso_model.predict(x_test)

# Error and R2 Score
mse = mean_squared_error(y_test, y_pred_lasso)
r_sq = r2_score(y_test, y_pred_lasso)

print("Mean Square: ", mse)
print("R2 Score(Accuracy): ", r_sq)
```

```
Mean Square:  8.189191191318532
R2 Score(Accuracy):  0.6889017202827097
```

Ridge Regression

In [ ]:
```
# Training the dataset
ridge_model = Lasso(alpha=0.001, max_iter=10 ** 3)
ridge_model.fit(x_train, y_train)

# Intercept and Coefficient
(ridge_model.intercept_, ridge_model.coef_)
```

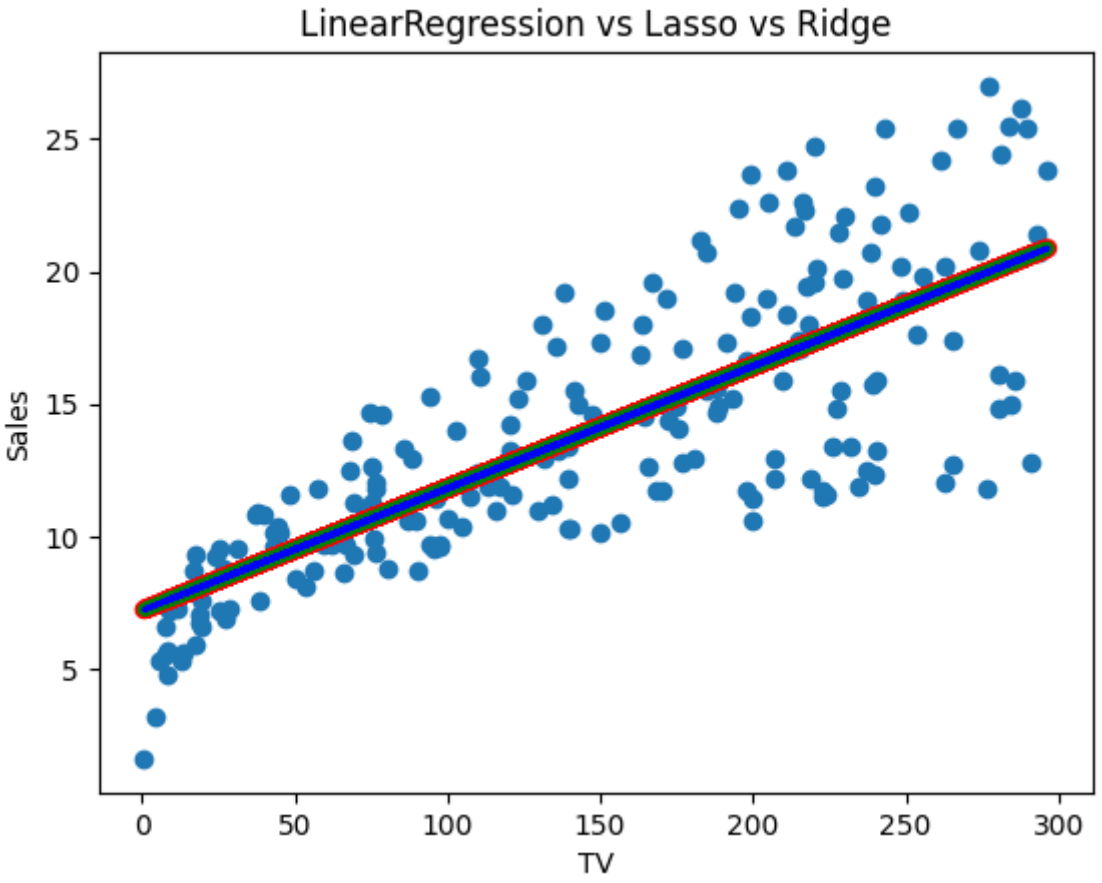
Out[ ]:
```
(7.194509056071489, array([0.04613512]))
```

In [ ]:
```
# Predicting the test set
y_pred_ridge = ridge_model.predict(x_test)

# Error and R2 Score
mse = mean_squared_error(y_test, y_pred_ridge)
r_sq = r2_score(y_test, y_pred_ridge)

print("Mean Square: ", mse)
print("R2 Score(Accuracy): ", r_sq)
```

```
Mean Square:  8.189191191318532
R2 Score(Accuracy):  0.6889017202827097
```

In [ ]:
```
# Plots to compare LinearRegression vs Lasso vs Ridge
plt.plot(x_test, y_pred_linear, c='r', lw=7)
plt.plot(x_test, y_pred_lasso, c='g', lw=5)
plt.plot(x_test, y_pred_ridge, c='b', lw=2)
plt.scatter(x, y)
plt.title("LinearRegression vs Lasso vs Ridge")
plt.xlabel("TV")
plt.ylabel("Sales")
```

Out[ ]:
```
Text(0, 0.5, 'Sales')
```



Conclusion:

Hence, Ridge and Lasso Regression has been performed and difference are identified successfully.

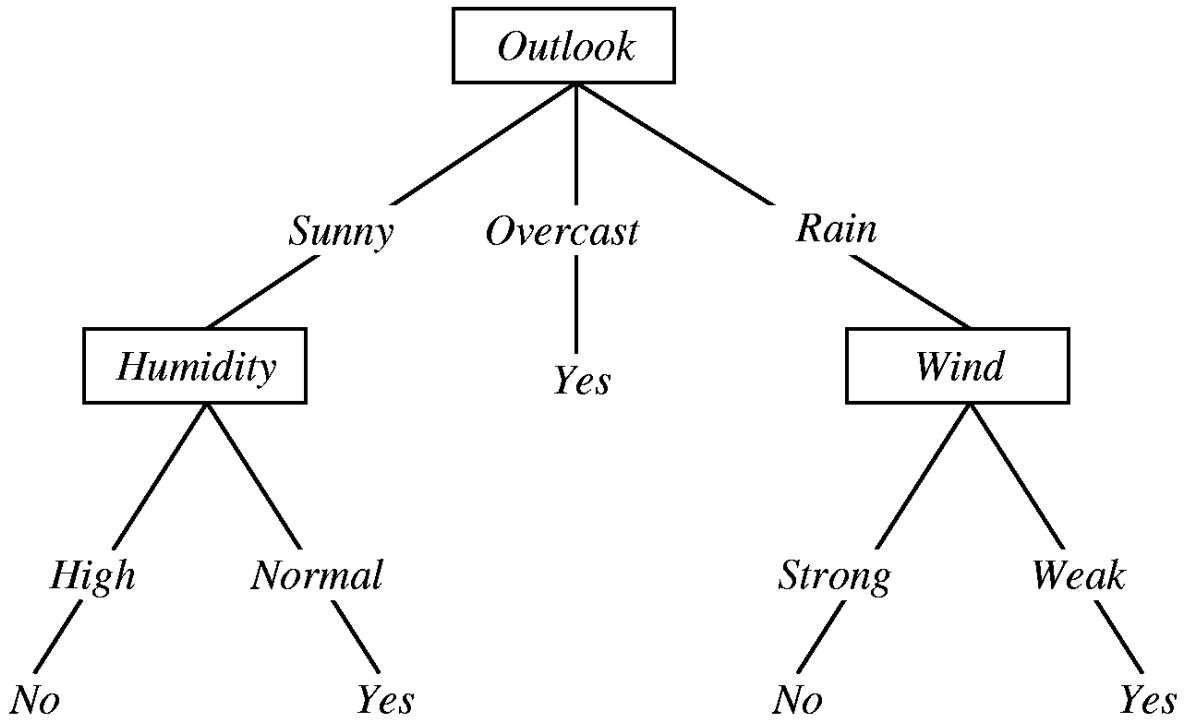
Experiment No.: 04

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



Entropy: Is a measure of uncertainty, purity and information content, it is expressed as follows

$$Entropy(S) = -p_+ \log_2(p_+) - p_- \log_2(p_-)$$

Information Gain: Expected reduction in entropy due to partitioning S on attribute A

$$Gain(S, A) = Entropy(S) - \sum_{v \epsilon values(A)} (|S_v|/|S|) \times Entropy(S_v)$$

In [ ]:
```
# Imports
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets, tree
from sklearn.metrics import (accuracy_score, f1_score,
                             recall_score, confusion_matrix,
                             precision_score, classification_report)
from sklearn.model_selection import train_test_split
```

In [ ]:
```
# Loading Dataset
iris_X, iris_y = datasets.load_iris(return_X_y = True)
iris_X, iris_y
```

In [ ]:
```
train_x, test_x, train_y, test_y = train_test_split(
    iris_X,
    iris_y,
    test_size=.3)
```

In [ ]:
```
model = tree.DecisionTreeClassifier()
model.fit(train_x, train_y)
```

Out[ ]:
```
▾ DecisionTreeClassifier

DecisionTreeClassifier()
```

In [ ]:
```
y_pred = model.predict(test_x)
y_pred_linear = y_pred
```

In [ ]:
```
print("Accuracy Score: ", accuracy_score(test_y, y_pred))
print("F1 Score: ", f1_score(test_y, y_pred, average="weighted"))
print("Recall Score: ", recall_score(test_y, y_pred, average="weighted"))
print("Precision Score: ", precision_score(test_y, y_pred, average="weighted"))
print("Confusion Matrix:-\n", confusion_matrix(test_y, y_pred))
```

```
Accuracy Score:  0.9555555555555556
F1 Score:  0.9555555555555556
Recall Score:  0.9555555555555556
Precision Score:  0.9555555555555556
Confusion Matrix:-
 [[18  0  0]
 [ 0 11  1]
 [ 0  1 14]]
```

In [ ]:
```
print(classification_report(test_y, y_pred))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        18
           1       0.92      0.92      0.92        12
           2       0.93      0.93      0.93        15

    accuracy                           0.96        45
   macro avg       0.95      0.95      0.95        45
weighted avg       0.96      0.96      0.96        45
```

Conclusion:

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

Experiment No.: 05

AIM:

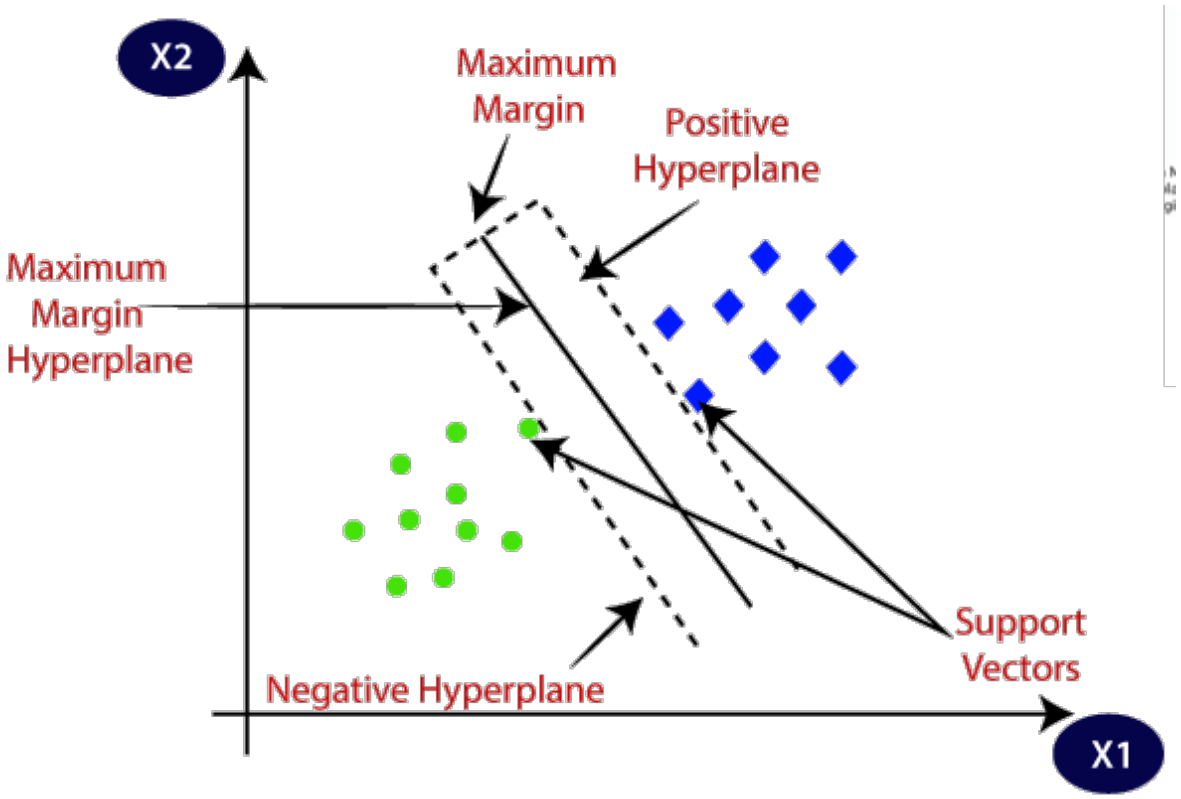
Program to implement Support Vector Machine (SVM) Classifier on iris dataset

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:



In [ ]:
```
# Imports
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets, svm
from sklearn.metrics import accuracy_score, f1_score,
                            recall_score, confusion_matrix,
                            precision_score
from sklearn.model_selection import train_test_split
```

In [ ]:
```
# Loading Dataset
iris_X, iris_y = datasets.load_iris(return_X_y = True)
iris_X, iris_y
```

In [ ]:
```
train_x, test_x, train_y, test_y = train_test_split(
    iris_X,
    iris_y,
    test_size=.3)
```

SVC with kernel='linear'

In [ ]:
```
model = svm.SVC(kernel='linear')
model.fit(train_x, train_y)
```

Out[ ]:
```
SVC
SVC(kernel='linear')
```

In [ ]:
```
y_pred = model.predict(test_x)
y_pred_linear = y_pred
```

In [ ]:
```
print("Accuracy Score: ", accuracy_score(test_y, y_pred))
print("F1 Score: ", f1_score(test_y, y_pred, average="weighted"))
print("Recall Score: ", recall_score(test_y, y_pred, average="weighted"))
print("Precision Score: ", precision_score(test_y, y_pred, average="weighted"))
print("Confusion Matrix:-\n", confusion_matrix(test_y, y_pred))
```

```
Accuracy Score:  0.9555555555555556
F1 Score:  0.9557834757834758
Recall Score:  0.9555555555555556
Precision Score:  0.9619047619047619
Confusion Matrix:-
 [[17  0  0]
 [ 0 12  0]
 [ 0  2 14]]
```

SVC with kernel='rbf'

In [ ]:
```
model = svm.SVC(kernel='rbf')
model.fit(train_x, train_y)
```

Out[ ]:
```
SVC
SVC()
```

In [ ]:
```
y_pred = model.predict(test_x)
y_pred_rbf = y_pred
```

In [ ]:
```
print("Accuracy Score: ", accuracy_score(test_y, y_pred))
print("F1 Score: ", f1_score(test_y, y_pred, average="weighted"))
print("Recall Score: ", recall_score(test_y, y_pred, average="weighted"))
print("Precision Score: ", precision_score(test_y,
    y_pred, average="weighted"))
print("Confusion Matrix:-\n", confusion_matrix(test_y, y_pred))
```

```
Accuracy Score:  0.9555555555555556
F1 Score:  0.9555555555555556
Recall Score:  0.9555555555555556
Precision Score:  0.9555555555555556
Confusion Matrix:-
 [[17  0  0]
 [ 0 11  1]
 [ 0  1 15]]
```

SVC with kernel='sigmoid'

In [ ]:
```
model = svm.SVC(kernel='sigmoid')
model.fit(train_x, train_y)
```

Out[ ]:
```
▾          SVC
SVC(kernel='sigmoid')
```

In [ ]:
```
y_pred = model.predict(test_x)
y_pred_sigmoid = y_pred
```

In [ ]:
```
print("Accuracy Score: ", accuracy_score(test_y, y_pred))
print("F1 Score: ", f1_score(test_y, y_pred, average="weighted"))
print("Recall Score: ", recall_score(test_y, y_pred, average="weighted"))
print("Precision Score: ", precision_score(test_y,
        y_pred, average="weighted", labels=np.unique(y_pred)))
print("Confusion Matrix:-\n", confusion_matrix(test_y, y_pred))
```

```
Accuracy Score:  0.26666666666666666
F1 Score:  0.11228070175438598
Recall Score:  0.26666666666666666
Precision Score:  0.26666666666666666
Confusion Matrix:-
 [[ 0 17  0]
 [ 0 12  0]
 [ 0 16  0]]
```

Conclusion:

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

Experiment No.: 06

AIM:

Program to implement Logistic Regression on a dataset

Description

Logistic regression is a statistical method used to model the probability of a binary outcome (such as yes/no or true/false) based on one or more predictor variables. The goal of the model is to fit a logistic function to the data, which maps the predictor variables to the probability of the binary outcome. Unlike linear regression, which is used to predict continuous outcomes, logistic regression is specifically designed for binary classification. The model is trained on a labeled dataset, where the outcome variable has two possible values, and the predictor variables may be continuous or categorical. Once the model is trained, it can be used to predict the probability of the outcome based on new input data. The logistic function itself is an S-shaped curve, which rises steeply at first and then flattens out as the predictor variable approaches the upper or lower bounds of its range. Overall, logistic regression is a useful tool for predicting binary outcomes and exploring the relationship between predictor variables and the probability of the outcome.

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

Where z can be a function. Eg. $f_{w,b}(x) = wx + b$ where, w is weight, b is bias, x is feature or input variable and $f_{w,b}(x)$ is predicted value.

Logistic Regression

In [ ]:
```
# Imports
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score, confusion_matrix, f1_score)
```

In [ ]:
```
# Loading Dataset
X_iris, y_iris = datasets.load_iris(return_X_y=True)
```

In [ ]:
```
# Creating Training and Test Sets
X_train, X_test, y_train, y_test = train_test_split(
    X_iris, y_iris, test_size=.3)
```

In [ ]:
```
# Creating Model
model = LogisticRegression()
model.fit(X_train, y_train)
```

Out[ ]:
```
▾ LogisticRegression

LogisticRegression()
```

In [ ]:
```
# Predicting
y_predict = model.predict(X_test)
y_predict.shape
```

Out[ ]:
```
(45,)
```

In [ ]:
```
# Different Scores for Model
print("Accuracy Score: ", accuracy_score(y_test, y_predict))
print("F1 Score: ", f1_score(y_test, y_predict, average="weighted"))
print("Recall Score: ", recall_score(y_test, y_predict, average="weighted"))
print("Precision Score: ", precision_score(
    y_test, y_predict, average="weighted"))
print("Confusion Matrix:-\n", confusion_matrix(y_test, y_predict))
```

```
Accuracy Score:  0.9111111111111111
F1 Score:  0.9108206245461148
Recall Score:  0.9111111111111111
Precision Score:  0.9288888888888889
Confusion Matrix:-
 [[10  0  0]
 [ 0 15  4]
 [ 0  0 16]]
```

Conclusion:

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

Experiment No.: 07

AIM:

Program to implement Naive Bayesian on a dataset

Description

Naive Bayesian is a classification algorithm based on the Bayes theorem, which predicts the likelihood of a class for given input features. It assumes that all the features are independent of each other, and the probability of one feature is not dependent on the probability of another feature. Hence, it is called naive, as it simplifies the assumption of the correlation between features to ease the calculations. It is widely used in many applications such as email spam detection, text classification, and sentiment analysis. Naive Bayesian is easy to implement and provides high accuracy with a small dataset, making it a popular choice for many machine learning problems.

$$P(c_i|x_1, x_2, \ldots, x_n) = \frac{P(c_i) \prod_{j=1}^{n} P(x_j|c_i)}{\sum_{k=1}^{K} P(c_k) \prod_{j=1}^{n} P(x_j|c_k)}$$

where c_i represents the i-th class of the data, $x_1, x_2, \ldots, x_n$ are the features or attributes of the data, $P(c_i)$ is the prior probability of class c_i, and $P(x_j|c_i)$ is the conditional probability of feature x_j given class c_i. The denominator is the normalization factor, ensuring that the sum of probabilities over all classes is equal to 1.

In [ ]:
```
# Imports
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score, confusion_matrix, f1_score)
```

In [ ]:
```
# Loading Dataset
X_iris, y_iris = datasets.load_iris(return_X_y=True)
```

In [ ]:
```
# Creating Training and Test Sets
X_train, X_test, y_train, y_test = train_test_split(
    X_iris, y_iris, test_size=.3)
```

In [ ]:
```
# Creating Model
model = GaussianNB()
model.fit(X_train, y_train)
```

Out[ ]:
```
▾ GaussianNB

GaussianNB()
```

In [ ]:
```
# Predicting
y_predict = model.predict(X_test)
y_predict.shape
```

Out[ ]:
```
(45,)
```

In [ ]:
```
# Different Scores for Model
print("Accuracy Score: ", accuracy_score(y_test, y_predict))
print("F1 Score: ", f1_score(y_test, y_predict, average="weighted"))
print("Recall Score: ", recall_score(y_test, y_predict, average="weighted"))
print("Precision Score: ", precision_score(
    y_test, y_predict, average="weighted"))
print("Confusion Matrix:-\n", confusion_matrix(y_test, y_predict))
```

```
Accuracy Score:  0.9111111111111111
F1 Score:  0.9111111111111111
Recall Score:  0.9111111111111111
Precision Score:  0.9111111111111111
Confusion Matrix:-
 [[17  0  0]
 [ 0 11  2]
 [ 0  2 13]]
```

Conclusion:

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

Experiment No.: 08

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 $\{(x_1, y_1), \ldots, (x_n, y_n)\}$, where each x_i is a feature vector and each y_i is its corresponding class label, and a new unlabeled sample x:

1. Compute the distance or similarity between x and each x_i 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 x, forming a set S.
3. Assign x the class label that is most frequent among the k nearest neighbors $(y_{i_1}, y_{i_2}, \ldots, y_{i_k}) \in S$.

Mathematically, this can be expressed as:

$$S = \{x_{i_1}, x_{i_2}, \ldots, x_{i_k}\},$$
$$y = \arg \max_{j \in \{1,\ldots,C\}} \sum_{l=1}^{k} [y_{i_l} = j],$$

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

In [ ]:
```
# Imports
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score, confusion_matrix, f1_score)
```

In [ ]:
```
# Loading Dataset
X_iris, y_iris = datasets.load_iris(return_X_y=True)
```

In [ ]:
```
# Creating Training and Test Sets
X_train, X_test, y_train, y_test = train_test_split(X_iris, y_iris, test_size=.3)
```

In [ ]:
```
# Creating Model
model = KNeighborsClassifier(n_neighbors=5)
model.fit(X_train, y_train)
```

Out[ ]:
```
▾ KNeighborsClassifier

KNeighborsClassifier()
```

In [ ]:
```
# Predicting
y_predict = model.predict(X_test)
y_predict.shape
```

Out[ ]:
```
(45,)
```

In [ ]:
```
# Different Scores for Model
print("Accuracy Score: ", accuracy_score(y_test, y_predict))
print("F1 Score: ", f1_score(y_test, y_predict, average="weighted"))
print("Recall Score: ", recall_score(y_test, y_predict, average="weighted"))
print("Precision Score: ", precision_score(y_test, y_predict, average="weighted"))
print("Confusion Matrix:-\n", confusion_matrix(y_test, y_predict))
```

```
Accuracy Score:  0.9555555555555556
F1 Score:  0.9550925925925925
Recall Score:  0.9555555555555556
Precision Score:  0.9607843137254902
Confusion Matrix:-
 [[17  0  0]
 [ 0 11  2]
 [ 0  0 15]]
```

Conclusion:

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

Experiment No.: 09

AIM:

Program to implement K-Means Clustering on a dataset

Description

K-means is a common clustering algorithm used in machine learning and data mining. It aims to divide a dataset into K clusters, where K is a predetermined number of clusters set by the user. Each cluster contains similar data points, based on their distance from the centroid of that cluster. The algorithm works as follows: it first initializes K centroids randomly, then iteratively assigns each data point to its closest centroid, and recalculates the centroid of each cluster based on the average of all the data points within that cluster. This process continues until no further changes to the centroids (and hence the clusters) occur or until max iterations are reached. The K-means algorithm is relatively simple to implement and computationally efficient, making it a popular choice for a variety of clustering tasks.

Step 1: Initialization

Choose k initial centroids $\{c_1, \ldots, c_k\}$

Step 2: Assignment

For each point x_i, assign it to the nearest centroid:
Let $S_j = \{x_p : ||x_p - c_j|| \leq ||x_p - c_l|| \text{ for all } l \neq j\}$
where $||\cdot||$ is the Euclidean distance

Step 3: Update Centroid

Update the centroids based on the points that they are now a

$$c_j = \frac{1}{|S_j|} \sum_{x_i \in S_j} x_i$$

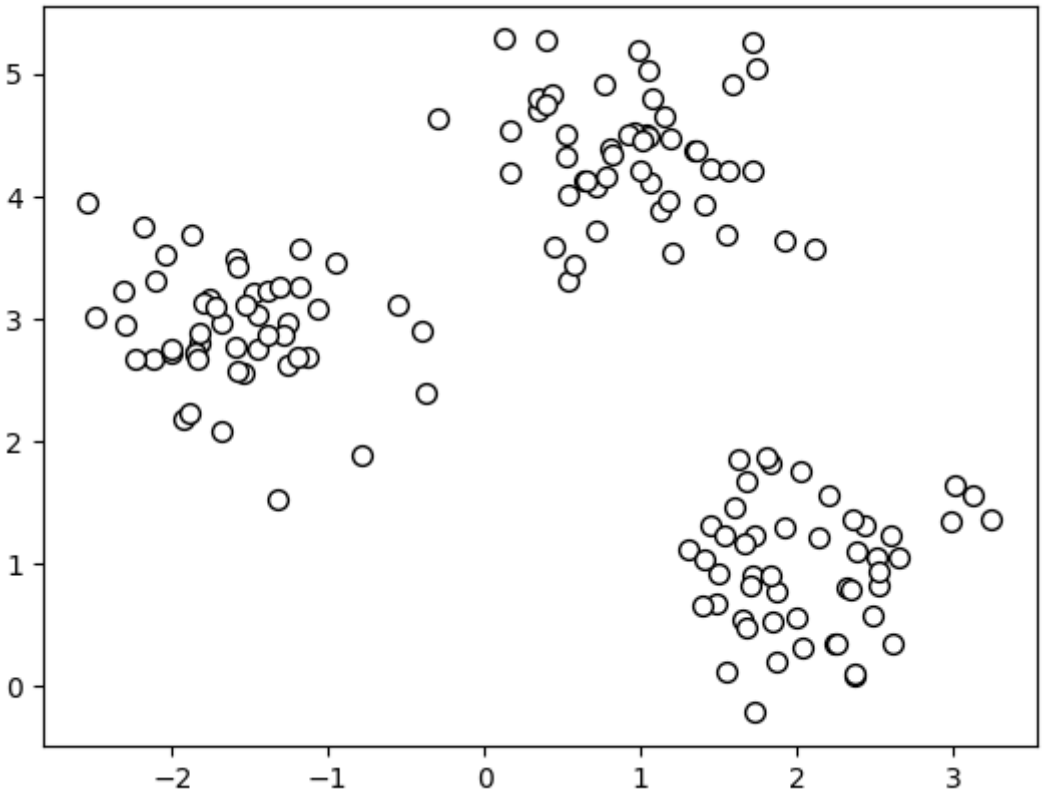
Step 4: Repeat Steps

Repeat steps 2 and 3 until convergence.

In [ ]:
```python
# Imports
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans
```

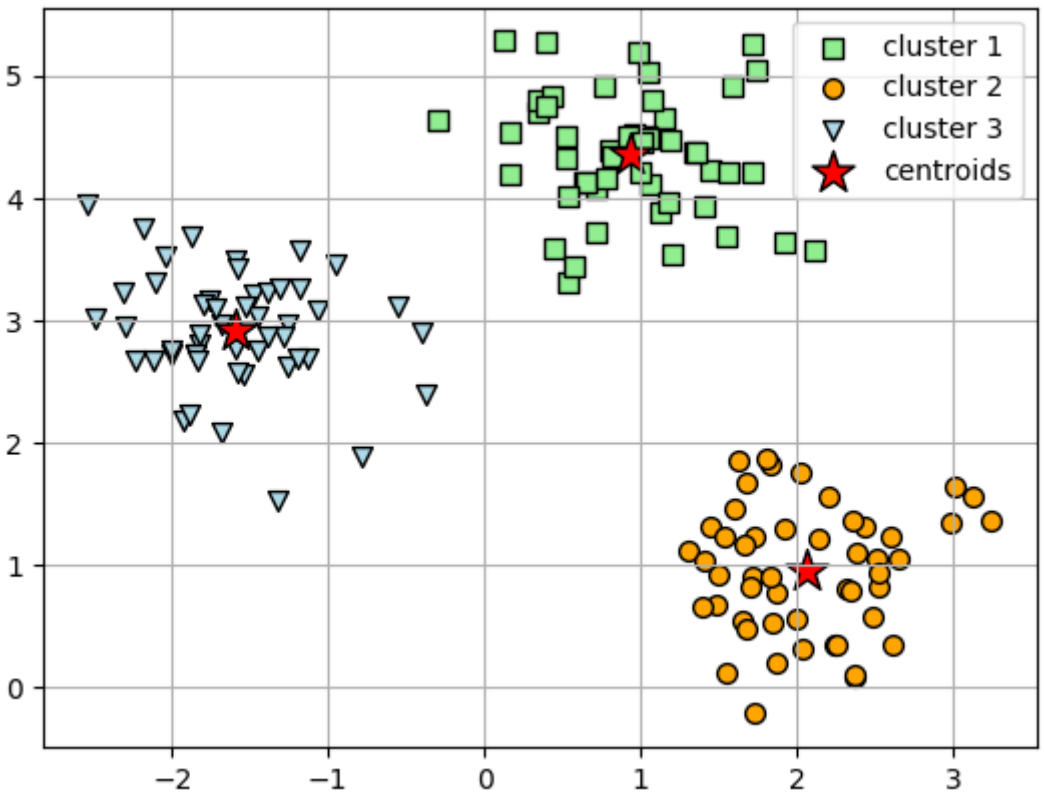
In [ ]:
```python
# Dataset
X, y = make_blobs(
    n_samples=150, n_features=2,
    centers=3, cluster_std=.5,
    shuffle=True, random_state=0
)
```

In [ ]:
```
# Plots for Dataset
plt.scatter(
   X[:, 0], X[:, 1],
   c='white', marker='o',
   edgecolor='black', s=50
)
plt.show()
```



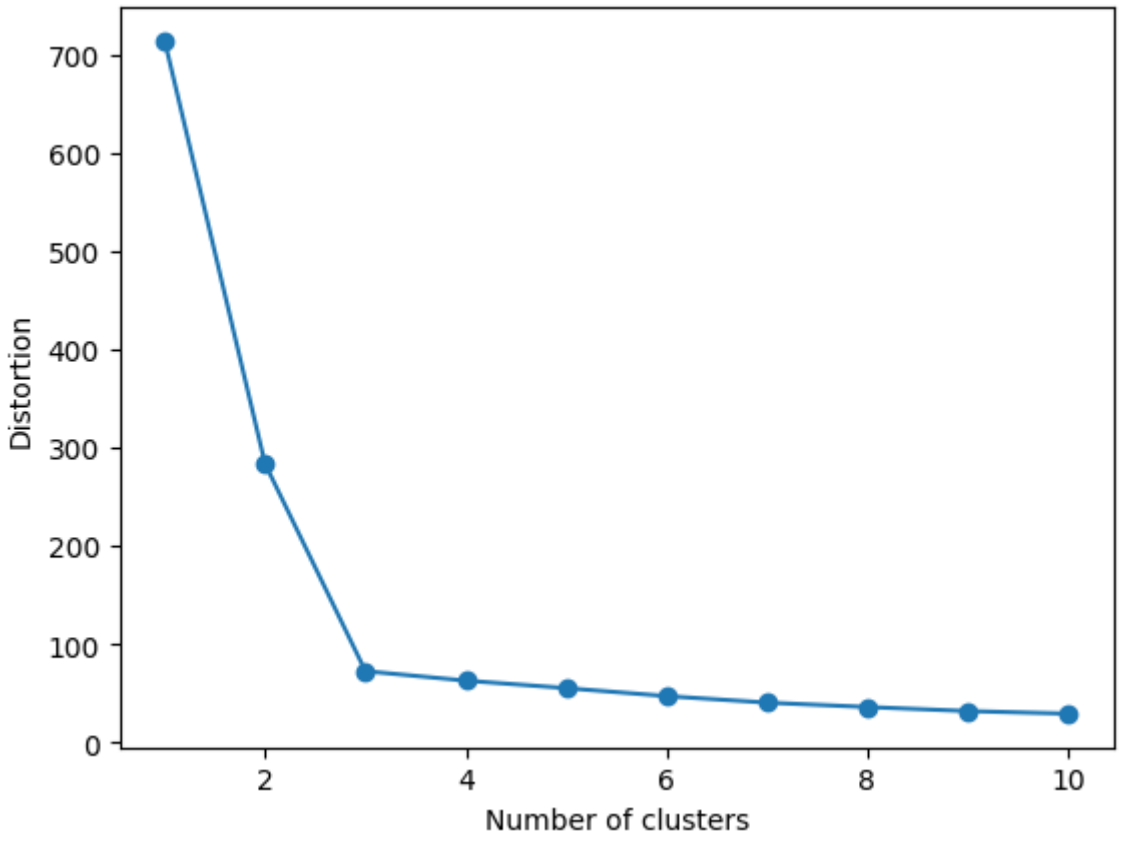
In [ ]:
```
# Traning Model
m = KMeans(
    n_clusters=3, init='random',
    n_init=10, max_iter=300,
    tol=1e-04, random_state=0
)
y_m = m.fit_predict(X)
```

In [ ]:
```
# Plots with centroids
plt.scatter(
    X[y_m == 0, 0], X[y_m == 0, 1],
    s=50, c='lightgreen',
    marker='s', edgecolor='black',
    label='cluster 1'
)

plt.scatter(
    X[y_m == 1, 0], X[y_m == 1, 1],
    s=50, c='orange',
    marker='o', edgecolor='black',
    label='cluster 2'
)

plt.scatter(
    X[y_m == 2, 0], X[y_m == 2, 1],
    s=50, c='lightblue',
    marker='v', edgecolor='black',
    label='cluster 3'
)

# plot the centroids
plt.scatter(
    m.cluster_centers_[:, 0], m.cluster_centers_[:, 1],
    s=250, marker='*',
    c='red', edgecolor='black',
    label='centroids'
)
plt.legend(scatterpoints=1)
plt.grid()
plt.show()
```



In [ ]:
```
# calculate distortion for a range of number of cluster
distortions = []
for i in range(1, 11):
    m = KMeans(
        n_clusters=i, init='random',
        n_init=10, max_iter=300,
        tol=1e-04, random_state=0
    )
    m.fit(X)
    distortions.append(m.inertia_)
```

In [ ]:
```
# Plot
plt.plot(range(1, 11), distortions, marker='o')
plt.xlabel('Number of clusters')
plt.ylabel('Distortion')
plt.show()
```



Conclusion:

Hence, The K-Means Clustering has been implemented for Blobs dataset, different cluster were identified successfully

Experiment No.: 10

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:

$$
\begin{aligned}
&\textbf{Input: } \{x_1, x_2, \ldots, x_n\} \\
&\textbf{Output: } \text{Dendrogram of clusters} \\
&\textbf{Step 1: } \text{Initialize the clusters as } C = \{\{x_1\}, \{x_2\}, \ldots, \{x_n\}\} \\
&\textbf{Step 2: } \text{Compute the pairwise distances between all clusters} \\
&\quad \text{using a suitable distance metric } d(c_i, c_j) \\
&\textbf{Step 3: } \text{Find the two closest clusters } c_{i^*}, c_{j^*} \in C \\
&\quad \text{according to the distance metric} \\
&\textbf{Step 4: } \text{Merge the two closest clusters into a single cluster } c_{i^*} \cup c_{j^*} \\
&\textbf{Step 5: } \text{Update the set of clusters } C = C \setminus \{c_{i^*}, c_{j^*}\} \cup \{c_k\} \\
&\quad \text{where } c_k = c_{i^*} \cup c_{j^*} \\
&\textbf{Step 6: } \text{Repeat steps 2-5 until all points are in a single cluster}
\end{aligned}
$$

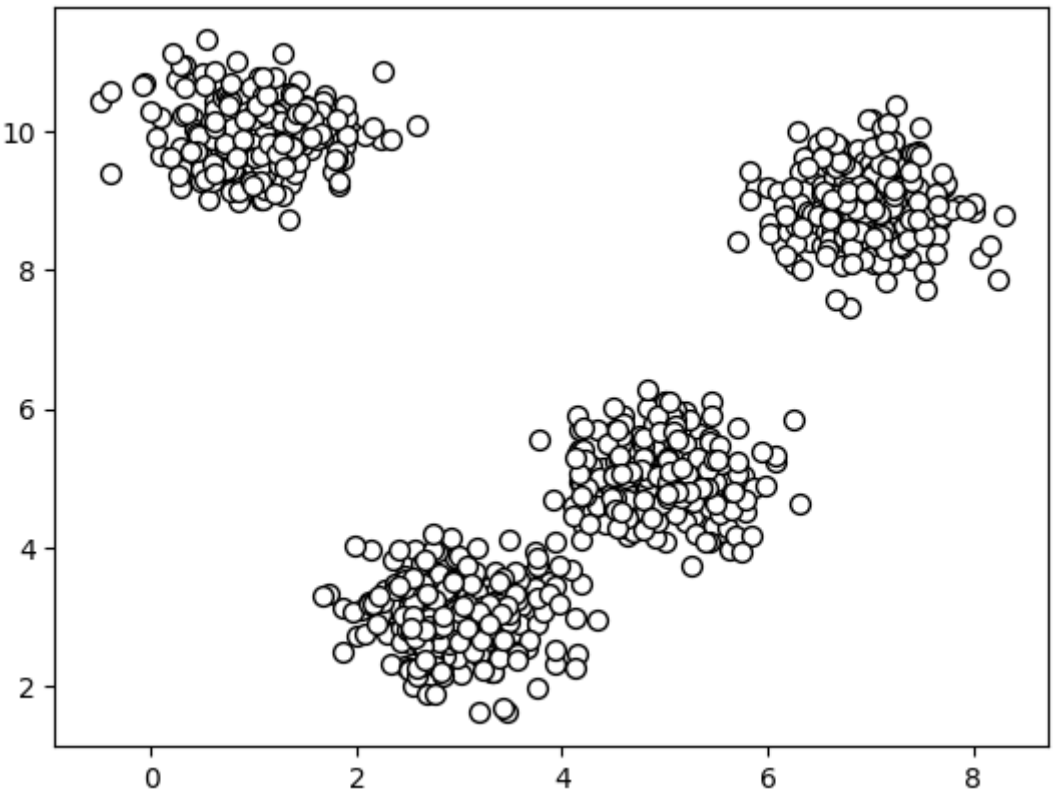
After applying this algorithm, we obtain a dendrogram of clusters that represents the hierarchical structure of the data.

In [ ]:
```
# Imports
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
```

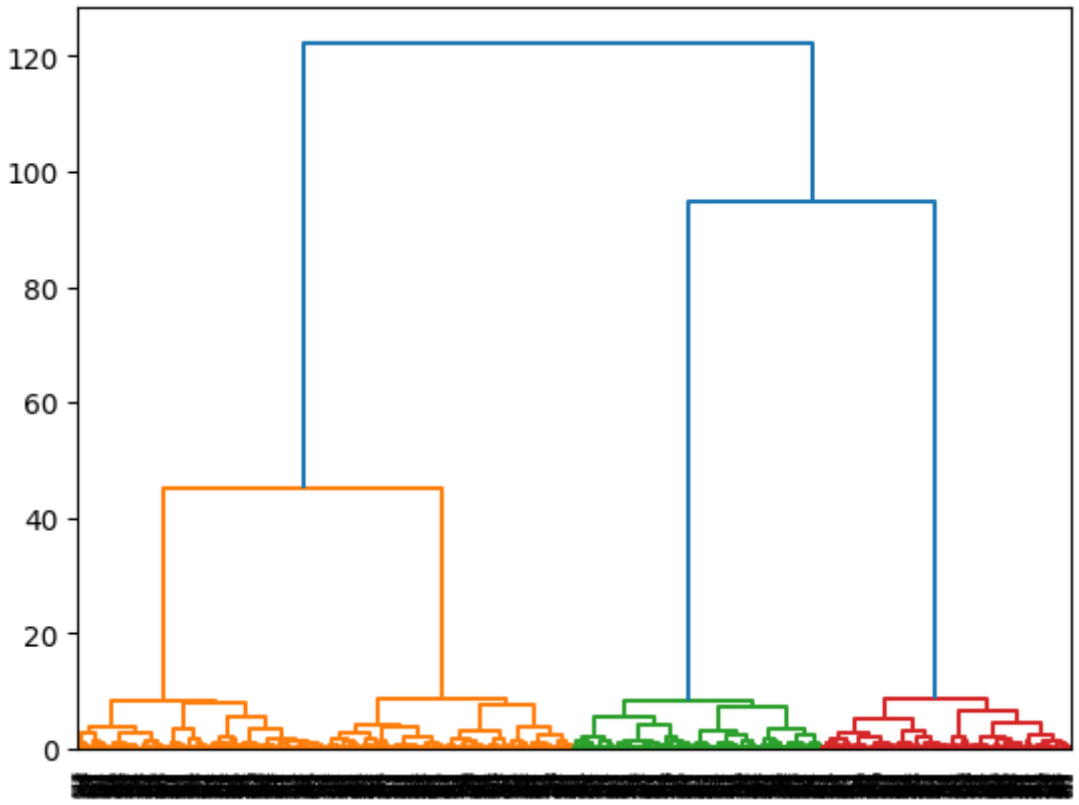
In [ ]:
```
# Configuration options
num_samples_total = 1000
cluster_centers = [(3,3), (7,9), (1, 10), (5, 5)]
num_classes = len(cluster_centers)
epsilon = 1.0
min_samples = 13
```

In [ ]:
```
# Dataset
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),
)
```

In [ ]:
```
# Plots for Dataset
plt.scatter(
   X[:, 0], X[:, 1],
   c='white', marker='o',
   edgecolor='black', s=50
)
plt.show()
```



In [ ]:
```
dendrogram = sch.dendrogram(sch.linkage(X, method='ward'))
```

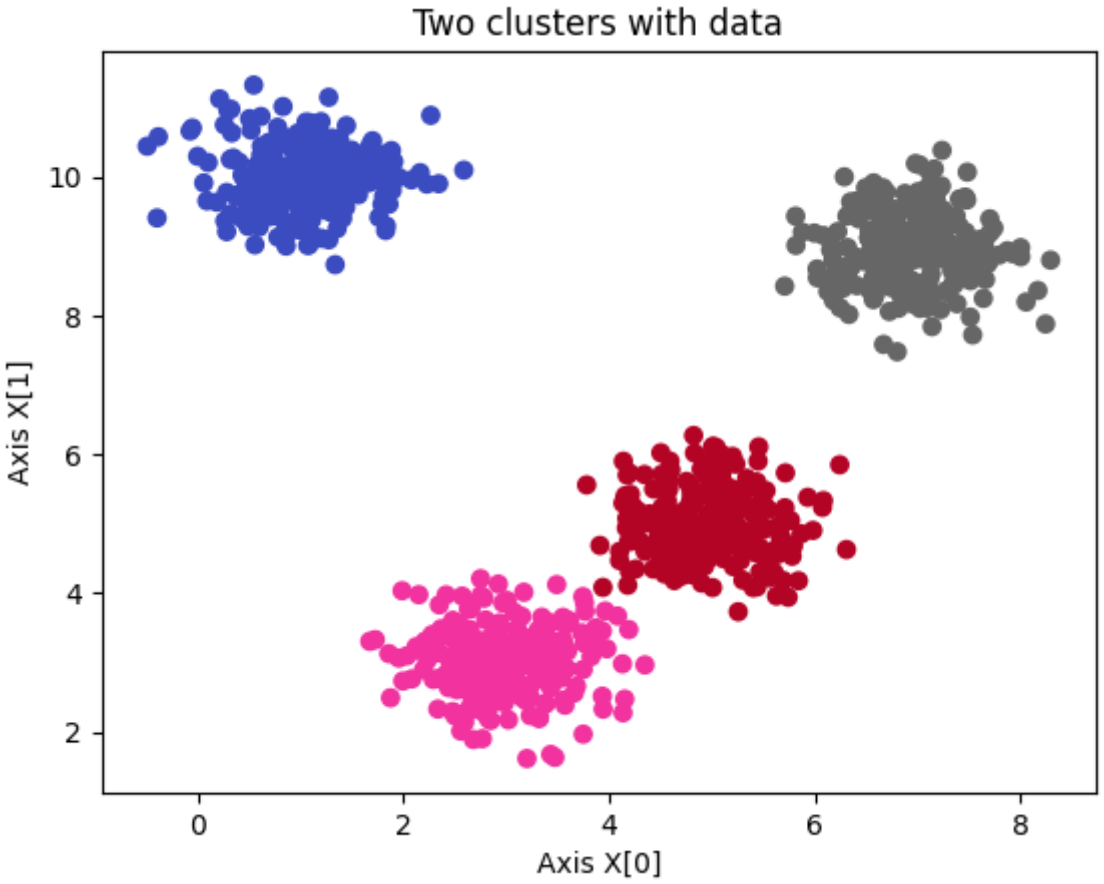


In [ ]:
```
# Traning Model
m = AgglomerativeClustering(
    n_clusters=num_classes,
    metric='euclidean',
    linkage='ward',
)
y_m = m.fit(X)
labels = y_m.labels_
```

In [ ]:
```
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)
```

```
Estimated no. of clusters: 4
Estimated no. of noise points: 0
```

In [ ]:
```
# Generate scatter plot for training data
colors = list(map(lambda x: ['#3b4cc0', '#b40426', '#666666', '#f1329f'][x], label
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()
```



Conclusion:

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

Experiment No.: 11

AIM:

Program to implement DBSCAN Clustering on a dataset

Description

DBSCAN (Density-Based Spatial Clustering of Applications with Noise) is a density-based clustering algorithm that groups together data points that are closely packed together and separates the outliers or noise points. The algorithm defines a cluster as a dense region of points and identifies points in low-density regions as noise points. DBSCAN considers two important parameters: Eps, which defines the radius of the neighborhood around a data point, and MinPts, the minimum number of points that must be present within a cluster. It classifies points into three categories: core points, which have at least MinPts in their neighborhood; border points, which are within Eps distance of a core point, but have less than MinPts in their neighborhood; and noise points, which are not part of any cluster. The algorithm uses connected component analysis to form clusters by grouping together core points that are reachable from each other. DBSCAN is widely used in data mining, machine learning, and outlier detection.

The DBSCAN clustering algorithm:

Let X be a dataset with n observations, and let ϵ and minPts be user-defined parameters.

Begin by randomly selecting an unvisited observation, x in X.

If there are fewer than minPts observations within a distance of ϵ from x, mark x as noise.

Otherwise, mark x as a new cluster center and add it to the current cluster.

For each observation y in the ϵ-neighborhood of x, if y is unvisited, mark it as visited and add it to the current cluster.

If y is already a cluster center, merge the new cluster with the existing cluster.

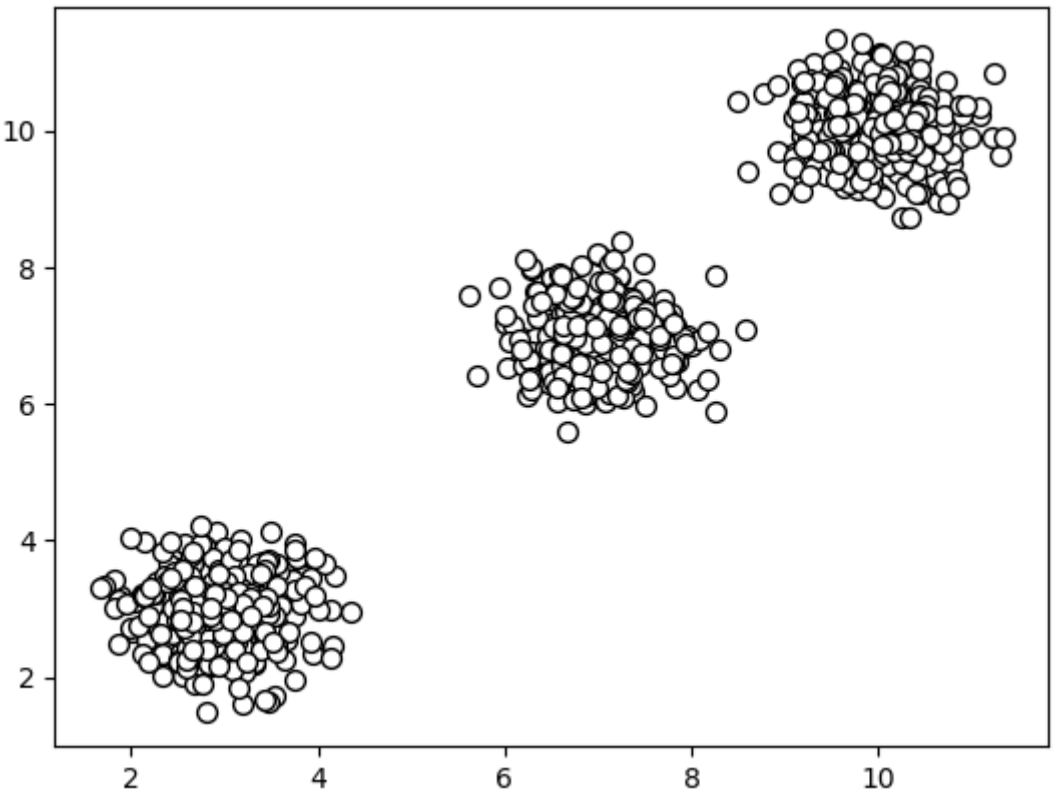
Repeat this process until all observations in X have been visited.

In [ ]:
```
# Imports
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from sklearn.cluster import DBSCAN
```

In [ ]:
```
# Configuration options
num_samples_total = 1000
cluster_centers = [(3, 3), (7, 7), (10, 10)]
num_classes = len(cluster_centers)
epsilon = 1.0
min_samples = 13
```

In [ ]:
```
# Dataset
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),
)
```

In [ ]:
```
# Plots for Dataset
plt.scatter(
   X[:, 0], X[:, 1],
   c='white', marker='o',
   edgecolor='black', s=50
)
plt.show()
```

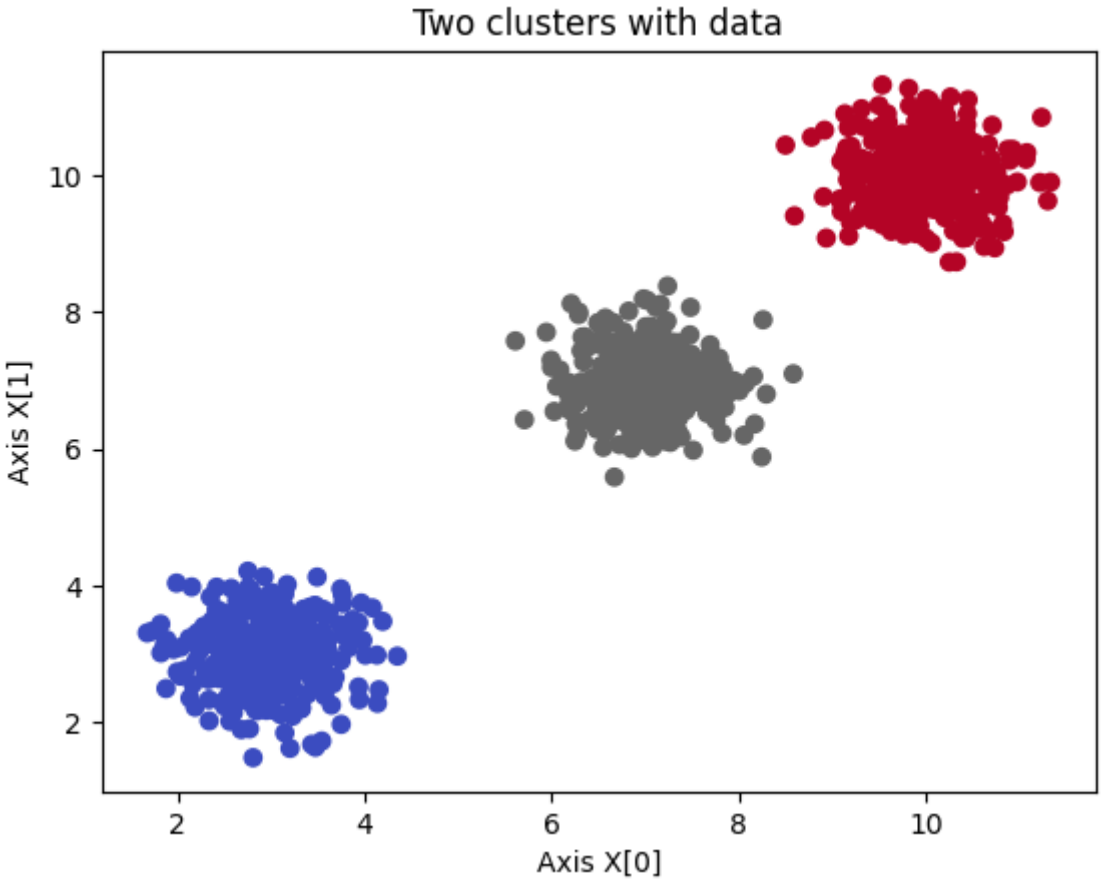


In [ ]:
```
# Traning Model
m = DBSCAN(
    eps=epsilon,
    min_samples=min_samples
)
y_m = m.fit(X)
labels = y_m.labels_
```

In [ ]:
```
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)
```

```
Estimated no. of clusters: 3
Estimated no. of noise points: 0
```

In [ ]:
```
# Generate scatter plot for training data
colors = list(map(lambda x: ['#3b4cc0', '#b40426', '#666666'][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()
```



Conclusion:

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

Experiment No.: 12

AIM:

Program to implement Gradient Descent Algorithm using tensorflow

In [ ]:
```
# Imports
import numpy as np
import tensorflow as tf
```

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 $theta_j$ is the rate of change of $J(theta)$ with respect to $theta_j$, or how much $J(theta)$ changes for a small change in $theta_j$. 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.

In [ ]:
```
# Generate some random data
x_train = np.random.rand(100, 1) * 10
y_train = 2 * x_train - 3 + np.random.randn(100, 1)
```

In [ ]:
```
# Define the model
model = tf.keras.Sequential([
    tf.keras.layers.Dense(units=1, input_shape=[1])
])
```

In [ ]:
```
# Define the loss function (mean squared error)
loss_fn = tf.keras.losses.MeanSquaredError()
```

In [ ]:
```
# Define the optimizer (gradient descent)
optimizer = tf.keras.optimizers.SGD(learning_rate=0.01)
```

In [ ]:
```
# Define the training dataset
train_dataset = tf.data.Dataset.from_tensor_slices(
        (x_train, y_train)).batch(32)
```

In [ ]:
```
# Train the model
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]))
```

```
Epoch 0: w = [[2.0318627]], b = [-2.8429263]
Epoch 100: w = [[2.0318627]], b = [-2.8429263]
Epoch 200: w = [[2.0318627]], b = [-2.8429263]
Epoch 300: w = [[2.0318627]], b = [-2.8429263]
Epoch 400: w = [[2.0318627]], b = [-2.8429263]
Epoch 500: w = [[2.0318627]], b = [-2.8429263]
Epoch 600: w = [[2.0318627]], b = [-2.8429263]
Epoch 700: w = [[2.0318627]], b = [-2.8429263]
Epoch 800: w = [[2.0318627]], b = [-2.8429263]
Epoch 900: w = [[2.0318627]], b = [-2.8429263]
```

In [ ]:
```
model.get_weights()
```

Out[ ]:
```
[array([[2.0318627]], dtype=float32), array([-2.8429263], dtype=float32)]
```

Conclusion:

Hence, The Gradient Descent Algorithm using Tensorflow has been implemented successfully