Experiment 1: Data Visualisation

AIM: Using the inbuilt Python modules to plot and visualise the specified dataset.

DESCRIPTION:

- `matplotlib.pyplot` is a plotting library for Python that provides a variety of tools for creating visualizations such as line plots, scatter plots, histograms, and more. It is widely used in data analysis and scientific research. The %matplotlib inline command is a magic command used in Jupyter notebooks to display the plots within the notebook itself.
- `numpy` is a Python library for numerical computations. It provides tools for working with arrays, matrices, and other numerical data structures, as well as a variety of mathematical functions for manipulating and analyzing the data. It is a core library in the scientific Python ecosystem and is used in many data analysis and machine learning tasks.
- `pandas` is a Python library for data manipulation and analysis. It provides tools for working with tabular data such as spreadsheets and databases, as well as tools for handling missing data and performing data aggregation and transformation. It is widely used in data analysis and machine learning tasks, particularly in areas such as finance, economics, and social sciences.
- `seaborn` is a Python data visualization library based on matplotlib. It provides a high-level interface for creating informative and attractive statistical graphics, including heatmaps, scatterplots, lineplots, barplots, and more.

In [1]:
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
%matplotlib inline
```

In [ ]:
```
! pip install seaborn
import seaborn as sns
```

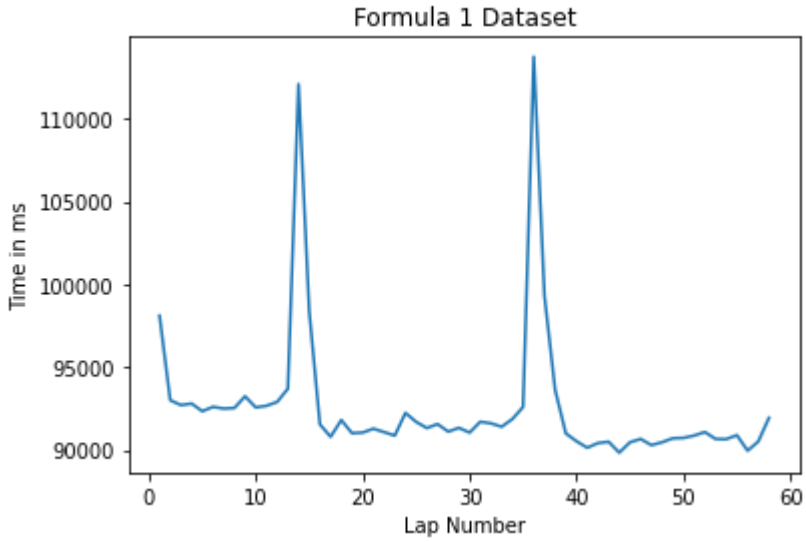
In [5]:
```
df = pd.read_csv("lap_times.csv")
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 528785 entries, 0 to 528784
Data columns (total 6 columns):
 #   Column        Non-Null Count   Dtype 
---  ------        --------------   ----- 
 0   raceId        528785 non-null  int64 
 1   driverId      528785 non-null  int64 
 2   lap           528785 non-null  int64 
 3   position      528785 non-null  int64 
 4   time          528785 non-null  object
 5   milliseconds  528785 non-null  int64 
dtypes: int64(5), object(1)
memory usage: 24.2+ MB
```

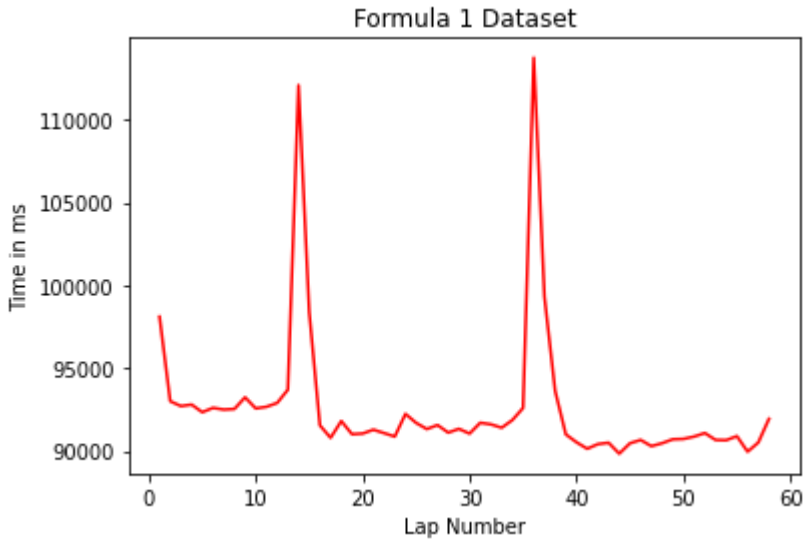
`plt.plot` is a function in the matplotlib.pyplot module that is used to create a line plot or a scatter plot. The basic syntax for plt.plot is as follows:

```
plt.plot(x, y, format_string, **kwargs)
```

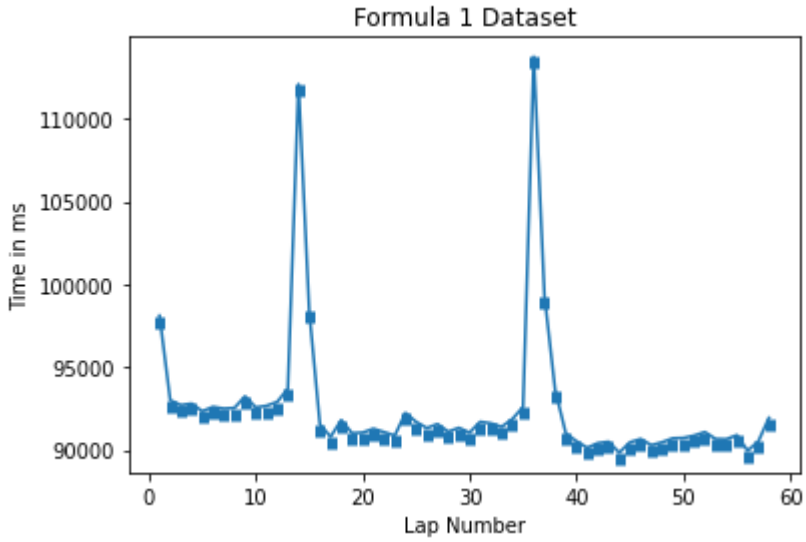
In [6]:
```
plt.plot(df['lap'][0:58], df['milliseconds'][0:58])
plt.title("Formula 1 Dataset")
plt.xlabel("Lap Number")
plt.ylabel("Time in ms")
plt.show()
```



In [7]:
```
plt.plot(df['lap'][0:58], df['milliseconds'][0:58], color = 'red')
plt.title("Formula 1 Dataset")
plt.xlabel("Lap Number")
plt.ylabel("Time in ms")
plt.show()
```

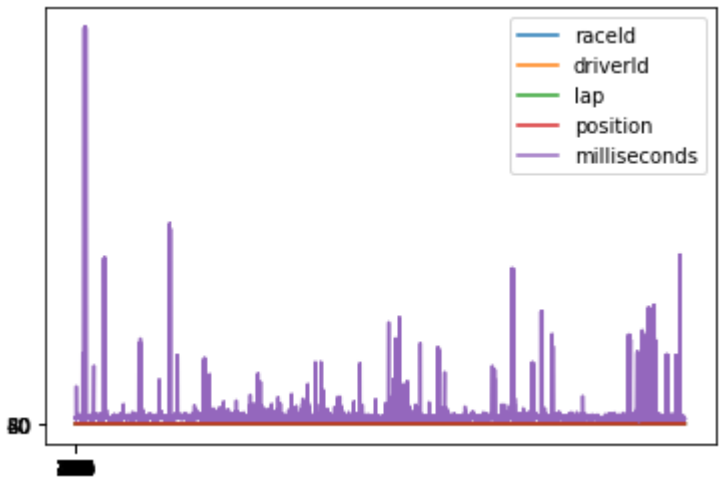


In [8]:
```
plt.plot(df['lap'][0:58], df['milliseconds'][0:58], marker = 3, mew = 5)
plt.title("Formula 1 Dataset")
plt.xlabel("Lap Number")
plt.ylabel("Time in ms")
plt.show()
```



In [9]:
```
df.plot(xticks=range(1,500),yticks=range(0,100,20))
```

Out[9]: <AxesSubplot: >

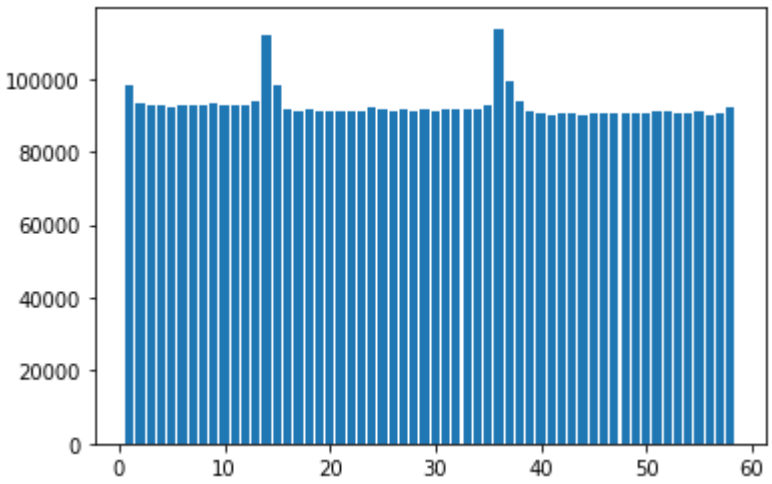


`plt.bar` is a function in the matplotlib.pyplot module that is used to create bar plots. A bar plot is a visualization that represents the data as rectangular bars, with the height or length of each bar proportional to the value being plotted.

```
plt.bar(x, height, width, bottom, align, color, **kwargs)
```

In [ ]:
```
plt.bar(df['lap'][0:58], df['milliseconds'][0:58])
```

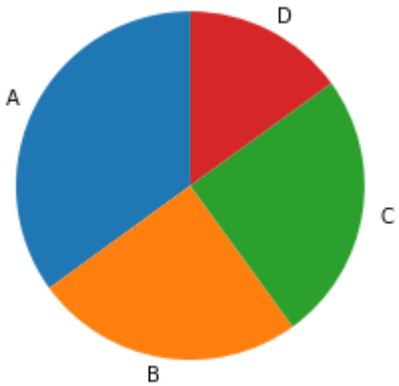
Out[13]:
```
<BarContainer object of 58 artists>
```



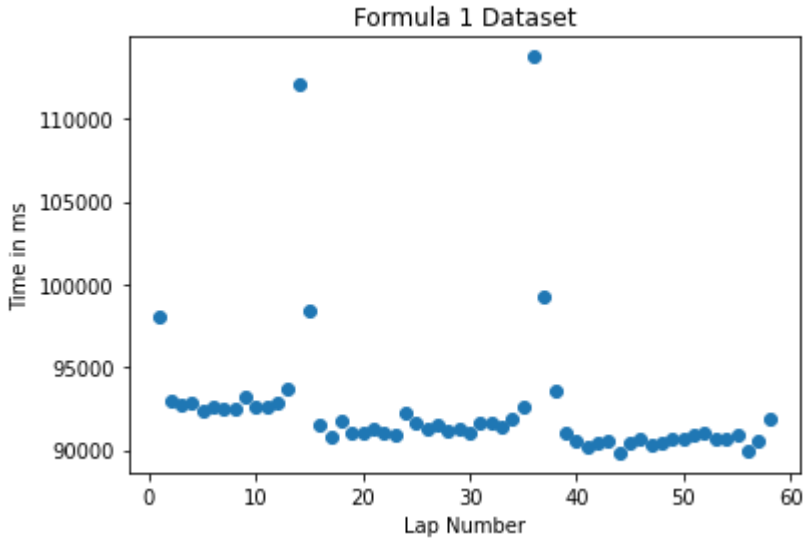
`plt.pie` is a function in the matplotlib.pyplot module that is used to create pie charts. A pie chart is a circular visualization that represents the data as slices of a pie, with the size or angle of each slice proportional to the value being plotted.

In [10]:
```
y = np.array([35, 25, 25, 15])
mylabels = ["A", "B", "C", "D"]
plt.pie(y, labels = mylabels, startangle = 90)
```

Out[10]:
```
([<matplotlib.patches.Wedge at 0x2df55e0aec0>,
  <matplotlib.patches.Wedge at 0x2df55e0b160>,
  <matplotlib.patches.Wedge at 0x2df55e0b640>,
  <matplotlib.patches.Wedge at 0x2df55e0bac0>],
 [Text(-0.9801071672559597, 0.49938956806635293, 'A'),
  Text(-0.172077952232839, -1.086457168210212, 'B'),
  Text(1.086457168210212, -0.17207795223283906, 'C'),
  Text(0.49938956806635265, 0.9801071672559598, 'D')])
```



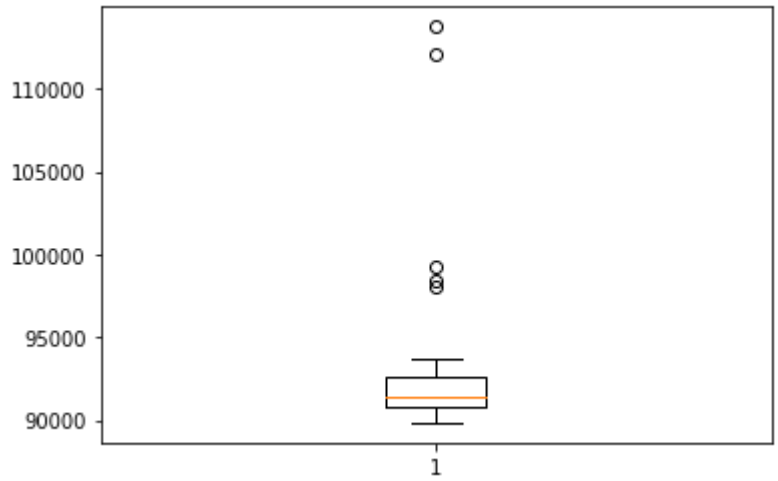
In [ ]:
```
plt.scatter(df['lap'][0:58], df['milliseconds'][0:58])
plt.title("Formula 1 Dataset")
plt.xlabel("Lap Number")
plt.ylabel("Time in ms")
plt.show()
```



`plt.boxplot` is a function in the matplotlib.pyplot module that is used to create box plots. A box plot is a visualization that represents the data using five statistics: the minimum value, the first quartile (Q1), the median, the third quartile (Q3), and the maximum value.

In [ ]:
```
plt.boxplot(df['milliseconds'][0:58])
```

Out[31]:
```
{'whiskers': [<matplotlib.lines.Line2D at 0x7f25a24489d0>,
  <matplotlib.lines.Line2D at 0x7f25a2448d30>],
 'caps': [<matplotlib.lines.Line2D at 0x7f25a24520d0>,
  <matplotlib.lines.Line2D at 0x7f25a2452430>],
 'boxes': [<matplotlib.lines.Line2D at 0x7f25a2448670>],
 'medians': [<matplotlib.lines.Line2D at 0x7f25a2452790>],
 'fliers': [<matplotlib.lines.Line2D at 0x7f25a2452a90>],
 'means': []}
```

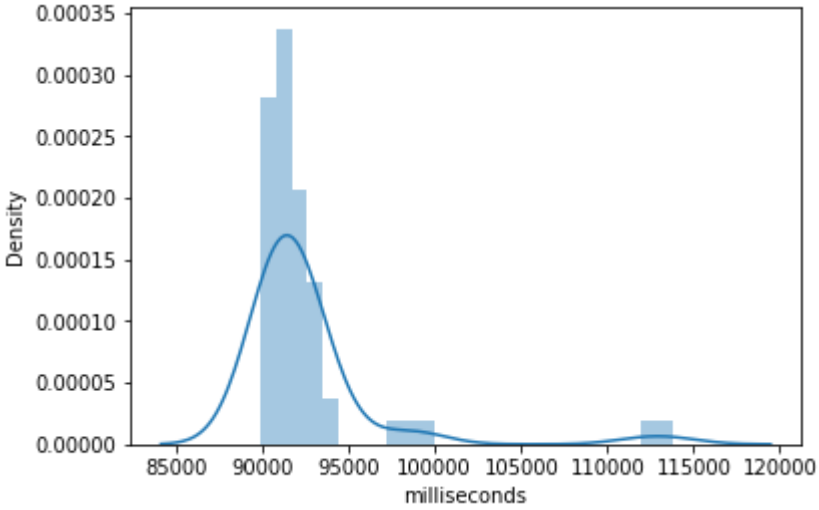


`sns.distplot` is a function in the seaborn module that is used to create a histogram with a density curve. A histogram is a visualization that represents the distribution of a dataset by dividing it into a set of contiguous bins and counting the number of values that fall into each bin. A density curve is a smooth function that represents the distribution of the data in a continuous form.

In [ ]:
```
sns.distplot(df['milliseconds'][0:58])
```

```
/usr/local/lib/python3.8/dist-packages/seaborn/distributions.py:2619: FutureWarnin
g: `distplot` is a deprecated function and will be removed in a future version. Ple
ase adapt your code to use either `displot` (a figure-level function with similar f
lexibility) or `histplot` (an axes-level function for histograms).
  warnings.warn(msg, FutureWarning)
```

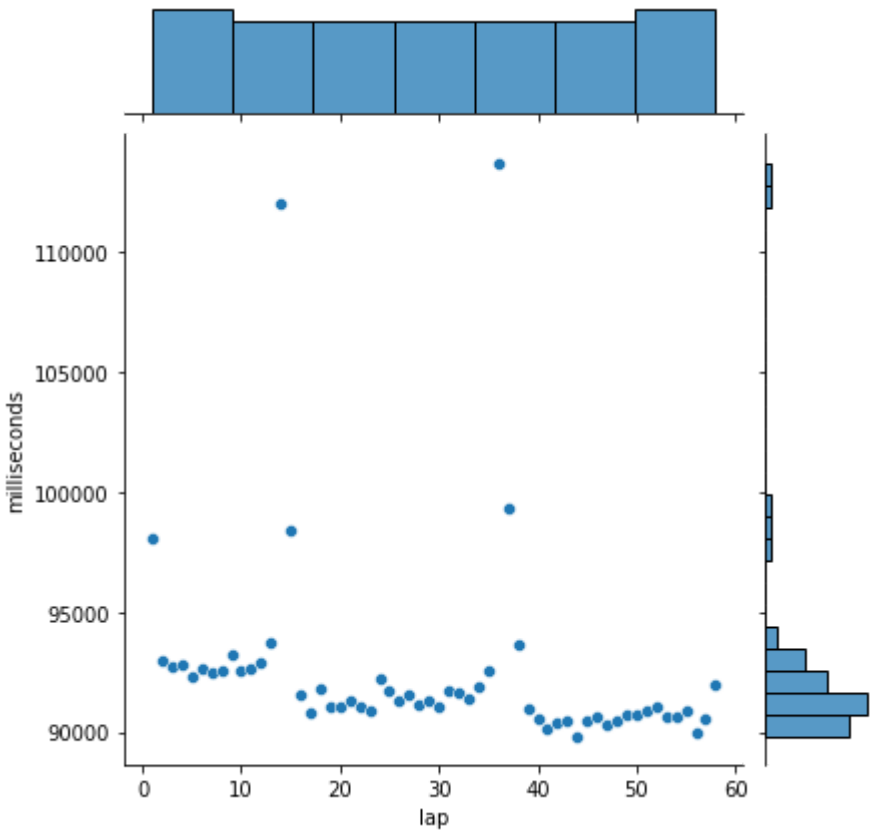
Out[34]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f2596014d30>
```



`sns.jointplot` is a function in the seaborn module that is used to create a joint plot, which combines two different visualizations: a scatter plot and a histogram. A scatter plot is a visualization that represents the relationship between two variables by plotting their values as points in a two-dimensional coordinate system. A histogram is a visualization that represents the distribution of a variable by dividing it into a set of contiguous bins and counting the number of values that fall into each bin.

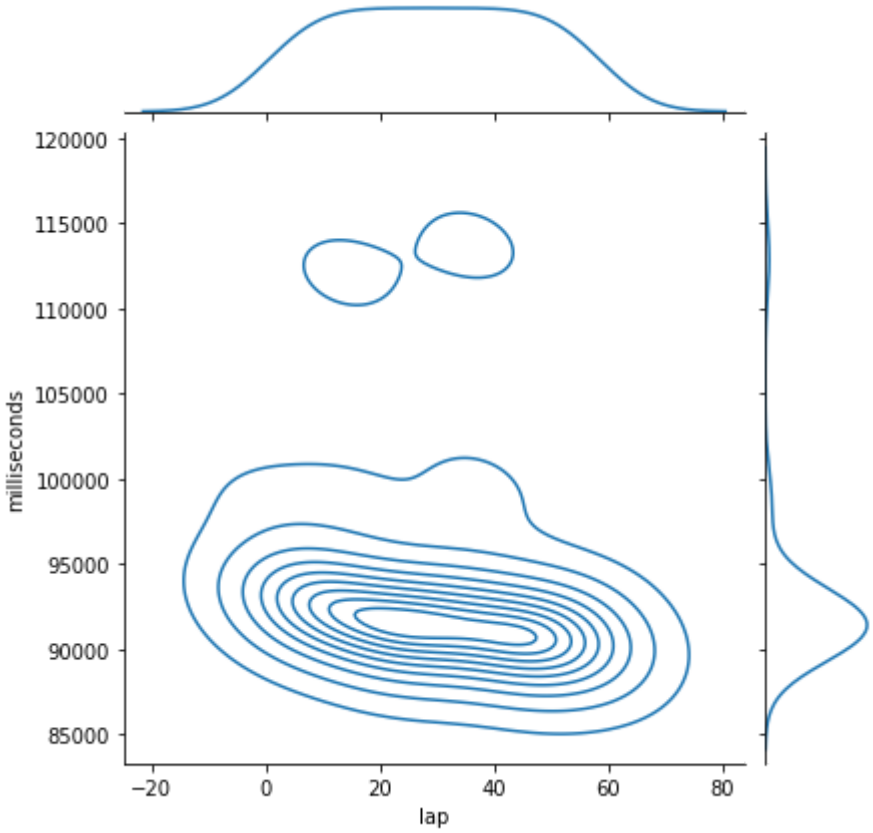
In [ ]:
```
sns.jointplot(x=df['lap'][:58], y=df['milliseconds'][:58])
```

Out[35]:
```
<seaborn.axisgrid.JointGrid at 0x7f25a24bd100>
```



In [ ]:
```
sns.jointplot(x=df['lap'][:58], y=df['milliseconds'][:58], kind = "kde")
```

Out[36]:
```
<seaborn.axisgrid.JointGrid at 0x7f2593f223d0>
```



CONCLUSION:

Using Python modules such as numpy, pandas, matplotlib and seaborn the correseponding dataset is retrieved, cleaned and visualised.

Viva Questions for Data Visualisation

1. What is data visualization, and why is it important in machine learning?
2. How do you install and import the required libraries for data visualisation in Python?
3. What is the purpose of the pandas library in data visualisation, and how is it used?
4. How do you create a scatter plot using matplotlib?
5. What is the purpose of the numpy library in data visualisation, and how is it used?
6. How do you create a bar plot using seaborn?
7. How do you customize the appearance of a plot using matplotlib?
8. How do you create a histogram using pandas?
9. How do you create a box plot using seaborn?
10. How do you compare multiple plots in a single figure using matplotlib?

Experiment 2: Linear Regression

AIM: To implement Linear Regression using the sci-kit learn library

DESCRIPTION:

Linear regression is a statistical technique used to model the relationship between a dependent variable and one or more independent variables. It assumes a linear relationship between the dependent variable and the independent variable(s), and seeks to find the best fit line (or hyperplane in higher dimensions) that explains the relationship between the variables.

In simple linear regression, there is only one independent variable and the relationship is modeled with a straight line. The equation for the line can be expressed as:

$$\hat{Y} = \hat{\beta}_0 + X_j\hat{\beta}_j$$

where y is the dependent variable, x is the independent variable, m is the slope of the line, and b is the intercept.

In multiple linear regression, there are multiple independent variables and the relationship is modeled with a hyperplane. The equation for the hyperplane can be expressed as:

$$\hat{Y} = \hat{\beta}_0 + \sum_{j=1}^{p} X_j\hat{\beta}_j$$

The goal of linear regression is to find the values of the coefficients that minimize the difference between the predicted values and the actual values of the dependent variable. This is typically done using a method called least squares regression, which minimizes the sum of the squared differences between the predicted and actual values.

$$error = (Y - \hat{Y}_i)$$

`Scikit-learn (sklearn)` is a free and open-source Python library used for machine learning and data analysis. It is built on top of NumPy, SciPy, and matplotlib, and provides a wide range of tools for machine learning tasks such as classification, regression, clustering, dimensionality reduction, model selection, and data preprocessing.

In [1]:
```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_absolute_error, mean_squared_error
```

In [2]:
```
df = pd.read_csv("/content/lap_times.csv")
df.pop("time")
df.dropna()
print("Dataset Information")
df.info()
print("Dataset Values")
df.head()
```

```
Dataset Information
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 528785 entries, 0 to 528784
Data columns (total 5 columns):
 #   Column        Non-Null Count   Dtype
---  ------        --------------   -----
 0   raceId        528785 non-null  int64
 1   driverId      528785 non-null  int64
 2   lap           528785 non-null  int64
 3   position      528785 non-null  int64
 4   milliseconds  528785 non-null  int64
dtypes: int64(5)
memory usage: 20.2 MB
Dataset Values
```

Out[2]:

	raceId	driverId	lap	position	milliseconds
0	841	20	1	1	98109
1	841	20	2	1	93006
2	841	20	3	1	92713
3	841	20	4	1	92803
4	841	20	5	1	92342

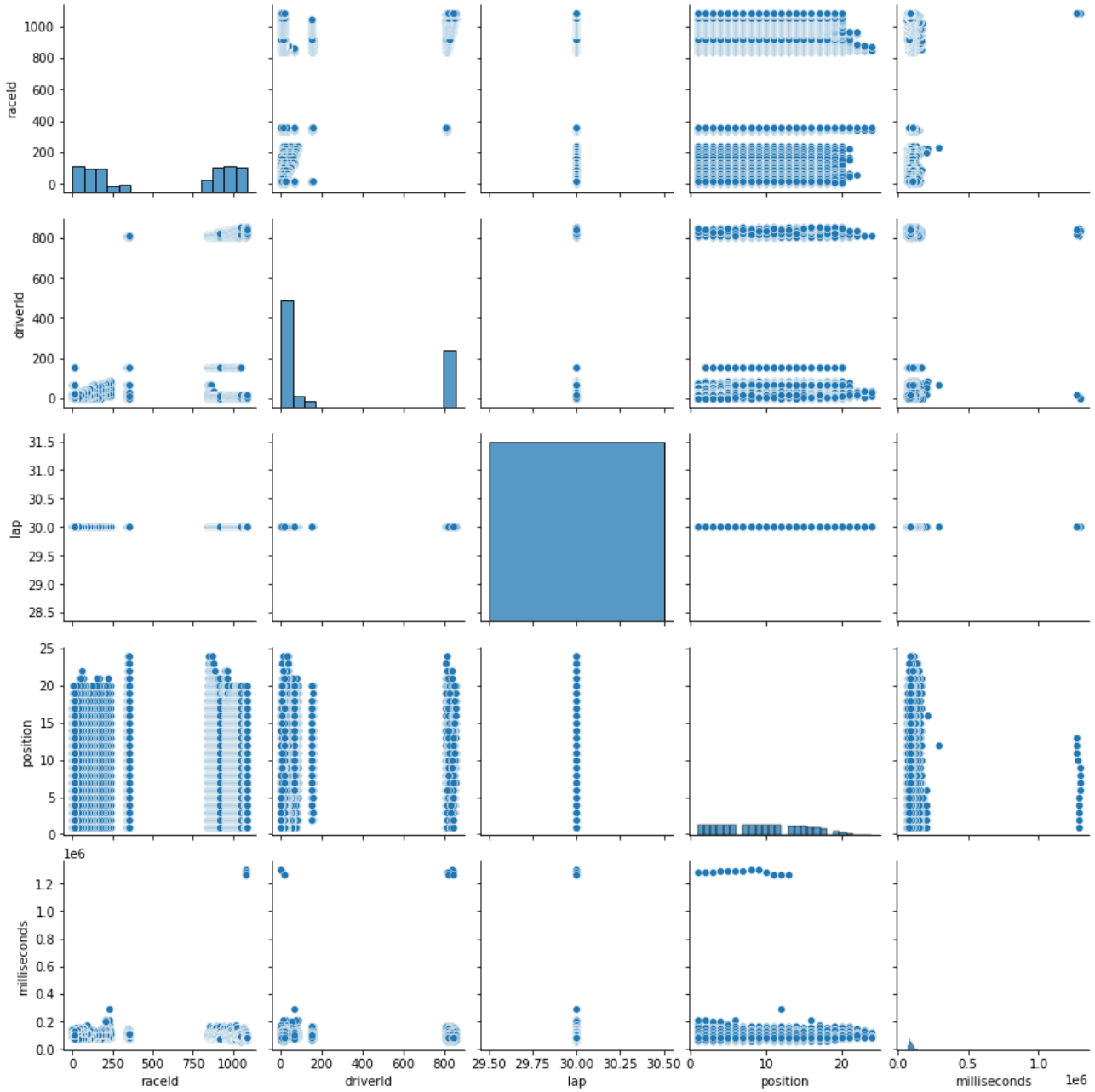
In [3]:
```
df.describe()
```

Out[3]:

	raceId	driverId	lap	position	milliseconds
count	528785.000000	528785.000000	528785.000000	528785.000000	5.287850e+05
mean	541.681793	277.785648	29.967802	9.650514	9.567868e+04
std	419.900529	370.144257	18.410254	5.541874	7.533537e+04
min	1.000000	1.000000	1.000000	1.000000	5.540400e+04
25%	126.000000	15.000000	14.000000	5.000000	8.206100e+04
50%	352.000000	35.000000	29.000000	9.000000	9.067200e+04
75%	960.000000	815.000000	44.000000	14.000000	1.022090e+05
max	1086.000000	855.000000	87.000000	24.000000	7.507547e+06

In [5]:
```
sns.pairplot(df)
```

Out[5]:
```
<seaborn.axisgrid.PairGrid at 0x7fc41fdf06d0>
```



In [6]:
```
print("Correlation")
df.corr()
```

```
Correlation
```

Out[6]:

	raceId	driverId	lap	position	milliseconds
raceId	1.000000	0.676163	NaN	0.068959	0.091771
driverId	0.676163	1.000000	NaN	0.193690	0.060703
lap	NaN	NaN	NaN	NaN	NaN
position	0.068959	0.193690	NaN	1.000000	0.005281
milliseconds	0.091771	0.060703	NaN	0.005281	1.000000

In [7]:
```
df.describe()
```

Out[7]:

	raceId	driverId	lap	position	milliseconds
count	8747.000000	8747.000000	8747.0	8747.000000	8.747000e+03
mean	546.266263	280.303990	30.0	9.657025	9.671870e+04
std	419.644694	371.243887	0.0	5.470845	4.857071e+04
min	1.000000	1.000000	30.0	1.000000	5.780300e+04
25%	127.000000	15.000000	30.0	5.000000	8.321350e+04
50%	354.000000	35.000000	30.0	9.000000	9.260500e+04
75%	961.000000	815.000000	30.0	14.000000	1.035170e+05
max	1086.000000	855.000000	30.0	24.000000	1.297841e+06

Normalizing the dataset

In [8]:
```
scaler = MinMaxScaler()
df["raceId"] = scaler.fit_transform(df["raceId"].values.reshape(-1, 1))
df["driverId"] = scaler.fit_transform(df["driverId"].values.reshape(-1, 1))
df["position"] = scaler.fit_transform(df["position"].values.reshape(-1, 1))
df["milliseconds"] = scaler.fit_transform(df["milliseconds"].values.reshape(-1, 1))
df
```

Out[8]:

	raceId	driverId	lap	position	milliseconds
29	0.774194	0.022248	30	0.000000	0.026815
87	0.774194	0.000000	30	0.043478	0.026891
145	0.774194	0.018735	30	0.173913	0.026710
203	0.774194	0.944965	30	0.086957	0.027157
261	0.774194	0.014052	30	0.130435	0.028598
...	...	...	...	...	...
528470	1.000000	0.996487	30	0.608696	0.022315
528538	1.000000	1.000000	30	0.695652	0.021838
528607	1.000000	0.022248	30	0.478261	0.022281
528676	1.000000	0.992974	30	0.739130	0.022423
528745	1.000000	0.984778	30	0.521739	0.021975

8747 rows × 5 columns

In [9]:
```
X = df[['driverId', 'lap', 'position', 'milliseconds']]
Y = df['raceId']
print(X.shape)
print(Y.shape)
```

```
(8747, 4)
(8747,)
```

In [10]:
```
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size = 0.2)
```

In [11]:
```
regressor = LinearRegression()
regressor.fit(np.array(X_train), y_train)
```

Out[11]:
```
LinearRegression()
```
In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

In [12]:
```
y_pred = regressor.predict(np.array(X_test))
```

In [19]:
```
y_test = scaler.fit_transform(np.array(y_test).reshape(-1, 1))
y_pred = scaler.fit_transform(np.array(y_pred).reshape(-1, 1))
```

In [20]:
```
df_preds = pd.DataFrame({'Actual': y_test.squeeze(), 'Predicted': y_pred.squeeze()})
print(df_preds)
```

```
        Actual  Predicted
0     0.810138   0.077504
1     0.090323   0.069950
2     0.058986   0.059871
3     0.326267   0.025737
4     0.013825   0.044555
...        ...        ...
1745  0.090323   0.028005
1746  0.137327   0.014820
1747  0.825806   0.553062
1748  0.101382   0.049665
1749  0.861751   0.567543

[1750 rows x 2 columns]
```

In [21]:
```
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)
```

In [22]:
```
print(f'Mean absolute error: {mae}')
print(f'Mean squared error: {mse}')
print(f'Root mean squared error: {rmse}')
```

```
Mean absolute error: 0.27932962530823174
Mean squared error: 0.14803316286992774
Root mean squared error: 0.38475078020704223
```

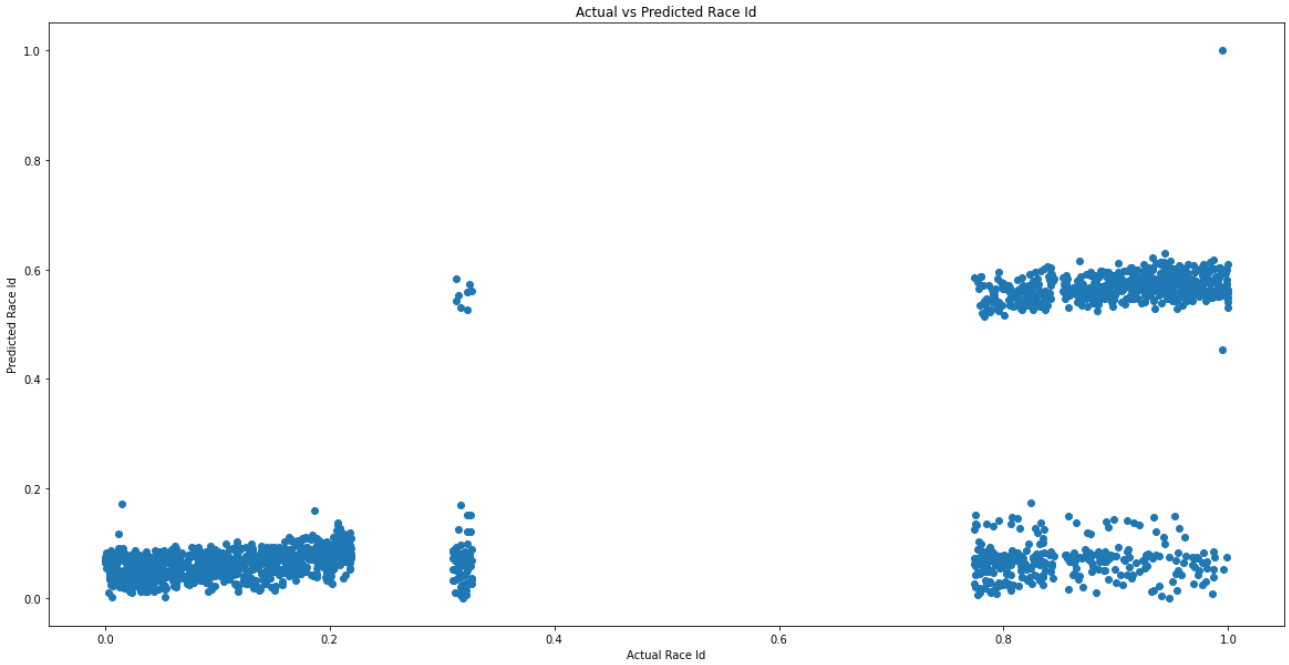
In [23]:
```
print(f"Intercept {regressor.intercept_}")
print(f"Coefficient {regressor.coef_}")

y = regressor.coef_*(X_test) + regressor.intercept_
```

```
Intercept 0.33193015675659165
Coefficient [ 6.03203446e-01 -1.73472348e-15 -1.02138862e-01  4.58503206e-01]
```

In [24]:
```
y_test = scaler.inverse_transform(np.array(y_test))
y_pred = scaler.inverse_transform(np.array(y_pred))
```

In [28]:
```
plt.figure(figsize = (20, 10))
plt.scatter(y_test, y_pred)
plt.title("Actual vs Predicted Race Id")
plt.xlabel("Actual Race Id")
plt.ylabel("Predicted Race Id")
```

Out[28]:
```
Text(0, 0.5, 'Predicted Race Id')
```



CONCLUSION:

Thus, Linear Regression is succesfully implemented and the RMSE, MAE and MSE are calculated.

Viva Questions for Linear Regression

1. What is linear regression, and how is it used in machine learning?
2. What is the difference between simple linear regression and multiple linear regression?
3. How do you define the target variable and the feature variables in linear regression?
4. What is the purpose of the RMSE metric in linear regression, and how is it calculated?
5. What is the difference between RMSE and R-squared in linear regression?
6. What is the interpretation of the slope and intercept coefficients in linear regression?
7. How do you perform linear regression using the scikit-learn library in Python?
8. What is the purpose of the train-test split in linear regression, and how is it performed?
9. How do you evaluate the performance of a linear regression model using RMSE?
10. How do you interpret the RMSE value in linear regression, and what is a good RMSE value?

Experiment 3: Ridge and Lasso Regression

AIM: To implement Ridge and Lasso Regression using the sci-kit learn library

DESCRIPTION:

Ridge Regression: Ridge regression is a regularization technique used in linear regression to prevent overfitting and improve the generalization performance of the model. In regular linear regression, the model tries to fit the training data as closely as possible, which can lead to overfitting, where the model becomes too complex and performs poorly on new, unseen data.

$$\sum_{j=1}^{p}(Y - \hat{Y}_i)^2 + \alpha \sum_{j=1}^{p} \beta_j^2$$

Lasso Regression: Lasso regression is a regularization technique used in linear regression to prevent overfitting and improve the generalization performance of the model, similar to ridge regression. However, unlike ridge regression, lasso regression uses the L1 norm of the coefficients as the regularization penalty, which has the effect of shrinking some of the coefficients to zero. This can lead to a more interpretable model, where some of the independent variables are excluded entirely.

$$\sum_{j=1}^{p}(Y - \hat{Y}_i)^2 + \alpha \sum_{j=1}^{p} |\beta_j|$$

In [31]:
```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import Ridge, Lasso
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_absolute_error, mean_squared_error

df = pd.read_csv("./lap_times.csv")
df.pop("time")
df.dropna()
print("Dataset Information")
df.head()
```

```
Dataset Information
```

Out[31]:

	raceId	driverId	lap	position	milliseconds
0	841	20	1	1	98109
1	841	20	2	1	93006
2	841	20	3	1	92713
3	841	20	4	1	92803
4	841	20	5	1	92342

In [6]:
```
print("Correlation")
df.corr()
```

```
Correlation
```

Out[6]:

	raceId	driverId	lap	position	milliseconds
raceId	1.000000	0.676163	NaN	0.068959	0.091771
driverId	0.676163	1.000000	NaN	0.193690	0.060703
lap	NaN	NaN	NaN	NaN	NaN
position	0.068959	0.193690	NaN	1.000000	0.005281
milliseconds	0.091771	0.060703	NaN	0.005281	1.000000

Normalizing the dataset

In [8]:
```
scaler = MinMaxScaler()
df["raceId"] = scaler.fit_transform(df["raceId"].values.reshape(-1, 1))
df["driverId"] = scaler.fit_transform(df["driverId"].values.reshape(-1, 1))
df["position"] = scaler.fit_transform(df["position"].values.reshape(-1, 1))
df["milliseconds"] = scaler.fit_transform(df["milliseconds"].values.reshape(-1, 1))
df
```

Out[8]:

	raceId	driverId	lap	position	milliseconds
29	0.774194	0.022248	30	0.000000	0.026815
87	0.774194	0.000000	30	0.043478	0.026891
145	0.774194	0.018735	30	0.173913	0.026710
203	0.774194	0.944965	30	0.086957	0.027157
261	0.774194	0.014052	30	0.130435	0.028598
...	...	...	...	...	...
528470	1.000000	0.996487	30	0.608696	0.022315
528538	1.000000	1.000000	30	0.695652	0.021838
528607	1.000000	0.022248	30	0.478261	0.022281
528676	1.000000	0.992974	30	0.739130	0.022423
528745	1.000000	0.984778	30	0.521739	0.021975

8747 rows × 5 columns

In [9]:
```
X = df[['driverId', 'lap', 'position', 'milliseconds']]
Y = df['raceId']
print(X.shape)
print(Y.shape)
```

```
(8747, 4)
(8747,)
```

In [10]:
```
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size = 0.2)
```

In [ ]:
```
regressor = Ridge()
regressor_lasso = Lasso()
regressor.fit(X_train, y_train)
regressor_lasso.fit(X_train, y_train)
```

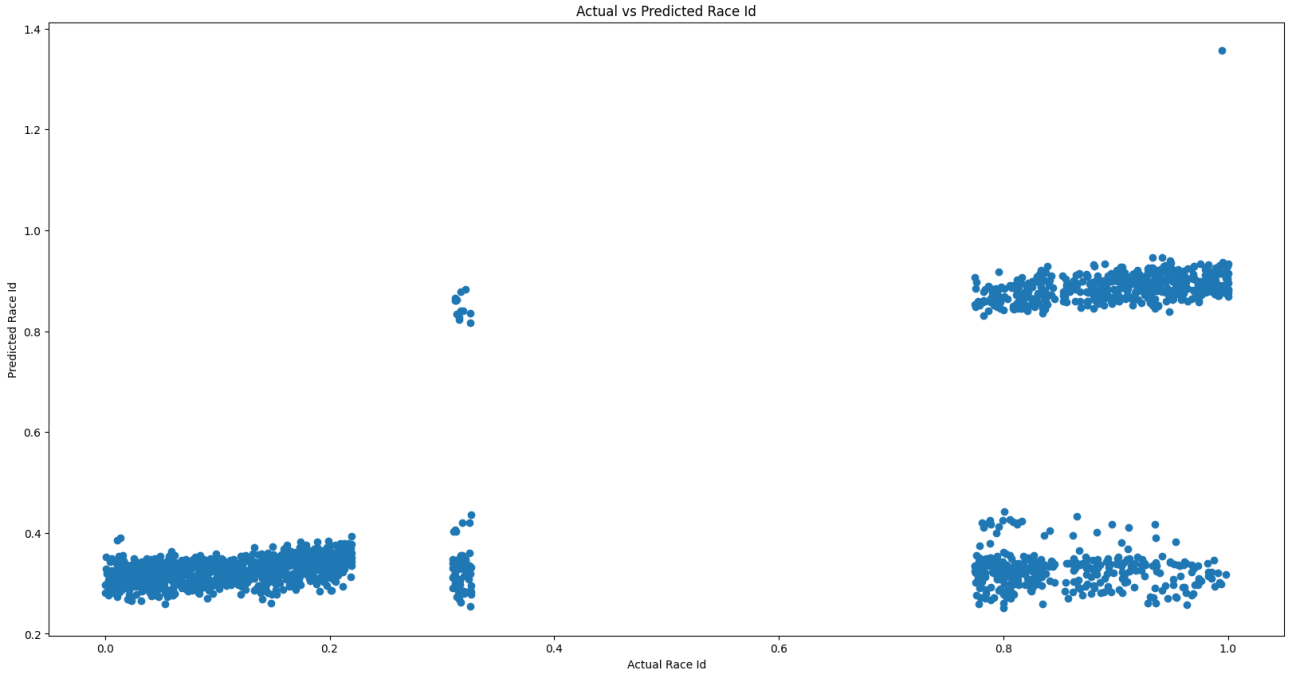
In [20]:
```
y_pred = regressor.predict(X_test)
y_lasso = regressor_lasso.predict(X_test)
```

In [25]:
```
print("Ridge Regression : ")
print(f'Mean absolute error: {mean_absolute_error(y_test, y_pred)}')
print(f'Mean squared error: {mean_squared_error(y_test, y_pred)}')
print(f'Root mean squared error: {np.sqrt(mse)}')

print("Lasso Regression : ")
print(f'Mean absolute error: {mean_absolute_error(y_test, y_lasso)}')
print(f'Mean squared error: {mean_squared_error(y_test, y_lasso)}')
print(f'Root mean squared error: {np.sqrt(mse_lasso)}')
```

```
Ridge Regression :
Mean absolute error: 0.2229741620316311
Mean squared error: 0.08165389079913406
Root mean squared error: 0.2857514493386413
Lasso Regression :
Mean absolute error: 0.3773851384252613
Mean squared error: 0.14832925376228748
Root mean squared error: 0.38513537069748277
```

In [34]:
```
plt.figure(figsize = (20, 10))
plt.scatter(y_test, y_pred)
plt.title("Actual vs Predicted Race Id")
plt.xlabel("Actual Race Id")
plt.ylabel("Predicted Race Id")
```

Out[34]:
```
Text(0, 0.5, 'Predicted Race Id')
```



CONCLUSION:

Thus, Ridge and Lasso Regression are succesfully implemented and the RMSE, MAE and MSE are calculated.

Viva Questions for Lasso and Ridge Regression

1. What is regularization, and why is it important in machine learning?
2. What is the difference between Lasso and Ridge regression?
3. How do Lasso and Ridge regression differ in terms of penalty functions?
4. What is the purpose of the regularization parameter in Lasso and Ridge regression?
5. How do you define the target variable and the feature variables in Lasso and Ridge regression?
6. How do you perform Lasso and Ridge regression using the scikit-learn library in Python?
7. How do you evaluate the performance of a Lasso or Ridge regression model using cross-validation?
8. What is the interpretation of the coefficients in Lasso and Ridge regression?
9. How do you select the optimal value of the regularization parameter in Lasso and Ridge regression?
10. What is the impact of the regularization parameter on the bias-variance trade-off in Lasso and Ridge regression?

Experiment 4: Decision Trees

AIM: To implement Decision Trees for classification using the sci-kit learn library.

DESCRIPTION:

Decision trees are a type of supervised learning algorithm that can be used for both classification and regression tasks. A decision tree works by recursively partitioning the data into subsets based on the values of one or more input features, until a stopping criterion is met. Each internal node of the tree represents a test of one of the input features, and each leaf node represents a predicted output value.

Entropy and information gain are concepts used in decision tree algorithms to help determine the best split for partitioning the data at each internal node.

Entropy is a measure of the impurity of a set of examples. Information gain is a measure of the reduction in entropy achieved by partitioning the examples based on a particular attribute. The attribute that results in the highest information gain is chosen as the splitting attribute at each internal node.

$$\text{Information Gain}(S,a) = \text{Entropy}(S) - \sum_{v \in \text{values}(a)} \frac{|S_v|}{|S|} \text{Entropy}(S_v)$$

In [7]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix
```

In [8]:
```
# Reading the Iris.csv file
data = load_iris()

# Extracting Attributes / Features
X = data.data

# Extracting Target / Class Labels
y = data.target
```

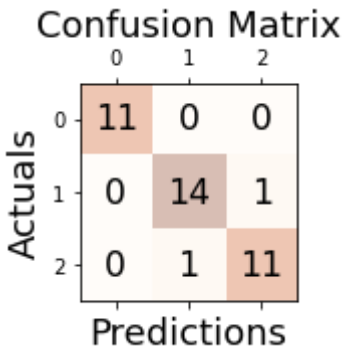
In [9]:
```
X_train, X_test, y_train, y_test = train_test_split(X,y, random_state = 50, test_size
```

In [10]:
```
clf = DecisionTreeClassifier()
clf.fit(X_train,y_train)
```

Out[10]:
```
DecisionTreeClassifier
DecisionTreeClassifier()
```

In [11]:
```
y_pred = clf.predict(X_test)
```

In [12]:
```
from sklearn.metrics import confusion_matrix, classification_report

conf_matrix = confusion_matrix(y_true=y_test, y_pred=y_pred)

fig, ax = plt.subplots(figsize=(2, 2))
ax.matshow(conf_matrix, cmap=plt.cm.Oranges, alpha=0.3)
for i in range(conf_matrix.shape[0]):
    for j in range(conf_matrix.shape[1]):
        ax.text(x=j, y=i,s=conf_matrix[i, j], va='center', ha='center', size='xx-larg

plt.xlabel('Predictions', fontsize=18)
plt.ylabel('Actuals', fontsize=18)
plt.title('Confusion Matrix', fontsize=18)
plt.show()
```



In [13]:
```
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        11
           1       0.93      0.93      0.93        15
           2       0.92      0.92      0.92        12

    accuracy                           0.95        38
   macro avg       0.95      0.95      0.95        38
weighted avg       0.95      0.95      0.95        38
```

CONCLUSION:

Thus, Decision Trees are succefully implemented using the sci-kit learn library and are used to perform classification on the *iris* dataset.

Viva Questions for Decision Tree Classifier

1. What is classification, and why is it important in machine learning?
2. What are the different types of classification algorithms?
3. What is a decision tree, and how is it used in classification?
4. How do you define the target variable and the feature variables in classification?
5. How do you split the dataset into training and testing sets for classification?
6. How do you measure the performance of a classification model?
7. What is the difference between accuracy, precision, recall, and F1-score in classification?
8. How do you build a decision tree classifier using the scikit-learn library in Python?
9. What is Entropy?
10. What is Information gain and how nodes are split in the decision tree?

Experiment 5: Support Vector Machines

AIM: To implement SVMs using the sci-kit learn library.

DESCRIPTION:

Support Vector Machines (SVMs) are a powerful class of supervised learning algorithms that can be used for both classification and regression tasks. SVMs work by finding the hyperplane that best separates the data into different classes, with the goal of maximizing the margin, which is the distance between the hyperplane and the closest data points.

SVMs can be used with different types of kernel functions, which are used to transform the data into a higher-dimensional space where it may be easier to find a separating hyperplane. Some of the most commonly used kernel functions are:

1. Linear kernel: This is the simplest kernel function, which simply computes the dot product of the input features. It works well for linearly separable data.
2. Polynomial kernel: This kernel function maps the data into a higher-dimensional space using a polynomial function. It can capture non-linear relationships between the input features.
3. Radial basis function (RBF) kernel: This is one of the most commonly used kernel functions, which maps the data into an infinite-dimensional space using a Gaussian kernel. It works well for non-linearly separable data and can capture complex decision boundaries.

In [57]:
```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

In [58]:
```
colnames=["sepal_length_in_cm", "sepal_width_in_cm","petal_length_in_cm","petal_width
dataset = pd.read_csv("https://archive.ics.uci.edu/ml/machine-learning-databases/iris
dataset.head()
```

Out[58]:

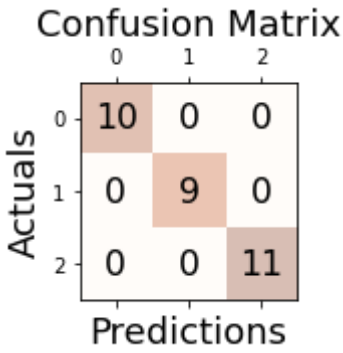
	sepal_length_in_cm	sepal_width_in_cm	petal_length_in_cm	petal_width_in_cm	class
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

In [59]:
```
X = dataset.iloc[:,:-1]
y = dataset.iloc[:, -1].values

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_sta
```

Creating a SVM with a Linear Kernel

In [60]:
```
#Create the SVM model
from sklearn.svm import SVC
classifier = SVC(kernel = 'linear', random_state = 0)
classifier.fit(X_train, y_train)
y_pred = classifier.predict(X_test)
```

In [61]:
```
from sklearn.metrics import confusion_matrix, classification_report

conf_matrix = confusion_matrix(y_true=y_test, y_pred=y_pred)

fig, ax = plt.subplots(figsize=(2, 2))
ax.matshow(conf_matrix, cmap=plt.cm.Oranges, alpha=0.3)
for i in range(conf_matrix.shape[0]):
    for j in range(conf_matrix.shape[1]):
        ax.text(x=j, y=i,s=conf_matrix[i, j], va='center', ha='center', size='xx-larg

plt.xlabel('Predictions', fontsize=18)
plt.ylabel('Actuals', fontsize=18)
plt.title('Confusion Matrix', fontsize=18)
plt.show()
```



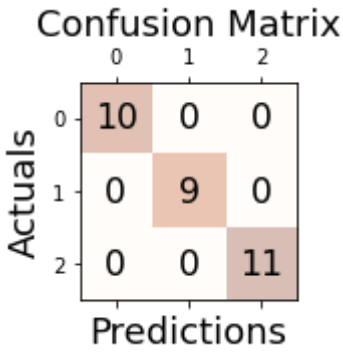
In [62]:
```
print(classification_report(y_test, y_pred))
```

```
                precision    recall  f1-score   support

    Iris-setosa       1.00      1.00      1.00        10
Iris-versicolor       1.00      1.00      1.00         9
 Iris-virginica       1.00      1.00      1.00        11

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

Creating a SVM with a Polynomial Kernel

In [63]:
```
from sklearn.svm import SVC
classifier = SVC(kernel = 'poly', random_state = 0)
classifier.fit(X_train, y_train)
y_pred = classifier.predict(X_test)
print(classification_report(y_test, y_pred))
```

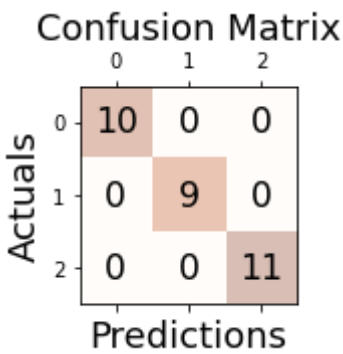
```
                precision    recall  f1-score   support

    Iris-setosa       1.00      1.00      1.00        10
Iris-versicolor       1.00      1.00      1.00         9
 Iris-virginica       1.00      1.00      1.00        11

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



Creating a SVM with a RBF Kernel

In [64]:
```
from sklearn.svm import SVC
classifier = SVC(kernel = 'rbf', random_state = 0)
classifier.fit(X_train, y_train)
y_pred = classifier.predict(X_test)
print(classification_report(y_test, y_pred))
```

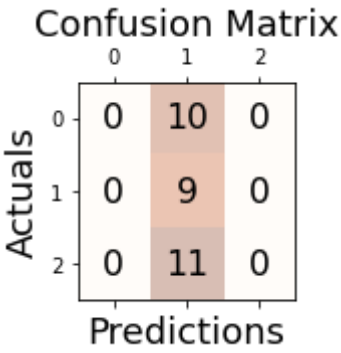
```
                precision    recall  f1-score   support

    Iris-setosa       1.00      1.00      1.00        10
Iris-versicolor       1.00      1.00      1.00         9
 Iris-virginica       1.00      1.00      1.00        11

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



Creating a SVM with a Sigmoid Kernel

In [65]:
```
from sklearn.svm import SVC
classifier = SVC(kernel = 'sigmoid', random_state = 0)
classifier.fit(X_train, y_train)
y_pred = classifier.predict(X_test)
print(classification_report(y_test, y_pred))
```

```
                precision    recall  f1-score   support

    Iris-setosa       0.00      0.00      0.00        10
Iris-versicolor       0.30      1.00      0.46         9
 Iris-virginica       0.00      0.00      0.00        11

       accuracy                           0.30        30
      macro avg       0.10      0.33      0.15        30
   weighted avg       0.09      0.30      0.14        30
```



CONCLUSION:

Thus, SVMs are succesfully implemented and are used to perform classification on *iris* dataset. By varying the kernel of the SVM, the corresponding recall, accuracy, precision and f1 score are calculated for each model.

Viva Questions for SVM

1. What is a Support Vector Machine (SVM), and how is it used in machine learning?
2. What is the difference between linear and nonlinear SVM?
3. What is a kernel function, and why is it used in SVM?
4. What are the different types of kernel functions used in SVM?
5. What is the purpose of the gamma parameter in the RBF kernel function?
6. How do you define the target variable and the feature variables in SVM?
7. How do you split the dataset into training and testing sets for SVM?
8. How do you measure the performance of an SVM model?
9. How do you build an SVM model using the scikit-learn library in Python?

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

Viva Questions for Logistic Regression

1. What is classification, and why is it important in machine learning?
2. What is logistic regression, and how is it used in classification?
3. How do you define the target variable and the feature variables in logistic regression?
4. What is the sigmoid function, and how is it used in logistic regression?
5. How do you split the dataset into training and testing sets for logistic regression?
6. How do you measure the performance of a logistic regression model?
7. What is the difference between accuracy, precision, recall, and F1-score in classification?
8. How do you build a logistic regression model using the scikit-learn library in Python?
9. Difference between Linear and Logistic Regression
10. When can Logistic Regression be applied? (binary or multi class)

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

Viva Questions for Naive Bayes Classifier

1. What is classification, and why is it important in machine learning?
2. What is the Naive Bayes classifier, and how is it used in classification?
3. How do you define the target variable and the feature variables in Naive Bayes?
4. What is the assumption made by the Naive Bayes classifier, and why is it called "naive"?
5. What are the different types of Naive Bayes classifiers?
6. How do you split the dataset into training and testing sets for Naive Bayes?
7. How do you measure the performance of a Naive Bayes classifier?
8. When can Naive Bayes be applied?
9. How do you build a Naive Bayes classifier using the scikit-learn library in Python?
10. What is the purpose of smoothing in Naive Bayes, and how is it performed?

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

Viva Questions for KNN

1. What is the K-Nearest Neighbors (KNN) algorithm, and how is it used in machine learning?
2. How does the KNN algorithm make predictions?
3. How do you choose the optimal value of K in KNN?
4. What is the difference between the Euclidean distance and the Manhattan distance in KNN?
5. How do you define the target variable and the feature variables in KNN?
6. How do you split the dataset into training and testing sets for KNN?
7. How do you measure the performance of a KNN model?
8. What are eager and lazy learners?
9. How do you build a KNN model using the scikit-learn library in Python?
10. What are the advantages and disadvantages of the KNN algorithm?

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

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

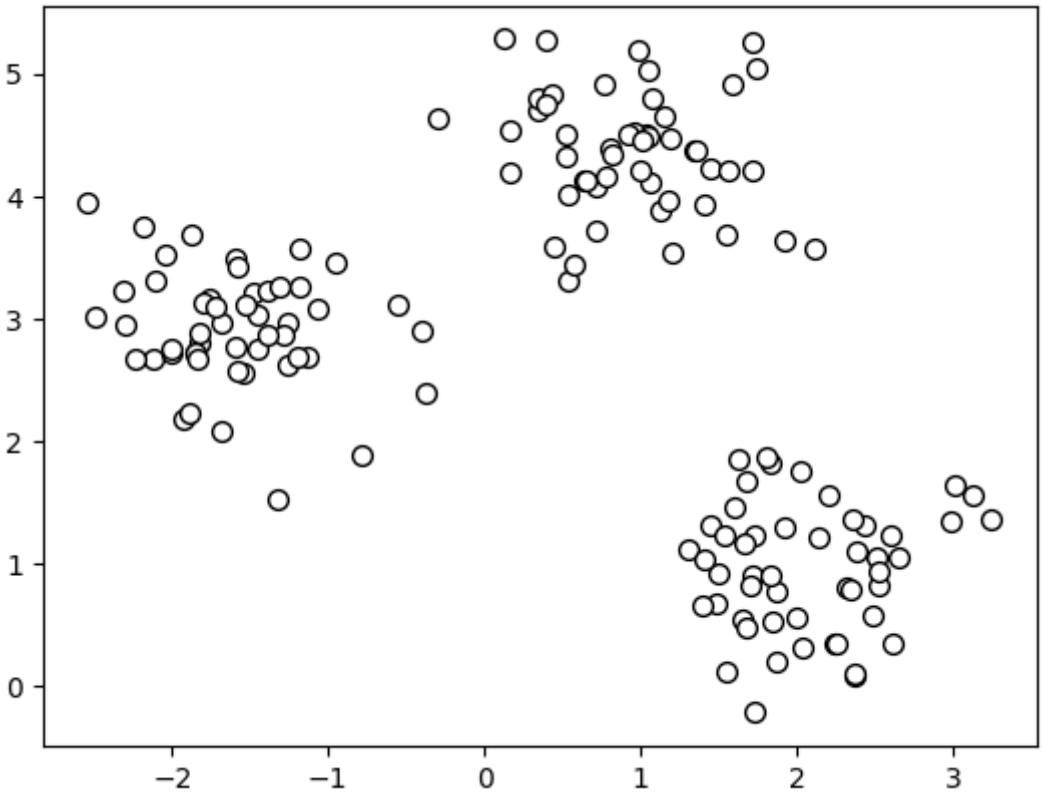
Step 4: Repeat Steps

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
```

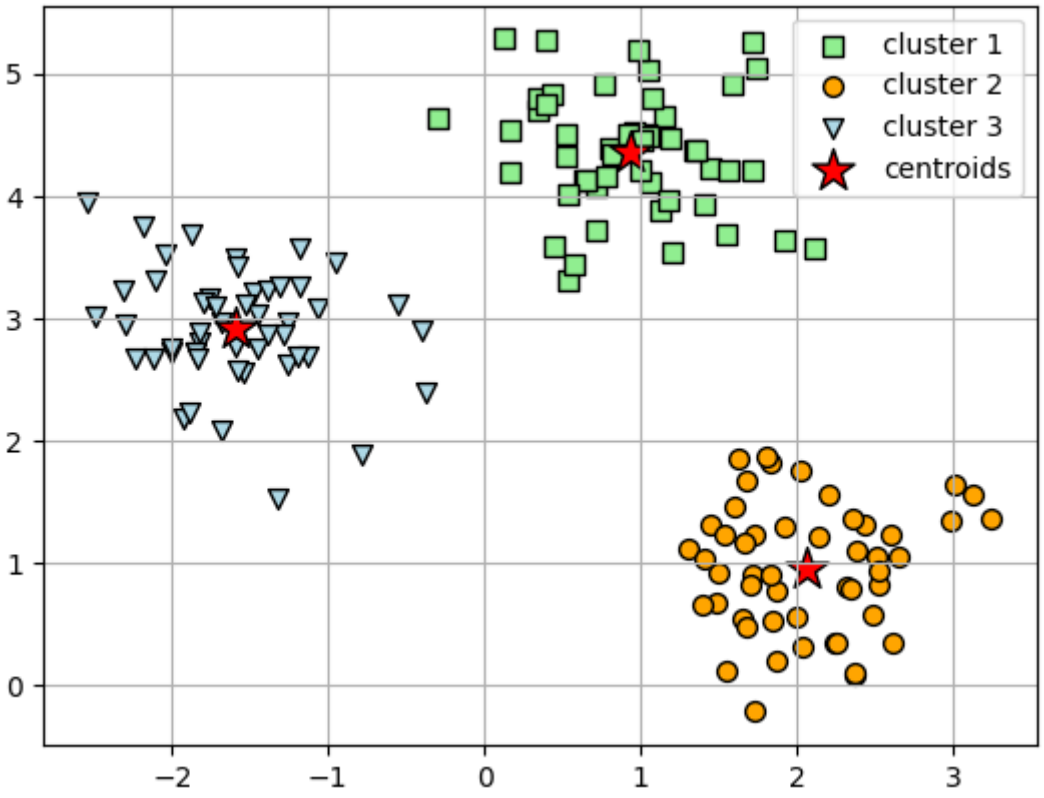
In [ ]:
```python
# Dataset
X, y = make_blobs(
    n_samples=150, n_features=2,
    centers=3, cluster_std=.5,
    shuffle=True, random_state=0
)
```

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



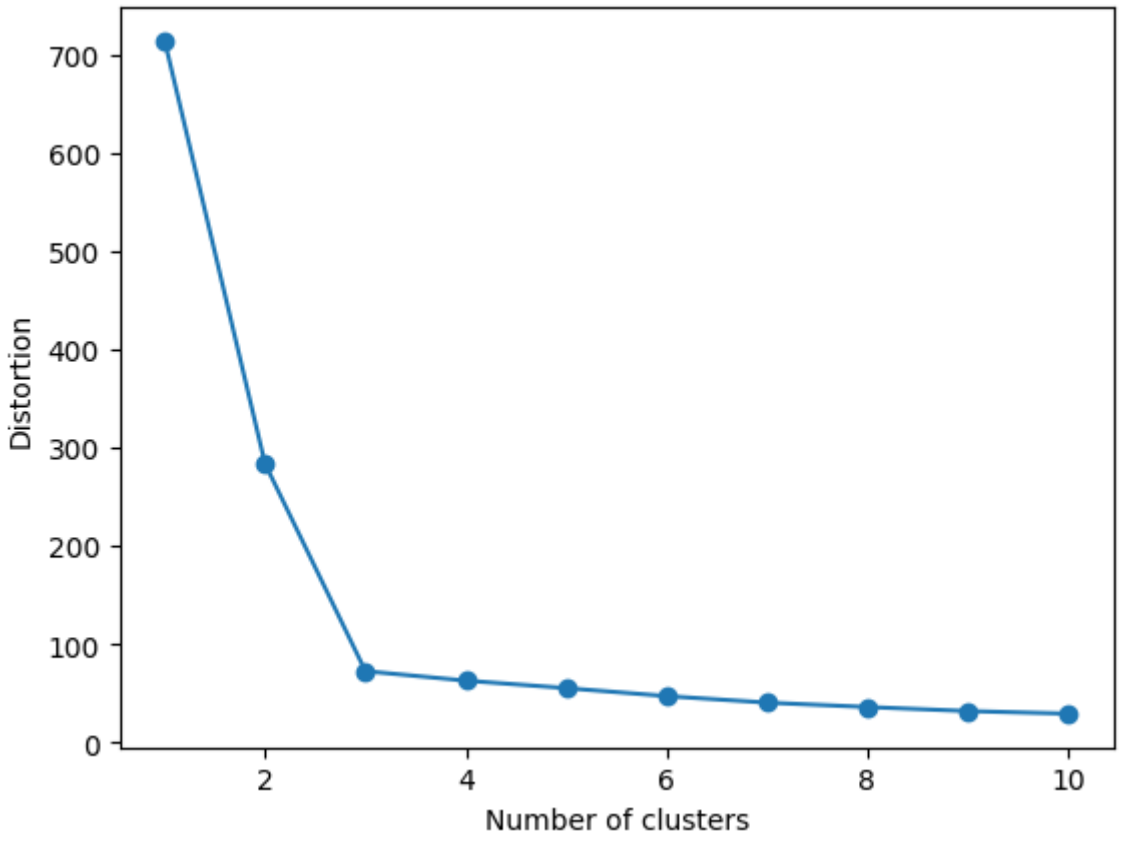
In [ ]:
```
# Traning Model
m = KMeans(
    n_clusters=3, init='random',
    n_init=10, max_iter=300,
    tol=1e-04, random_state=0
)
y_m = m.fit_predict(X)
```

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

Viva Questions for K Means

1. What is clustering, and why is it important in machine learning?
2. What is K-Means clustering, and how is it used in unsupervised learning?
3. How does the K-Means algorithm work?
4. How do you choose the optimal value of K in K-Means clustering?
5. What is the difference between the centroid initialization methods in K-Means?
6. How do you define the features for clustering in K-Means?
7. How do you measure the performance of a K-Means clustering model?
8. What is the elbow method, and how is it used to determine the optimal value of K?
9. How do you visualize the clusters in K-Means clustering?
10. What are the advantages and disadvantages of the K-Means algorithm?

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:

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

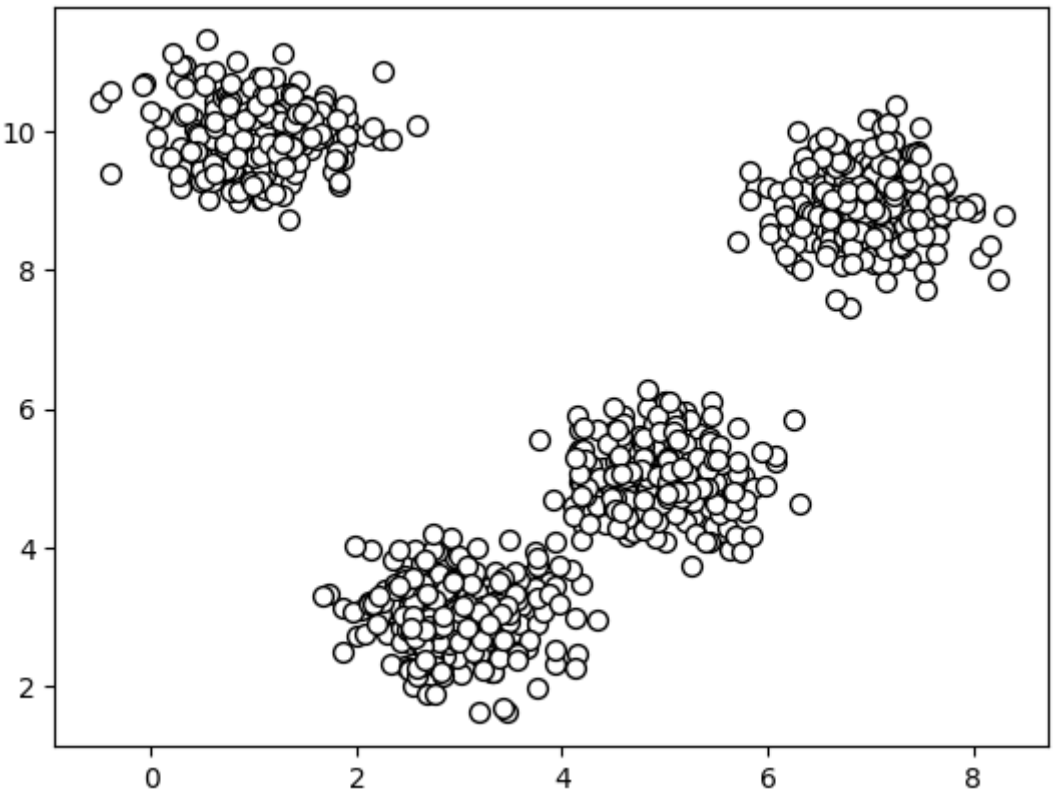
After applying this algorithm, we obtain a dendrogram of clusters that represents the hierarchical structure of the data.

In [ ]:
```python
# 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
```

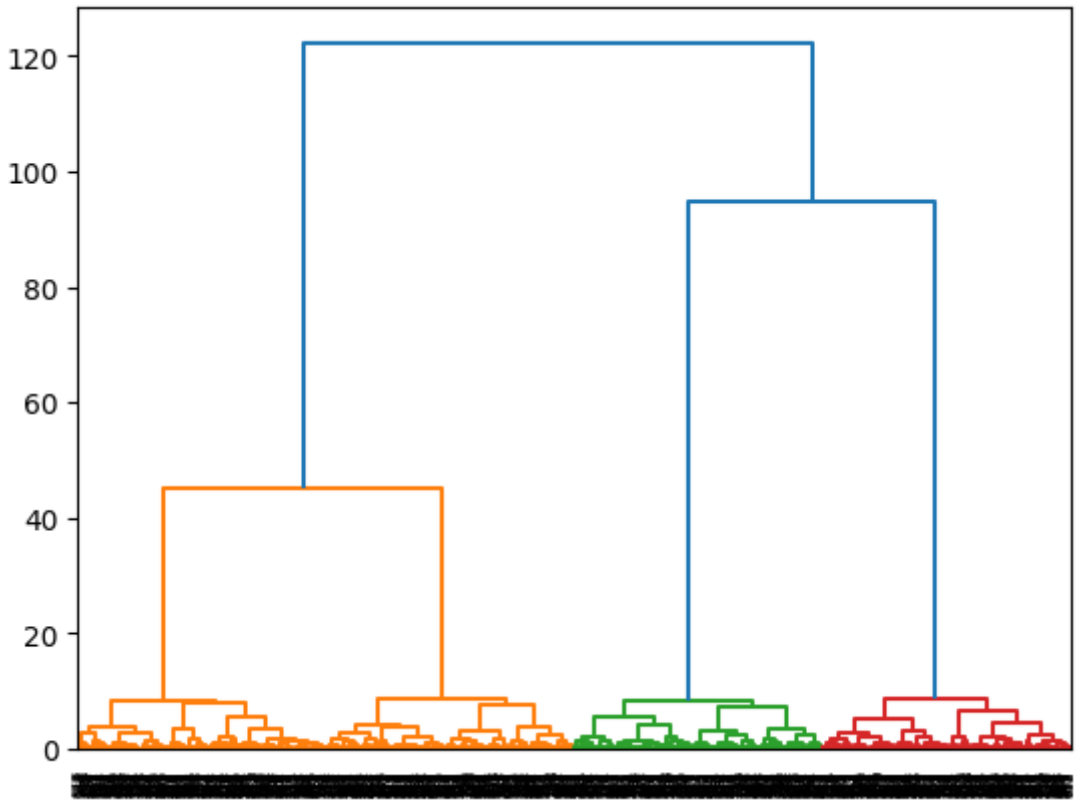
In [ ]:
```python
# Configuration options
num_samples_total = 1000
cluster_centers = [(3,3), (7,9), (1, 10), (5, 5)]
num_classes = len(cluster_centers)
epsilon = 1.0
min_samples = 13
```

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'))
```

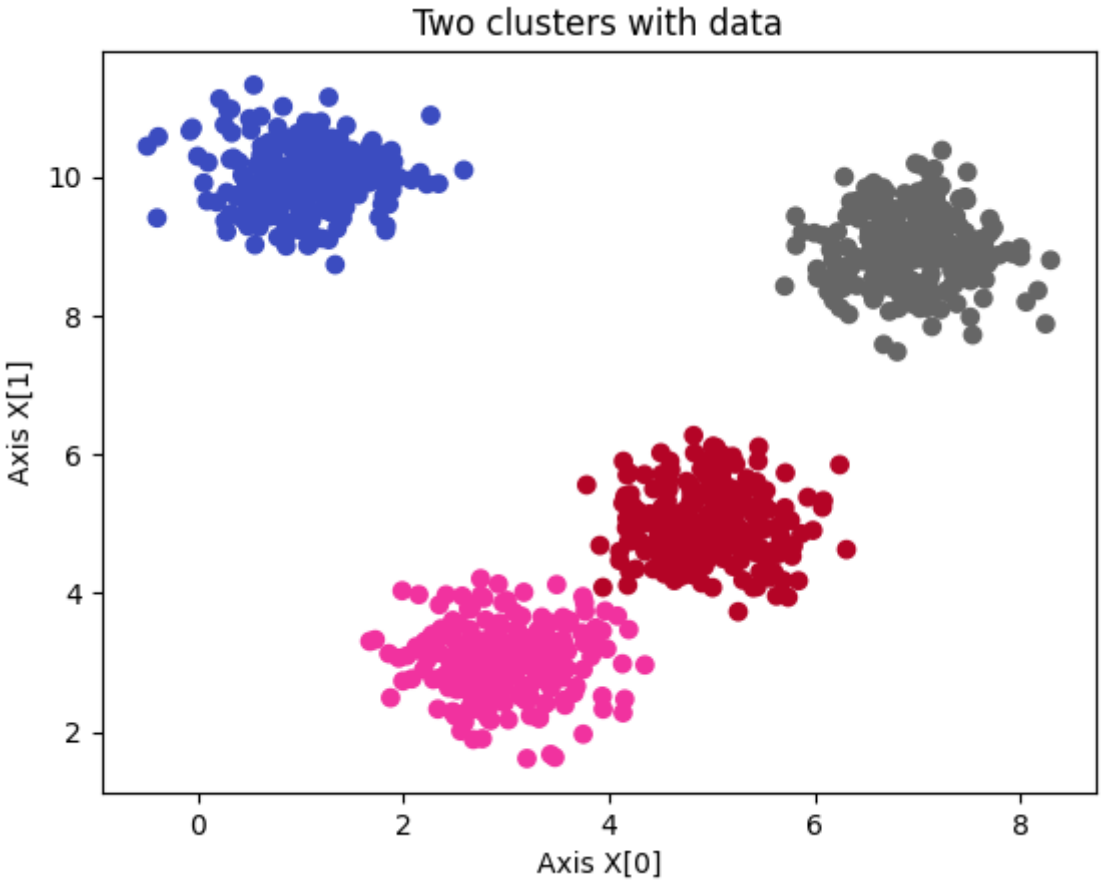


In [ ]:
```
# Traning Model
m = AgglomerativeClustering(
    n_clusters=num_classes,
    metric='euclidean',
    linkage='ward',
)
y_m = m.fit(X)
labels = y_m.labels_
```

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

Viva Questions for Agglomerative Clustering

1. What is clustering, and why is it important in machine learning?
2. What is Agglomerative Clustering, and how is it used in unsupervised learning?
3. How does the Agglomerative Clustering algorithm work?
4. What are the different linkage criteria used in Agglomerative Clustering?
5. How do you choose the optimal number of clusters in Agglomerative Clustering?
6. How do you define the features for clustering in Agglomerative Clustering?
7. How do you measure the performance of an Agglomerative Clustering model?
8. How do you visualize the clusters in Agglomerative Clustering?
9. What is the difference between Agglomerative Clustering and K-Means Clustering?
10. What are the advantages and disadvantages of Agglomerative Clustering?

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.

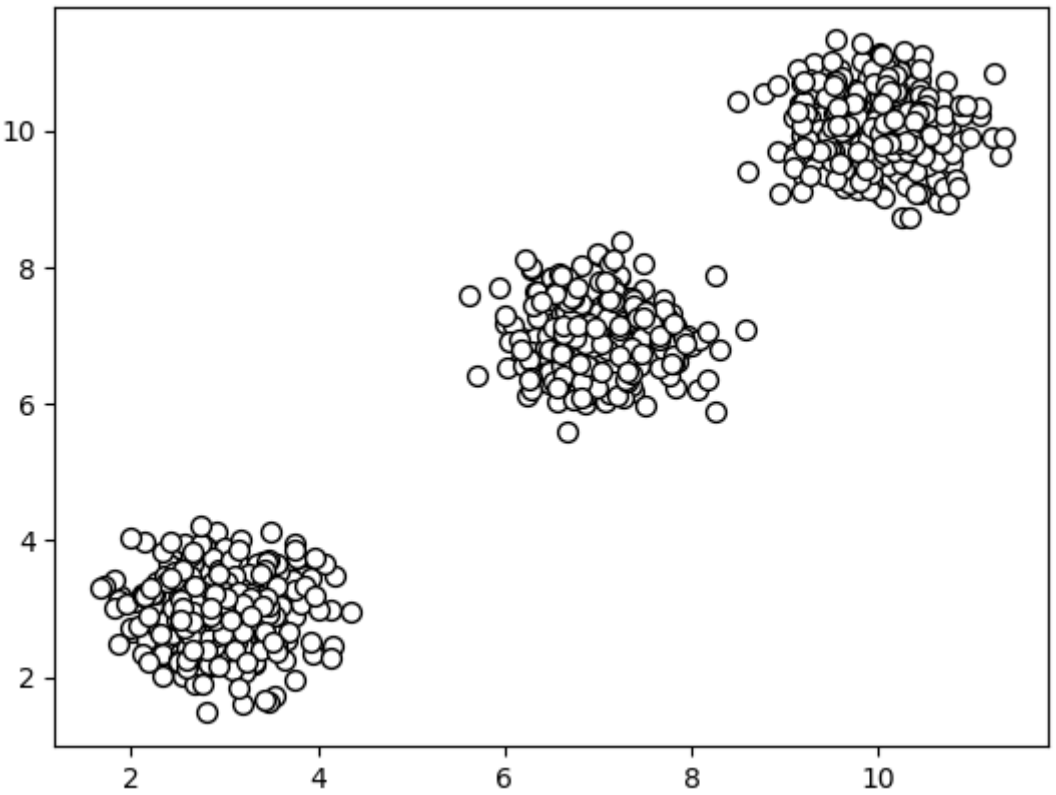
Repeat this process until all observations in X have been visited.

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()
```

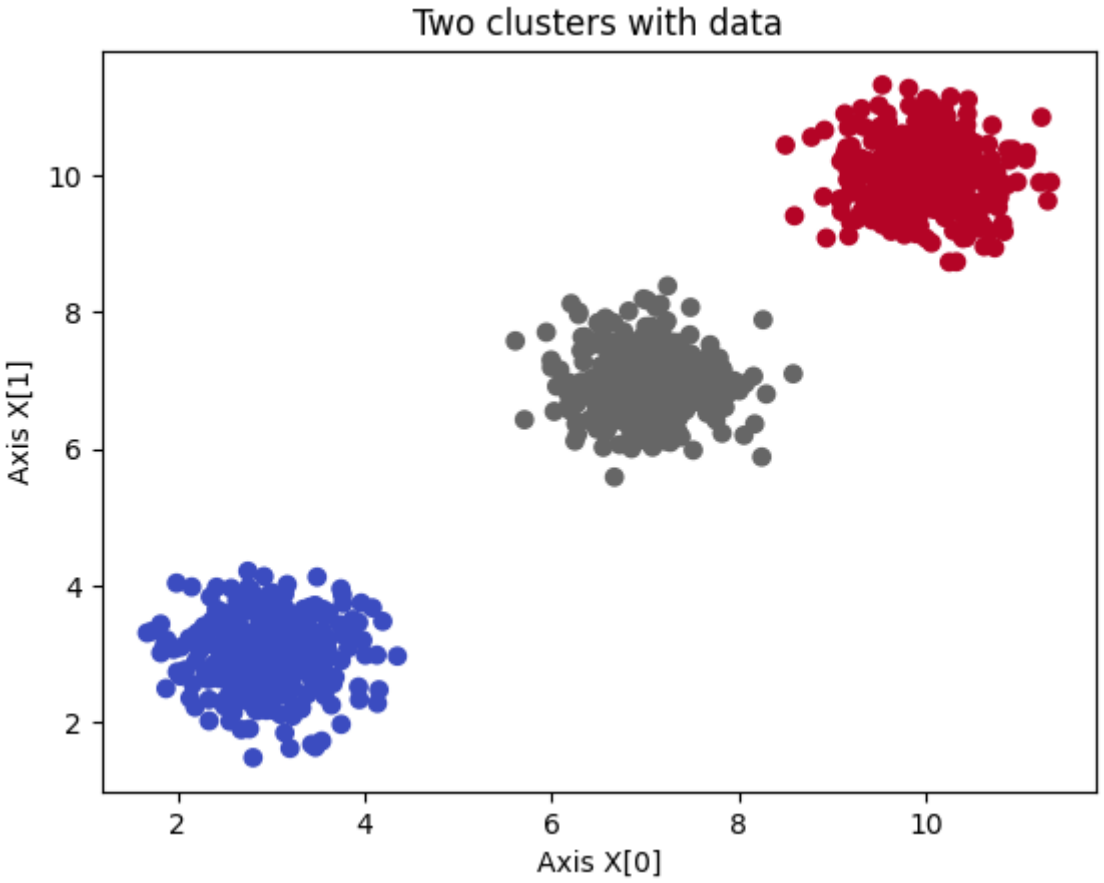


In [ ]:
```
# Traning Model
m = DBSCAN(
    eps=epsilon,
    min_samples=min_samples
)
y_m = m.fit(X)
labels = y_m.labels_
```

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

Viva Questions for DBSCAN

1. What is DBSCAN, and how is it used in unsupervised learning?
2. How does the DBSCAN algorithm work?
3. What are the key parameters in DBSCAN, and how do you choose their values?
4. How do you define the features for clustering in DBSCAN?
5. How do you handle noise and outliers in DBSCAN?
6. How do you measure the performance of a DBSCAN model?
7. How do you visualize the clusters in DBSCAN?
8. What are the advantages and disadvantages of DBSCAN compared to other clustering algorithms?
9. How can you improve the performance of DBSCAN in datasets with varying densities and shapes?

Experiment No: 12

AIM: To implement Gradient Descent using Tensorflow

DESCRIPTION:

Gradient descent is an optimization algorithm used to minimize the value of a function by iteratively adjusting the parameters of the function in the direction of steepest descent of the function's gradient. It is a widely used algorithm in machine learning and deep learning for training models. The general idea of gradient descent is to start with an initial guess for the parameters of the function and then repeatedly update these parameters in the direction of the negative gradient of the function, which indicates the direction of steepest descent. The size of each update is determined by a learning rate, which is a hyperparameter that must be chosen in advance. Repeat until convergence **{ w := w - alpha * (1/m) * sum((h(x) - y) * x) }**

In [8]:

```
import tensorflow as tf
import numpy as np
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])])
```

In [9]:

```
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)
```

In [10]:

```
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("Epoch {}: w = {}, b = {}".format(epoch, model.get_weights()[0], model.get
```

```
Epoch 0: w = [[1.508965]], b = [0.3028648]
Epoch 100: w = [[1.9142624]], b = [-2.5951152]
Epoch 200: w = [[1.9622564]], b = [-2.9493642]
Epoch 300: w = [[1.9681191]], b = [-2.9926374]
Epoch 400: w = [[1.9688351]], b = [-2.9979215]
Epoch 500: w = [[1.968923]], b = [-2.99857]
Epoch 600: w = [[1.9689333]], b = [-2.9986475]
Epoch 700: w = [[1.9689355]], b = [-2.998663]
Epoch 800: w = [[1.9689355]], b = [-2.998663]
Epoch 900: w = [[1.9689355]], b = [-2.998663]
```

CONCLUSION: The Gradient Descent Algorithm has been successfully implemented on a sample dataset

Viva Questions for Gradient Descent

1. What is TensorFlow, and how is it used in machine learning?
2. What is a tensor in TensorFlow, and how is it used to represent data?
3. What is a variable in TensorFlow, and how is it used to represent model parameters?
4. What is gradient descent, and how is it used to optimize the weights in a neural network?
5. What is the learning rate in gradient descent, and how does it affect the optimization process?
6. What is backpropagation, and how is it used to calculate gradients in a neural network?
7. What is the difference between batch gradient descent and stochastic gradient descent?
8. How do you implement a basic neural network using TensorFlow, including the definition of the loss function and the optimization using gradient descent?