List of Experiments

Experiment:1

Aim:-

Study of platform for Implementation of Assignments. Download the open-source software of your interest. Document the distinctfeatures and functionality of the software platform.

The Features And Functionalities Of Anaconda Navigator :

Anaconda Navigator is a graphical user interface (GUI) included with the Anaconda distribution, which is a popular platform for data science and scientific computing in Python.

Here are some distinct features and functionalities of Anaconda Navigator:

- **Package Management**: Anaconda Navigator provides an easy way to manage Python packages and environments. You can create, clone, and switch betweendifferent Python environments, making it simple to isolate and manage dependencies for different projects.
- **Package Search:** You can search for and install packages and libraries from the Anaconda Repository or the broader Python ecosystem using the built-in packagesearch feature. It simplifies the process of finding and installing data science libraries.
- **Environment Management:** Anaconda Navigator allows you to create and manage separate Python environments for different projects. This helps in avoiding conflicts between package versions and ensuring reproducibility.
- **Integrated Text Editors:** It includes integrated text editors like Jupyter Notebook and JupyterLab, which are popular choices for interactive data analysis and codingin Python.
- **Launch Applications:** Navigator enables you to launch various data science applications and tools directly from the interface. This includes Jupyter Notebook,JupyterLab, Spyder, RStudio, and more, making it convenient to access your favorite tools.
- **Manage Conda and Pip:** You can update, install, and manage packages boththrough Conda (Anaconda's package manager) and Pip (Python's package manager) using the Navigator interface.
- **Version Control Integration:** Anaconda Navigator can be integrated with versioncontrol systems like Git, allowing you to manage and track changes in your code and project.
- **Environments Export/Import:** You can export and import environment configurations to share with colleagues or replicate your development environment on different machines easily.
- **Dashboard:** Navigator provides a dashboard-like interface where you can see youravailable environments, installed packages, and launch applications. It offers a convenient overview of your data science setup.
- **Cross-Platform:** Anaconda Navigator is available for Windows, macOS, andLinux, making it accessible on various operating systems.
- **User-Friendly:** One of its main strengths is its user-friendly interface, making itaccessible to both beginners and experienced data scientists.

Overall, Anaconda Navigator streamlines the process of setting up, managing, and working with data science environments, packages, and tools, making it a valuable resource for data scientists

and Python developers.
Here are the steps to install Anaconda Navigator :

Installing Anaconda on Windows

Anaconda comes bundled with about 600 packages pre-installedincluding NumPy, Matplotlib and SymPy.

Follow the steps below to install the Anaconda distíibution of Python on Windows.

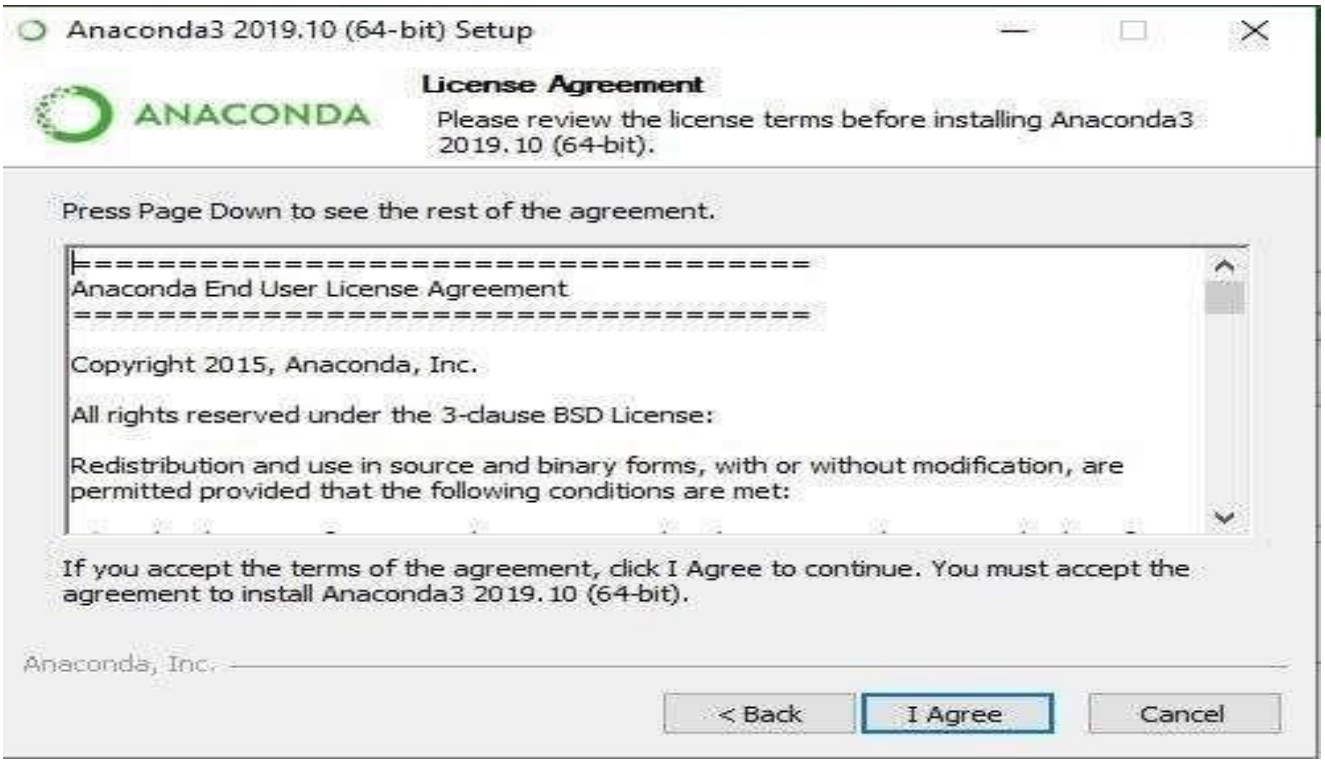
Download and install Anaconda:

Head over to anaconda.com and install the latest version of Anaconda. Make sure todownload the "Python 3.7 Version" for the appropriate architecture.

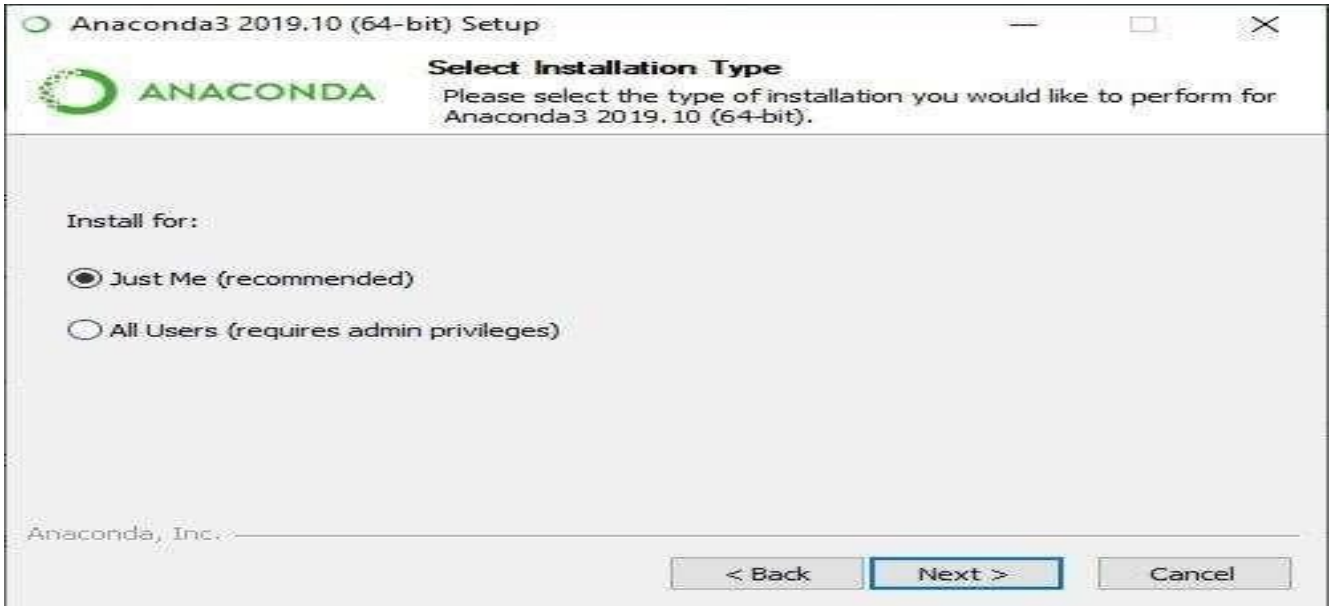
Begin with the installation process:

- **Getting Start**

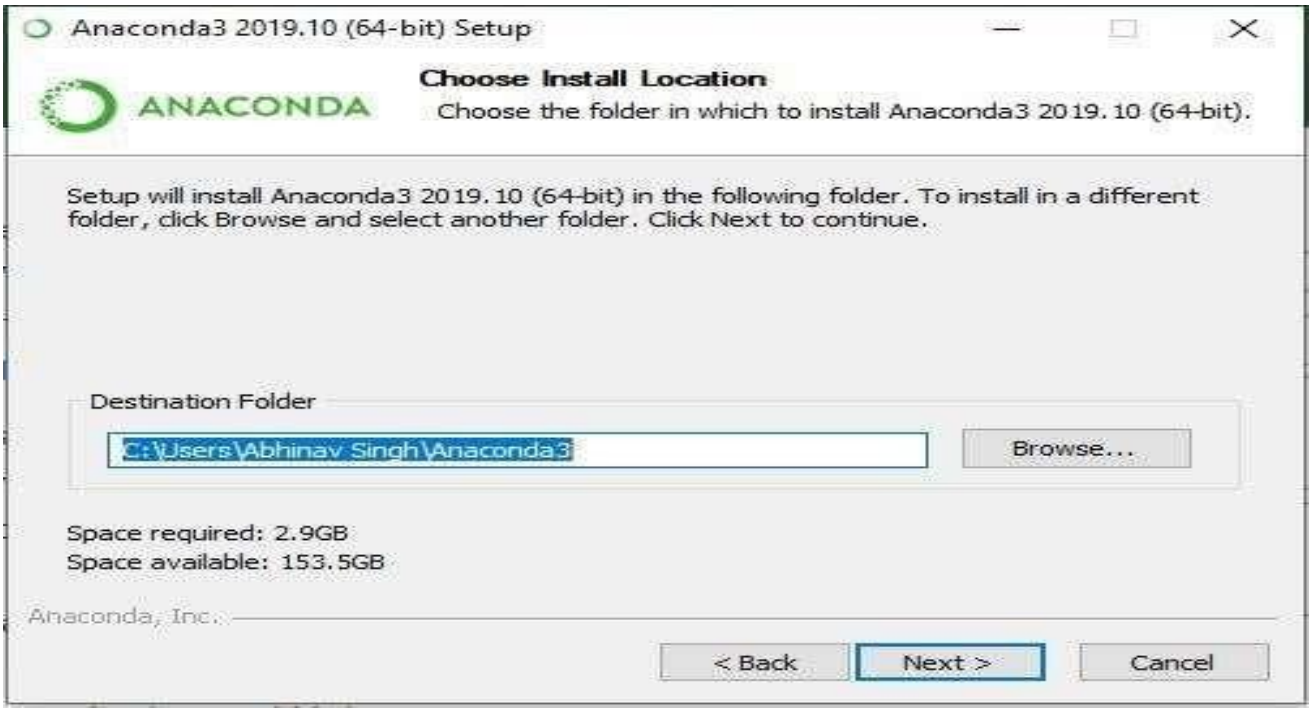
- **Getting through the License Agreement:**



- **Select Installation Type:** Select **Just Me** if you want the software to be used by asingle User

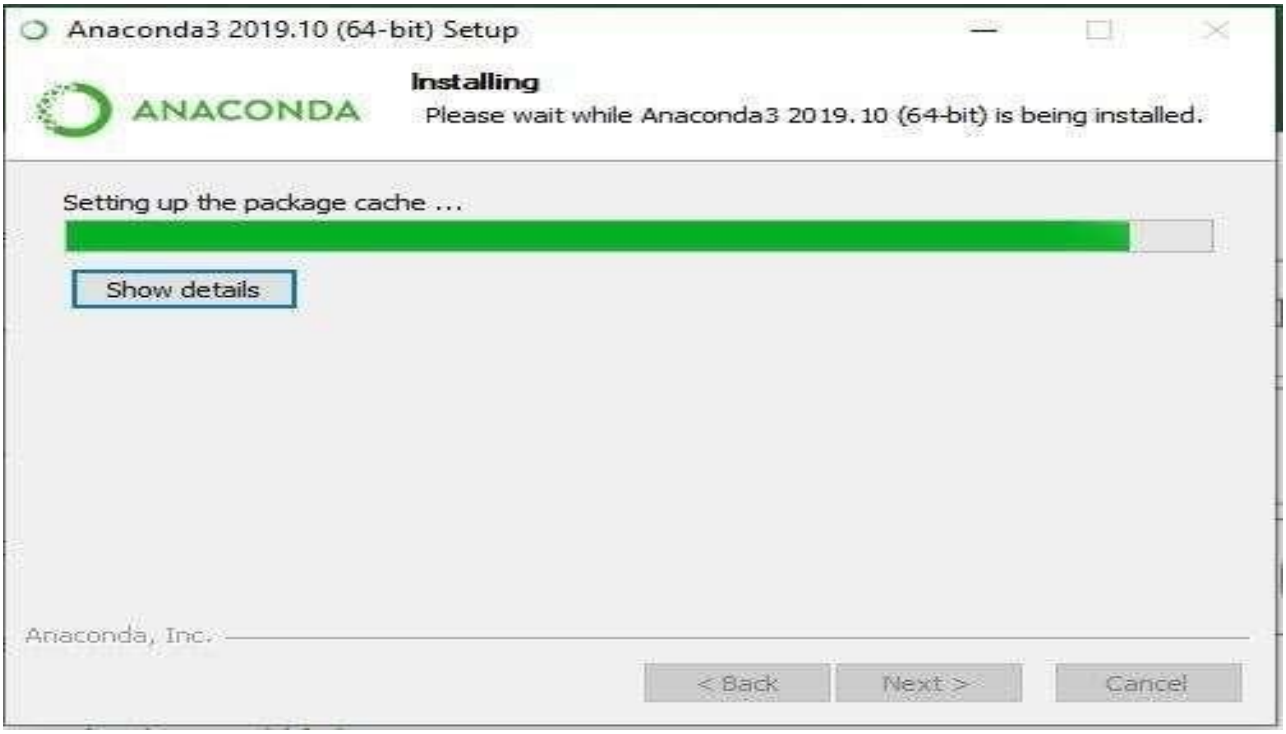


- **Choose Installation Location:**

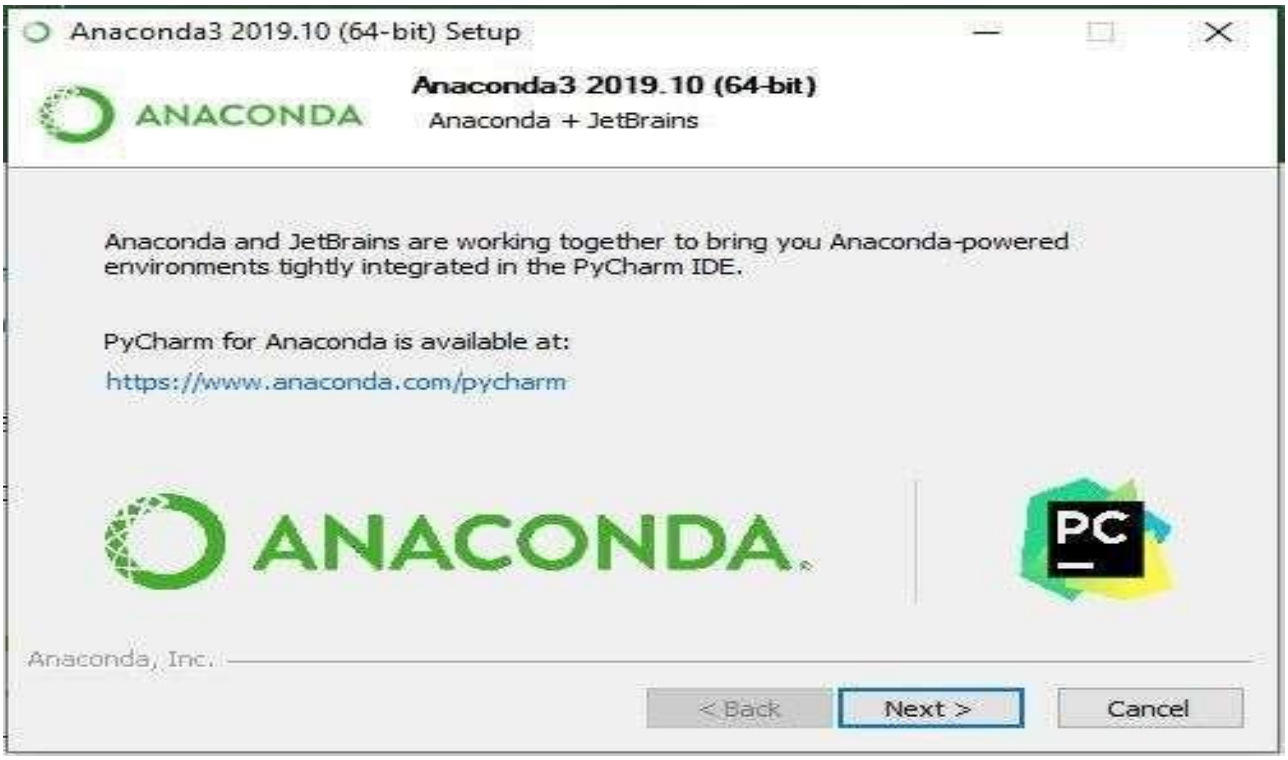


- **Advanced Installation Option:**

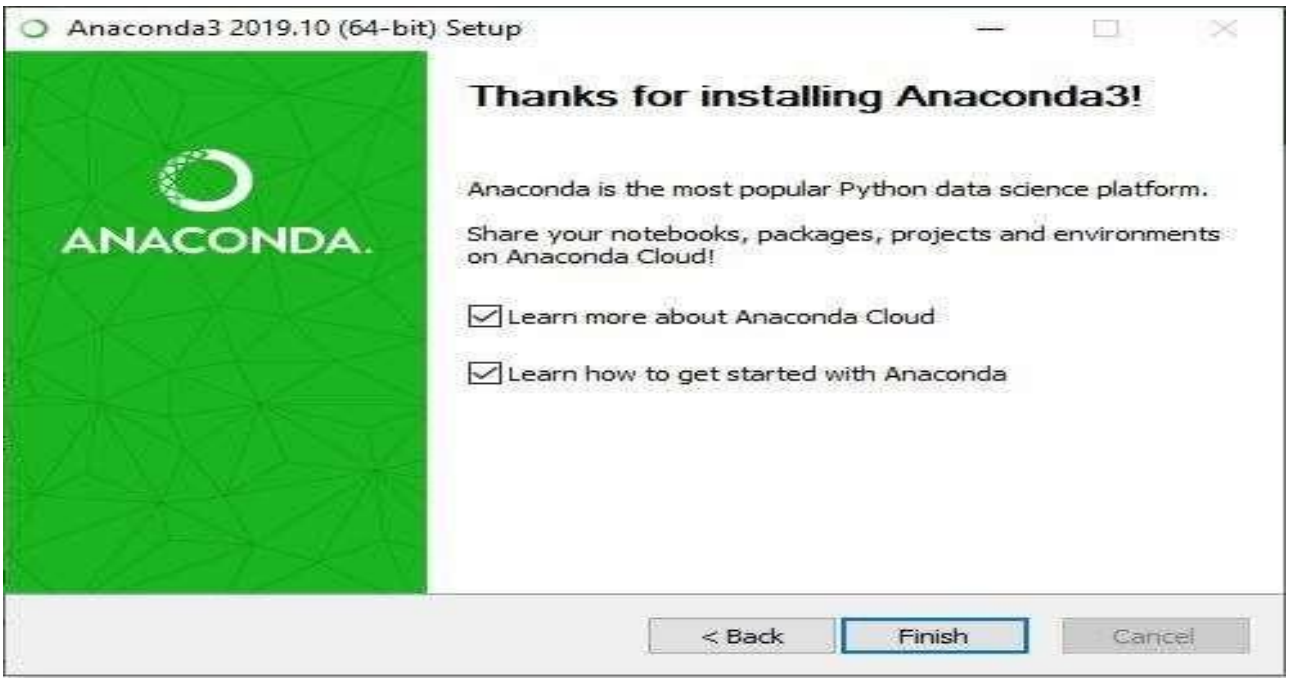
- **Getting through the Installation Process:**



- **Recommendation to Install Pycharm:**

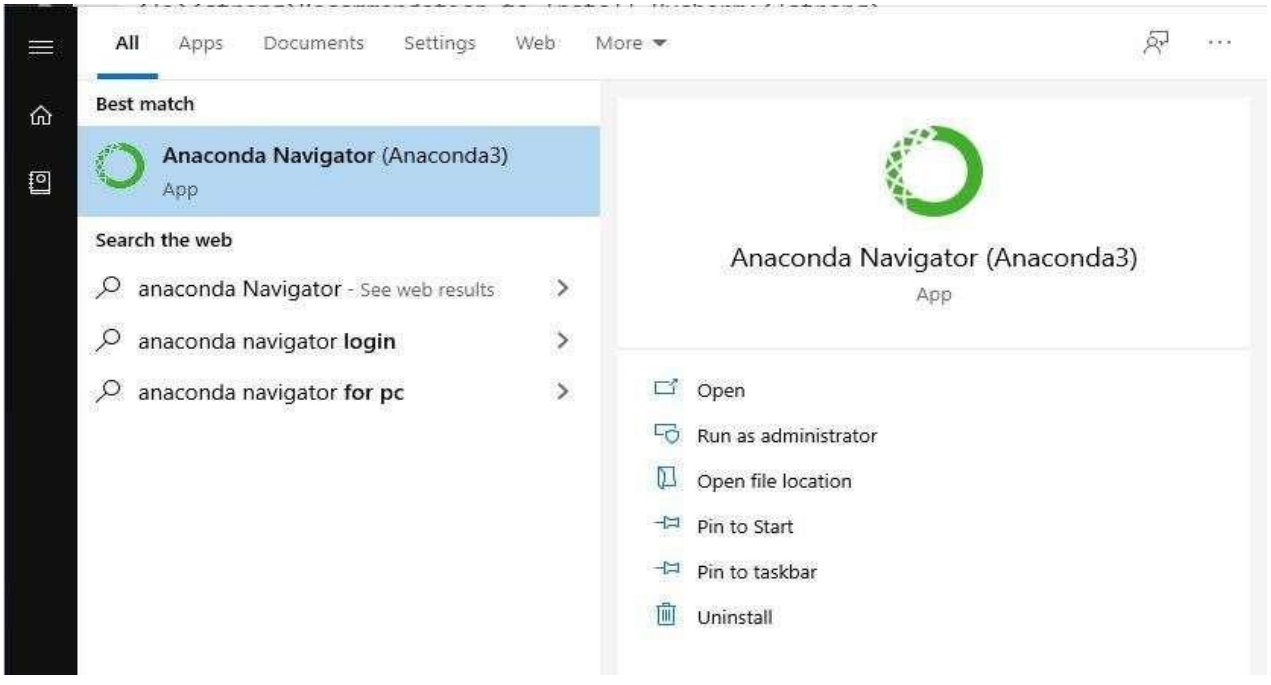


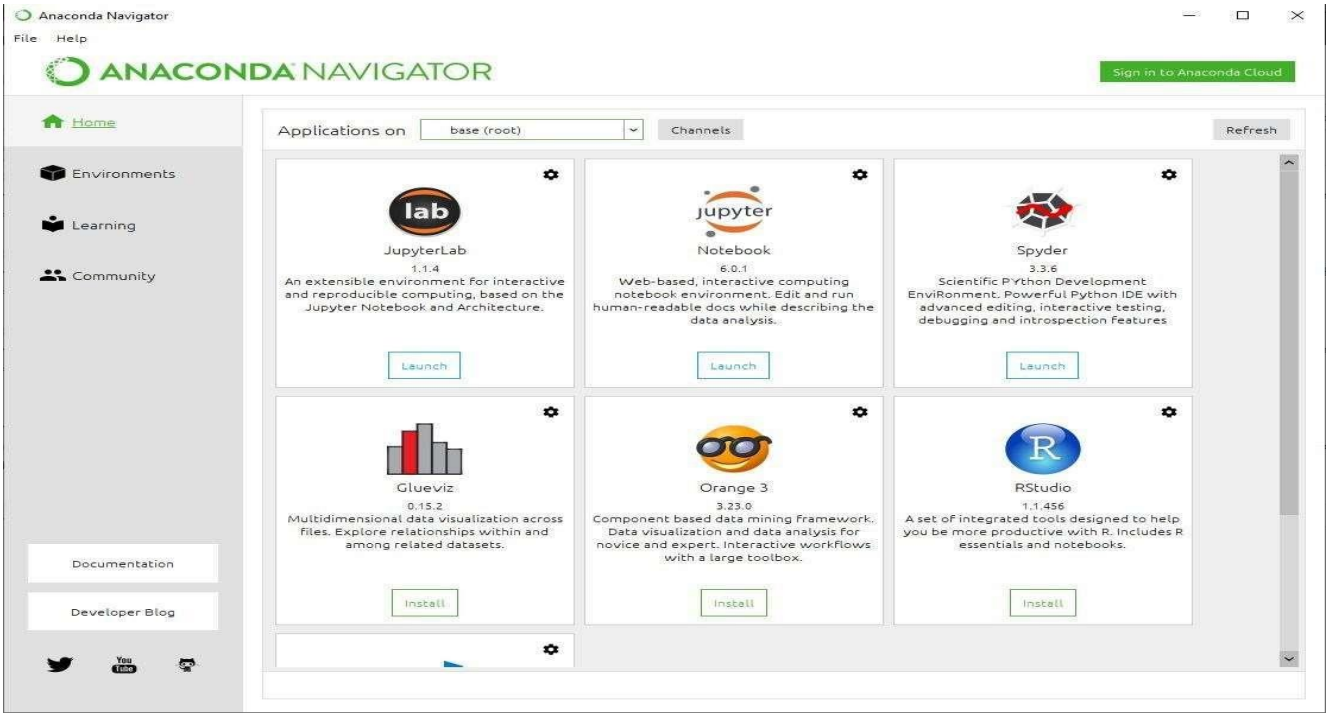
- **Finishing up the Installation:**



Woíking with Anaconda:

Once the installation process is done, Anaconda can be used to perform multiple operations. To begin using Anaconda, search for Anaconda Navigator from the StartMenu in Windows.





Conclusion:

In conclusion, Anaconda is an essential tool for data scientists and machine learning engineers. It comes with a wide range of pre-installed libraries, a package manager, cross-platform compatibility, Jupyter Notebook, and easy deployment, making it an ideal choice for machine learning projects.

Experiment:2

AIM:-

Supervised Learning – Regression: Generate a proper 2-D data set of N points. Split the data set into Training Data set and Test Data set.

a. Perform Linear Regression analysis with Least Squares Method.
b. Plot the graphs for Training MSE and Test MSE and comment on Curve Fitting and Generalization Error.
c. Verify the Effect of Data Set Size and Bias-Variance Trade off.
d. Apply Cross Validation and plot the graphs for errors.
e. Apply Subset Selection Method and plot the graphs for errors. Describe your findings in each case.

Solution:

Linear Regression:

- Linear Regression is a statistical modeling technique. But it is widely used in Machine Learning. In Linear Regression, we try to predict some output values (***y***) which are scalar values with the help of some input values (***x***).
- We call the input variables as independent variables. This is because these values do not depend on any other values. In the same sense, we call the output scalar values as dependent variables. This is because to find these output scalar values, we depend on the input variables.

Mathematical Representation of Linear Regression:

To know the representation of linear regression, we have to consider both the mathematical and machine learning aspects of the problem.
Say, we have a collection of labeled examples,

$$(\boldsymbol{x_i}, \boldsymbol{y_i})\boldsymbol{N_i} = \mathbf{1}$$

where **N** is the size or length of the dataset, $\mathbf{x_i}$ is the feature vector from **i =1,…, N, and** $\mathbf{y_i}$ is the real-valued target. For more clarity, you can call the collection of $\mathbf{x_i}$ as the independent variables and the collection of $\mathbf{y_i}$ as the dependent or target variables.If we simplify the above, then we can say that:

*We want to build a linear model with the combination of features in **x** and predict the scalar values in **y**.*

When we consider a simple linear regression, that is, a single ***x*** and a single ***y***, then we can write,

$$\boldsymbol{y} = \boldsymbol{\beta_0} + \boldsymbol{\beta_1 x}$$

where $\boldsymbol{\beta_0}$ is the intercept parameter and $\boldsymbol{\beta_1}$ is the slope parameter.
If we analyze the above equation, then we shall find that it is the same as the equation of a line:

$$\boldsymbol{y} = \boldsymbol{mx} + \boldsymbol{c}$$

where **m** is the slope and **c** is the y-intercept.For now, we just need to keep in mind that the above equations are for single-valued ***x***.

Types of Linear Regression:

- **Simple Linear Regression**
- **Multiple Linear Regression**

Simple Linear Regression:

- We know that in linear regression, we need to find out the dependent variables (***y***) by using the independent variables (***x***).
- In simple linear regression, for each observation **i =1,2,…,N**, there is only one $\mathbf{x_i}$ and one $\mathbf{y_i}$. This is the simplest form of linear regression.

Multiple Linear Regression:

In multiple linear regression, we have two or more independent variables **(x)** and one dependent variable **(y)**. So, instead of a single feature column, we have multiple feature columns and a target column.

Mean Squared Error Cost Function:

MSE is the average squared difference between the $\mathbf{y_i}$ values that we have from the labeled set and the predicted **y** new values. We have to minimize this average squared difference to get a good regression model on the given data.

The following is the formula for MSE:

$$\boldsymbol{MSE} = \sum_{\boldsymbol{i=1}}^{\boldsymbol{N}} (\boldsymbol{y_i} - (\boldsymbol{mx_i} + \boldsymbol{b}))^2$$

In the above equation, **N** is the number of labeled examples in the training dataset, $\mathbf{y_i}$ is the target value in the dataset, and $\mathbf{(mx_i+b)}$ is the predicted value of the target or you can say $\mathbf{y_{new}}$.
Minimizing the above cost function will mean finding the new target values as close as possible to the training target values. This will ensure that the linear model that we find will also lie close to the training examples.
To minimize the above cost function, we have to find the optimized values for the two parameters **m** and **b**. For that, we use the Gradient Descent method to find the partial derivatives with respect to **m** and **b**. In that case, the cost function looks something like the following:

$$\boldsymbol{f(m,b)} = \sum_{\boldsymbol{i=1}}^{\boldsymbol{n}} (\boldsymbol{y_i} - (\boldsymbol{mx_i} + \boldsymbol{b}))^2$$

Next, we use the chain rule to find the partial derivative of the above function. To solve the problem, we move through the data using updated values of slope and intercept and try to reduce the cost function as we do so. This part is purely mathematical and the machine learning implementation is obviously through coding.

a. Perform Linear Regression analysis with Least Squares Method.

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

In [2]:
```
df=pd.read_csv("values.csv",low_memory=False)
df.head()
```

Out[2]:

	5.1101	17.592
0	5.5277	9.1302
1	8.5186	13.6620
2	7.0032	11.8540
3	5.8598	6.8233
4	8.3829	11.8860

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

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 96 entries, 0 to 95
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   5.1101  96 non-null     float64
 1   17.592  96 non-null     float64
dtypes: float64(2)
memory usage: 1.6 KB
```

In [4]:
```
df.shape
```

Out[4]: (96, 2)

In [5]:
```
df.isnull().sum()
```

Out[5]:
```
5.1101    0
17.592    0
dtype: int64
```

In [6]:
```
x=df.iloc[:, 0]
y=df.iloc[:, 1]
x
```

Out[6]:
```
0      5.5277
1      8.5186
2      7.0032
3      5.8598
4      8.3829
       ...
91     5.8707
92     5.3054
93     8.2934
94    13.3940
95     5.4369
Name: 5.1101, Length: 96, dtype: float64
```

In [7]:
```
# Mean X and Y
mean_x = np.mean(x)
mean_y = np.mean(y)
```

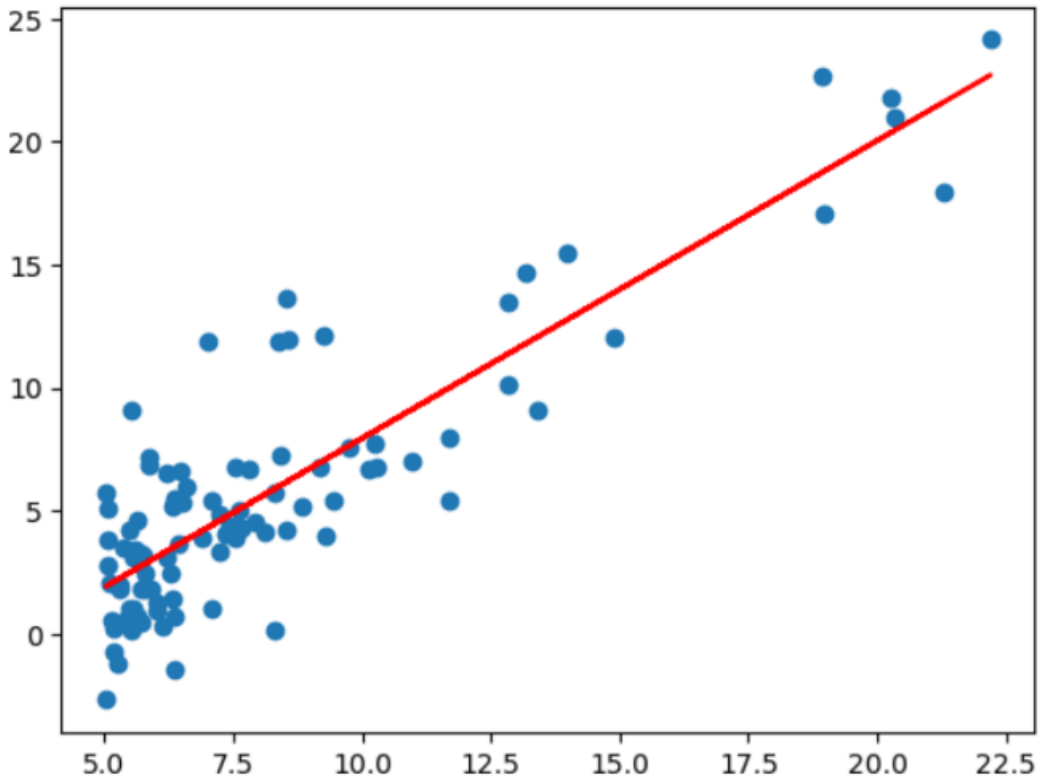
In [8]:
```
n = len(x)
numer = 0
denom = 0
for i in range(n):
    numer += (x[i] - mean_x) * (y[i] - mean_y)
    denom += (x[i] - mean_x) ** 2
m = numer / denom
c = mean_y - (m * mean_x)
```

In [9]:
```
print(m, c)
```

```
1.210073946912064 -4.150315520211127
```

In [10]:
```
Y=c + m*x
```

In [11]:
```
def myfunc(x):
  return m * x + c

mymodel = list(map(myfunc, x))

plt.scatter(x, y)
plt.plot(x, mymodel,color='r')
plt.show()
```



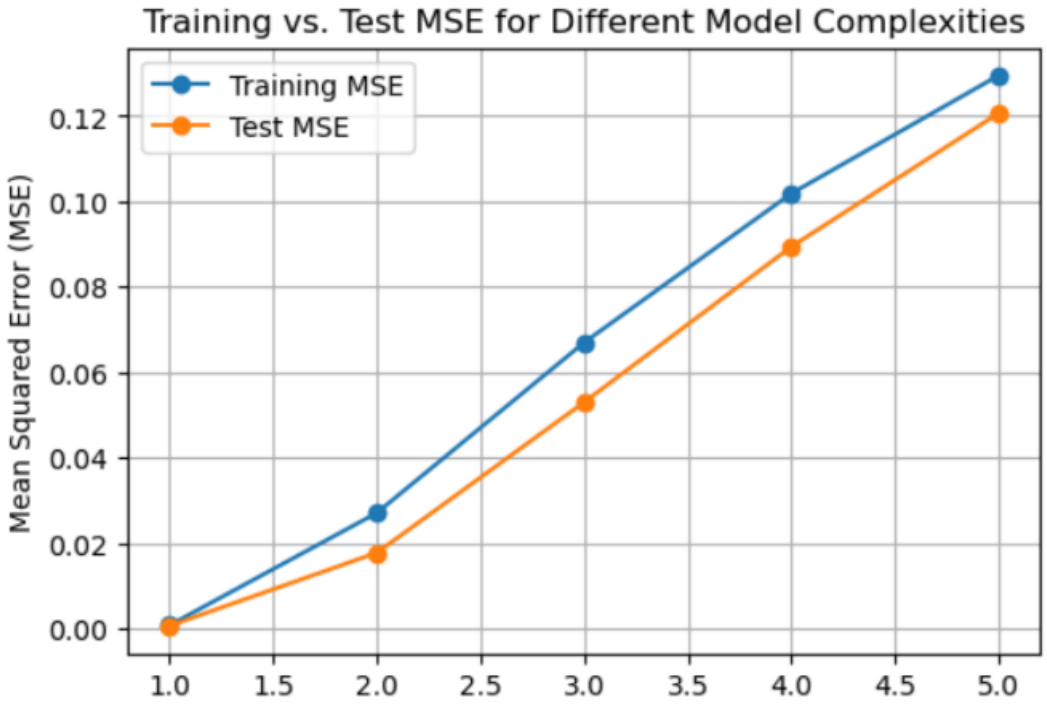
b. Plot the graphs for Training MSE and Test MSE and comment on Curve Fitting and Generalization Error.

In [12]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error


np.random.seed(0)
X = np.random.rand(100, 1)
y = 2 * X + 1 + 0.1 * np.random.rand(100, 1)
```

In [13]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, rando
train_mse = []
test_mse = []
```

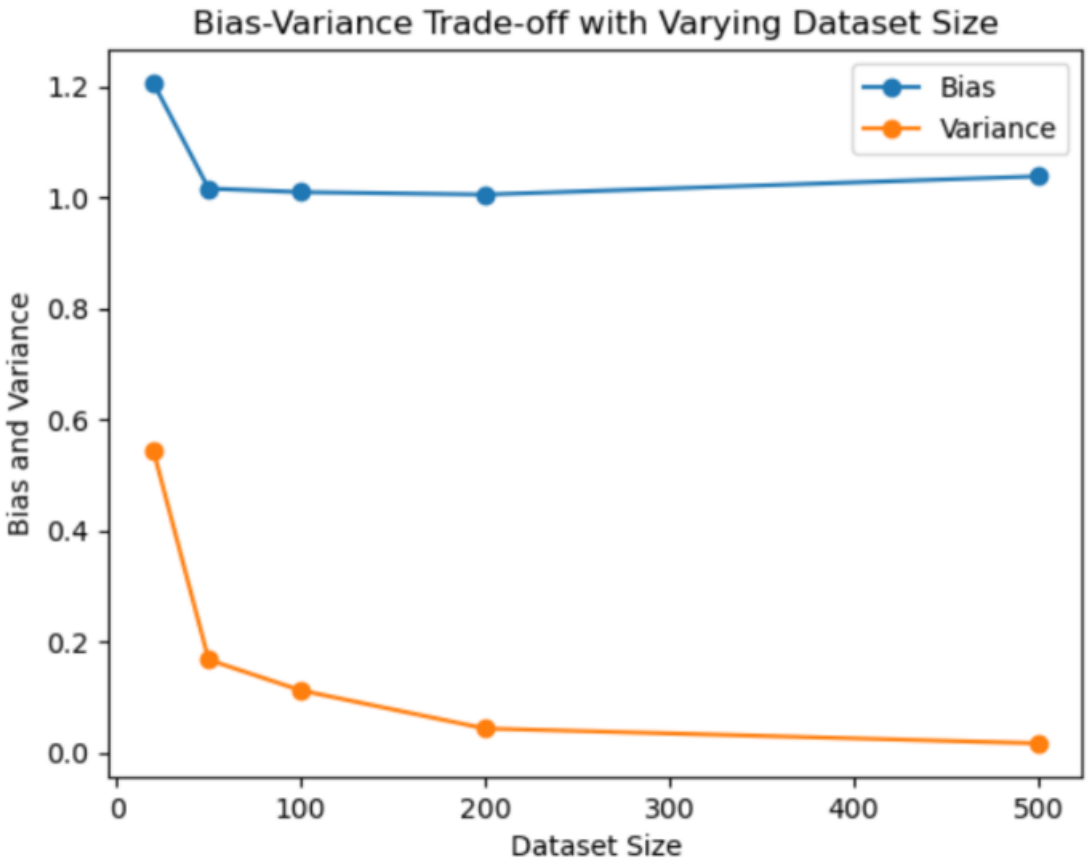
In [14]:
```
degrees = [1, 2, 3, 4, 5,]
for degree in degrees:

    X_train_poly = np.power(X_train, degree)
    X_test_poly = np.power(X_test, degree)


    model = LinearRegression()
    model.fit(X_train_poly, y_train)
    y_train_pred = model.predict(X_train_poly)
    y_test_pred = model.predict(X_test_poly)
    train_mse.append(mean_squared_error(y_train, y_train_pred))
    test_mse.append(mean_squared_error(y_test, y_test_pred))
```

In [15]:
```
plt.figure(figsize=(6, 4))
plt.plot(degrees, train_mse, label='Training MSE', marker='o')
plt.plot(degrees, test_mse, label='Test MSE', marker='o')
plt.xlabel('Model Complexity (Degree)')
plt.ylabel('Mean Squared Error (MSE)')
plt.title('Training vs. Test MSE for Different Model Complexities')
plt.legend()
plt.grid(True)
plt.show()
```



c. Verify the Effect of Data Set Size and Bias-Variance Trade off.

In [17]:
```
dataset_sizes = [20, 50, 100, 200, 500]
num_experiments = 100
bias_values = []
variance_values = []

for dataset_size in dataset_sizes:
    mse_values = []

    for _ in range(num_experiments):
        # Generate synthetic data with noise
        X = np.random.rand(dataset_size, 1)
        Y = 2 * X + 1 + np.random.randn(dataset_size, 1)

        # Split data into training and test sets
        split_index = int(0.8 * dataset_size)
        X_train, X_test = X[:split_index], X[split_index:]
        Y_train, Y_test = Y[:split_index], Y[split_index:]

        # Train a linear regression model
        model = LinearRegression()
        model.fit(X_train, Y_train)

        # Predict Y values
        Y_pred = model.predict(X_test)

        # Calculate Mean Squared Error (MSE)
        mse = mean_squared_error(Y_test, Y_pred)
        mse_values.append(mse)

    bias = np.mean(mse_values)
    variance = np.var(mse_values)
    bias_values.append(bias)
    variance_values.append(variance)


# Plot the results
plt.plot(dataset_sizes, bias_values, label='Bias', marker='o')
plt.plot(dataset_sizes, variance_values, label='Variance', marker='o')
plt.xlabel('Dataset Size')
plt.ylabel('Bias and Variance')
plt.legend()
plt.title('Bias-Variance Trade-off with Varying Dataset Size')
plt.show()
```



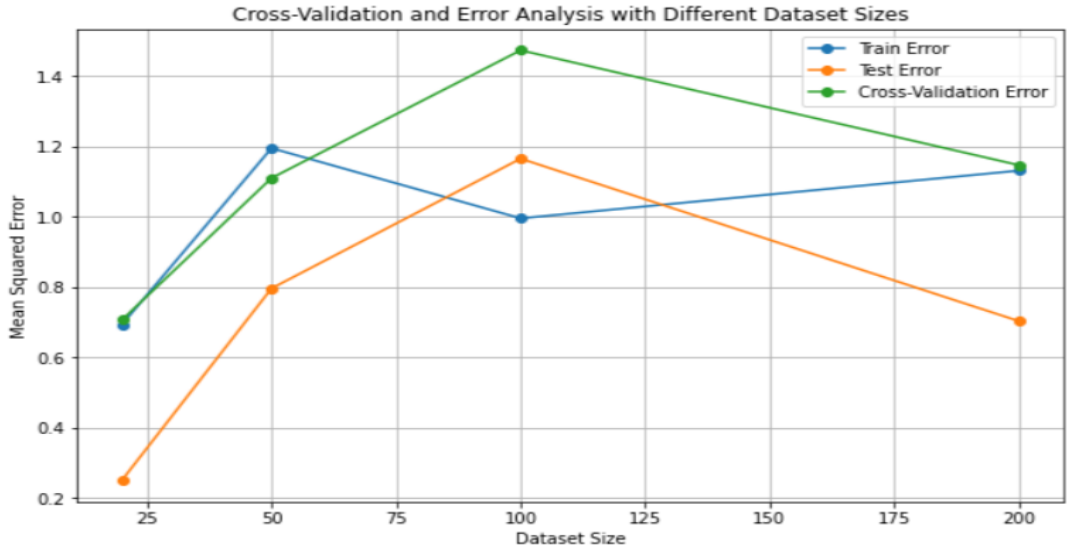
d. Apply Cross Validation and plot the graphs for errors.

Cross-Validation and Error Analysis with Different Dataset Sizes

```
from sklearn.model_selection import train_test_split, cross_val_score
# Generate synthetic data with a single linear relationship
def generate_data(size):
    np.random.seed(0)
    X = np.linspace(0, 10, size)
    Y = 2 * X + 1 + np.random.normal(0, 1, size)
    X = X.reshape(-1, 1)
    return X, Y

# Sizes of the datasets
sizes = [20, 50, 100, 200]

# Lists to store results
train_errors = []
test_errors = []
cross_val_errors = []
```

```python
# Train linear regression models for different dataset sizes and perform cross-validation
for size in sizes:
    X, Y = generate_data(size)

    # Split data into training and test sets
    X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2, random_state=42)

    # Create a linear regression model
    model = LinearRegression()
    model.fit(X_train, Y_train)

    # Calculate training error
    Y_train_pred = model.predict(X_train)
    train_error = mean_squared_error(Y_train, Y_train_pred)
    train_errors.append(train_error)

    # Calculate test error
    Y_test_pred = model.predict(X_test)
    test_error = mean_squared_error(Y_test, Y_test_pred)
    test_errors.append(test_error)

    # Perform k-fold cross-validation (e.g., k=5)
    cross_val_scores = cross_val_score(model, X, Y, cv=5, scoring='neg_mean_squared_error')
    cross_val_error = -cross_val_scores.mean()  # Negative MSE
    cross_val_errors.append(cross_val_error)
```

```python
# Plot training, test, and cross-validation errors for different dataset sizes
plt.figure(figsize=(10, 6))
plt.plot(sizes, train_errors, marker='o', label='Train Error')
plt.plot(sizes, test_errors, marker='o', label='Test Error')
plt.plot(sizes, cross_val_errors, marker='o', label='Cross-Validation Error')
plt.xlabel('Dataset Size')
plt.ylabel('Mean Squared Error')
plt.title('Cross-Validation and Error Analysis with Different Dataset Sizes')
plt.legend()
plt.grid(True)
plt.show()
```



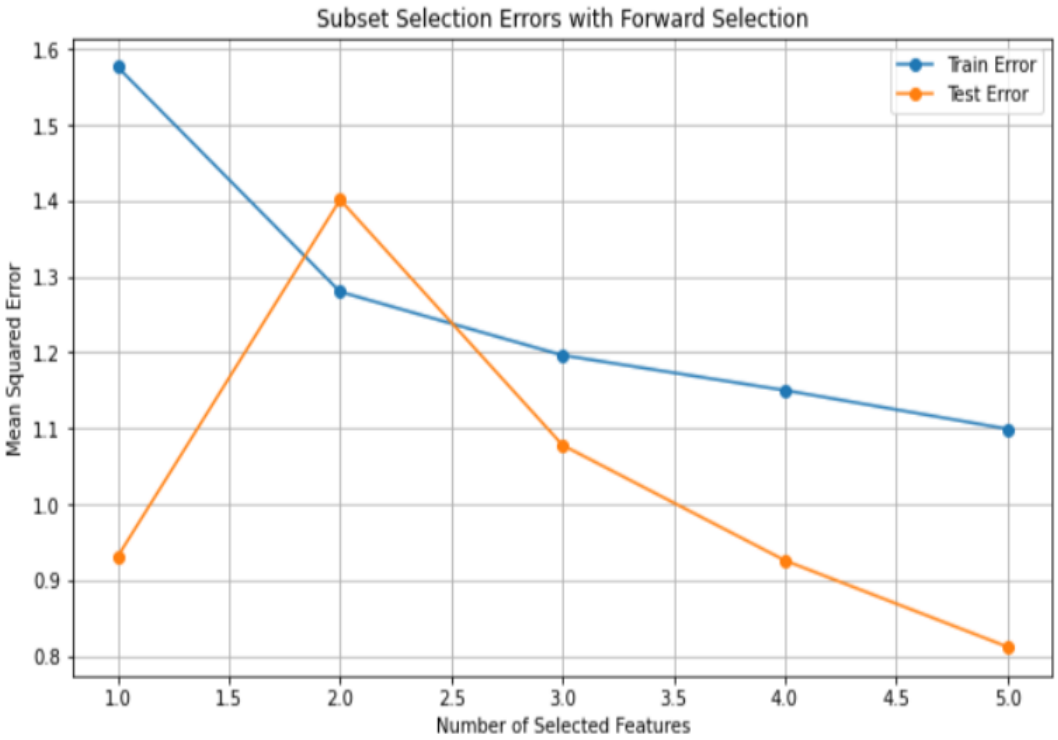
e. Apply Subset Selection Method and plot the graphs for errors. Describe your findings in each case.

Subset Selection Method

```
# Generate synthetic data with multiple features
def generate_data(size):
    np.random.seed(0)
    X = np.random.rand(size, 5)  # Five random features
    true_coefficients = np.array([2, -1, 0.5, 1.5, -0.5])
    Y = np.dot(X, true_coefficients) + np.random.normal(0, 1, size)
    return X, Y

# Size of the dataset
size = 100

# Generate synthetic data
X, Y = generate_data(size)

# Split data into training and test sets
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2, random_state=42)

# Lists to store results
train_errors = []
test_errors = []
selected_features = []
```

```
# Initialize an empty set for selected features
selected_feature_indices = set()

# Perform forward selection
for i in range(X.shape[1]):
    best_feature = None
    best_train_error = float('inf')

    # Try adding each feature that has not been selected yet
    for feature_index in range(X.shape[1]):
        if feature_index not in selected_feature_indices:
            # Include the current feature
            current_feature_indices = list(selected_feature_indices) + [feature_index]

            # Fit a linear regression model with the selected features
            model = LinearRegression()
            model.fit(X_train[:, current_feature_indices], Y_train)

            # Calculate training error
            Y_train_pred = model.predict(X_train[:, current_feature_indices])
            train_error = mean_squared_error(Y_train, Y_train_pred)

            # Update the best feature if necessary
            if train_error < best_train_error:
                best_feature = feature_index
                best_train_error = train_error
```

```
    # Add the best feature to the selected features
    selected_feature_indices.add(best_feature)
    selected_features.append(len(selected_feature_indices))

    # Fit a model with the selected features on the test data
    model = LinearRegression()
    model.fit(X_train[:, list(selected_feature_indices)], Y_train)

    # Calculate test error
    Y_test_pred = model.predict(X_test[:, list(selected_feature_indices)])
    test_error = mean_squared_error(Y_test, Y_test_pred)

    train_errors.append(best_train_error)
    test_errors.append(test_error)

# Plot the training and test errors as features are added
plt.figure(figsize=(10, 6))
plt.plot(selected_features, train_errors, marker='o', label='Train Error')
plt.plot(selected_features, test_errors, marker='o', label='Test Error')
plt.xlabel('Number of Selected Features')
plt.ylabel('Mean Squared Error')
plt.title('Subset Selection Errors with Forward Selection')
plt.legend()
plt.grid(True)
plt.show()
```



Experiment:3

AIM:-

Supervised Learning – Classification Implement Naïve Bayes Classifier and K-Nearest Neighbor Classifier on Data set of your choice. Test and Compare for Accuracy and Precision.

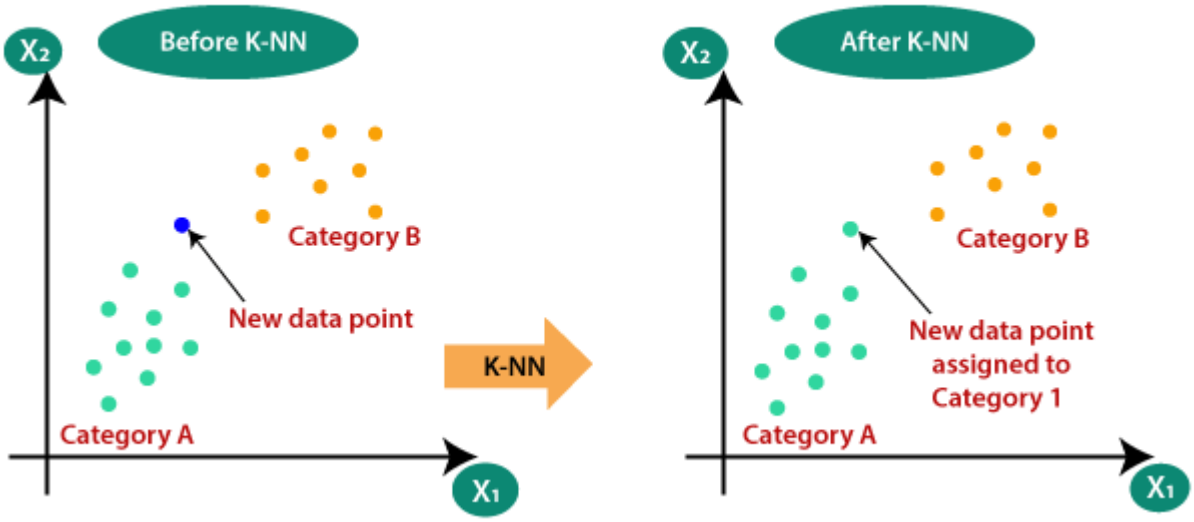
Solution:

K-Nearest Neighbor (KNN) Algorithm for Machine Learning

- K-NN algorithm assumes the similarity between the new case/data and available cases and put the new case into the category that is most similar to the available categories.
- K-NN algorithm stores all the available data and classifies a new data point based on the similarity. This means when new data appears then it can be easily classified into a well suite category by using K- NN algorithm.
- K-NN algorithm can be used for Regression as well as for Classification but mostly it is used for the Classification problems.
- K-NN is a **non-parametric algorithm**, which means it does not make any assumption on underlying data.
- It is also called a **lazy learner algorithm** because it does not learn from the training set immediately instead it stores the dataset and at the time of classification, it performs an action on the dataset.
- KNN algorithm at the training phase just stores the dataset and when it gets new data, then it classifies that data into a category that is much similar to the new data.
- **Example:** Suppose, we have an image of a creature that looks similar to cat and dog, but we want to know either it is a cat or dog. So, for this identification, we can use the KNN algorithm, as it works on a similarity measure. Our KNN model will find the similar features of the new data set to the cats and dogs' images and based on the most similar features it will put it in either cat or dog category.

Why do we need a K-NN Algorithm?

Suppose there are two categories, i.e., Category A and Category B, and we have a new data point x1, so this data point will lie in which of these categories. To solve this type of problem, we need a K-NN algorithm. With the help of K-NN, we can easily identify the category or class of a particular dataset. Consider the below diagram:

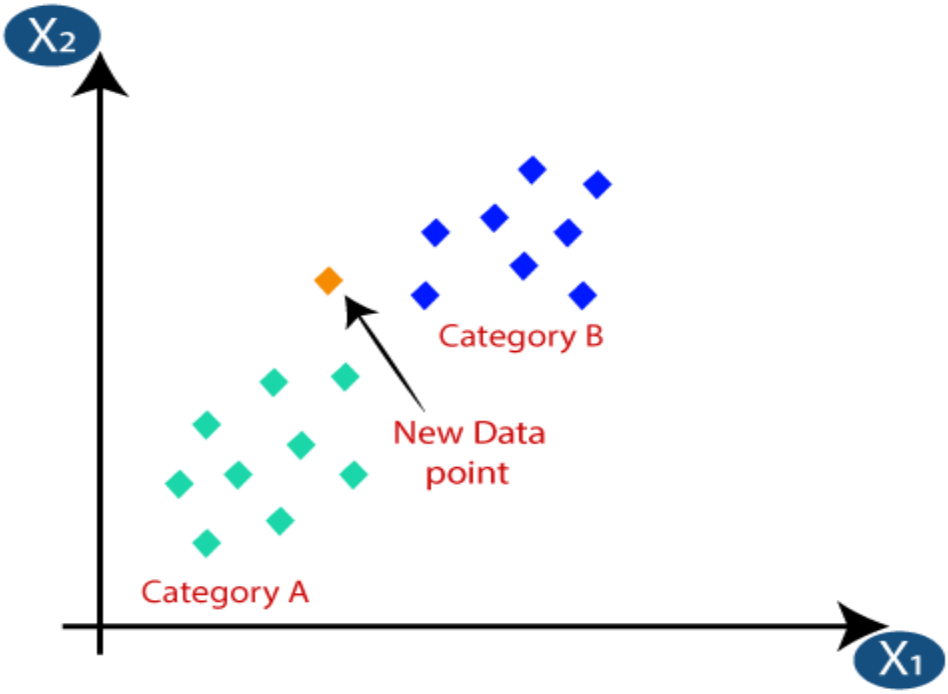


How does K-NN work?

The K-NN working can be explained on the basis of the below algorithm:

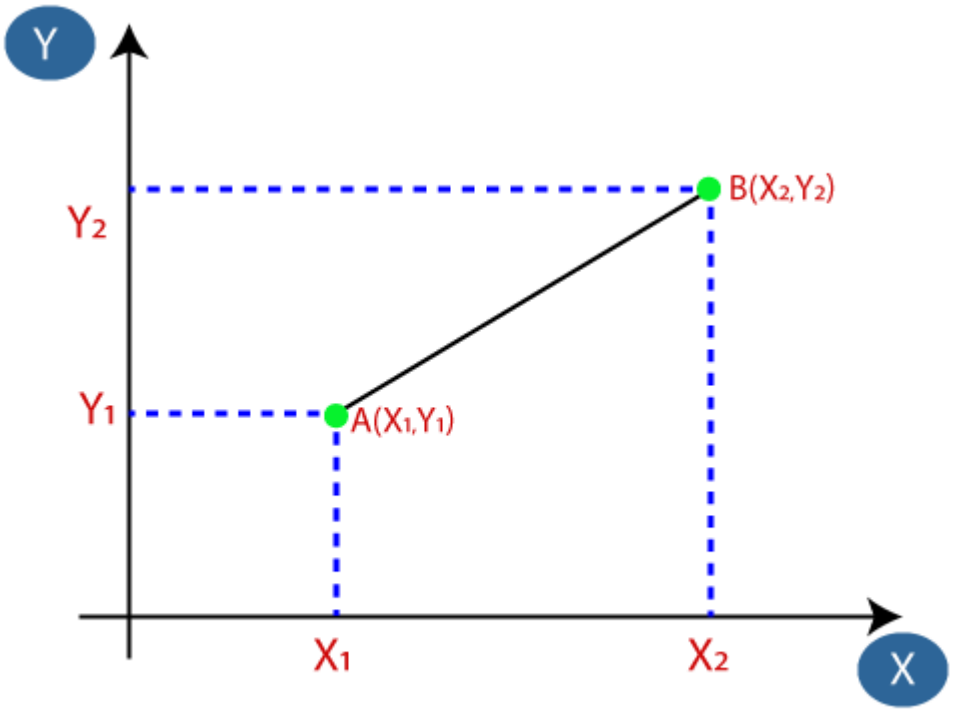
- **Step-1:** Select the number K of the neighbors
- **Step-2:** Calculate the Euclidean distance of **K number of neighbors**
- **Step-3:** Take the K nearest neighbors as per the calculated Euclidean distance.
- **Step-4:** Among these k neighbors, count the number of the data points in each category.
- **Step-5:** Assign the new data points to that category for which the number of the neighbor is maximum.
- **Step-6:** Our model is ready.

Suppose we have a new data point and we need to put it in the required category. Consider the below image:



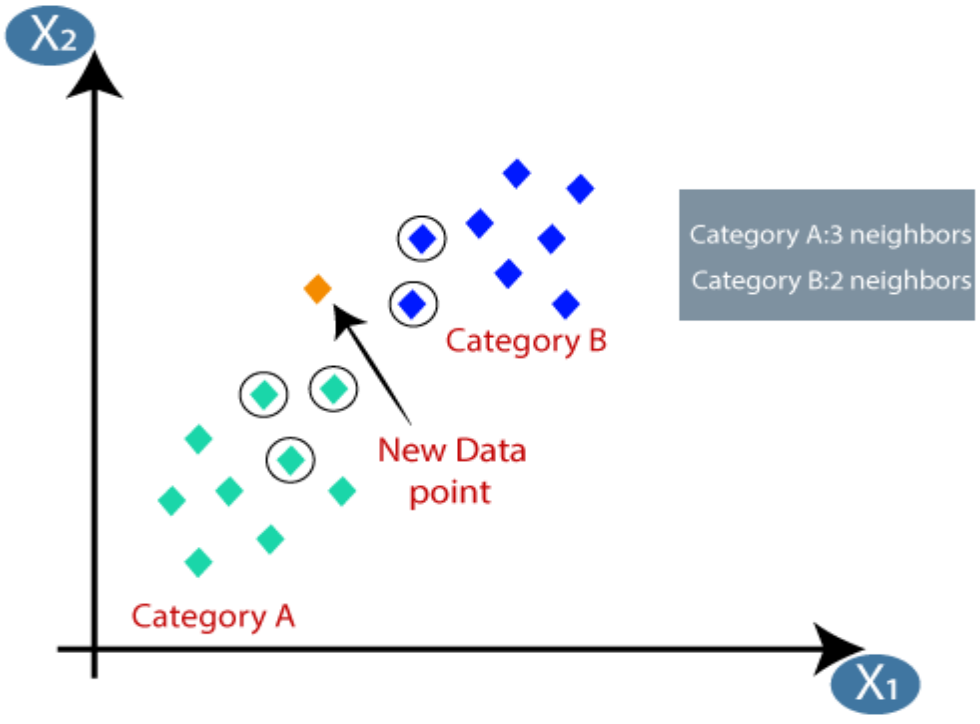
- Firstly, we will choose the number of neighbors, so we will choose the k=5.

- Next, we will calculate the **Euclidean distance** between the data points. The Euclidean distance is the distance between two points, which we have already studied in geometry. It can be calculated as:



$$\text{Euclidean Distance between } A_1 \text{ and } B_2 = \sqrt{(X_2-X_1)^2+(Y_2-Y_1)^2}$$

- By calculating the Euclidean distance, we got the nearest neighbors, as three nearest neighbors in category A and two nearest neighbors in category B. Consider the below image:



- As we can see the 3 nearest neighbors are from category A, hence this new data point must belong to category A.

Naïve Bayes Classifiers

Naive Bayes classifiers are a collection of classification algorithms based on **Bayes' Theorem**. It is not a single algorithm but a family of algorithms where all of them share a common principle, i.e. every pair of features being classified is independent of each other.
To start with, let us consider a dataset.
Consider a fictional dataset that describes the weather conditions for playing a game of golf. Given the weather conditions, each tuple classifies the conditions as fit("Yes") or unfit("No") for plaing golf.
Here is a tabular representation of our dataset.

	Outlook	Temperature	Humidity	Windy	Play Golf
0	Rainy	Hot	High	False	No
1	Rainy	Hot	High	True	No
2	Overcast	Hot	High	False	Yes
3	Sunny	Mild	High	False	Yes
4	Sunny	Cool	Normal	False	Yes
5	Sunny	Cool	Normal	True	No
6	Overcast	Cool	Normal	True	Yes
7	Rainy	Mild	High	False	No
8	Rainy	Cool	Normal	False	Yes
9	Sunny	Mild	Normal	False	Yes
10	Rainy	Mild	Normal	True	Yes
11	Overcast	Mild	High	True	Yes
12	Overcast	Hot	Normal	False	Yes
13	Sunny	Mild	High	True	No

The dataset is divided into two parts, namely, **feature matrix** and the **response vector**.

- Feature matrix contains all the vectors(rows) of dataset in which each vector consists of the value of **dependent features**. In above dataset, features are 'Outlook', 'Temperature', 'Humidity' and 'Windy'.
- Response vector contains the value of **class variable** (prediction or output) for each row of feature matrix. In above dataset, the class variable name is 'Play golf'.

Now, before moving to the formula for Naive Bayes, it is important to know about Bayes' theorem.

Bayes' Theorem:-

Bayes' Theorem finds the probability of an event occurring given the probability of another event that has already occurred. Bayes' theorem is stated mathematically as the following equation:

$$P(A|B) = \frac{P(B|A) \, . \, P(A)}{P(B)}$$

where A and B are events and P(B)? 0.

- Basically, we are trying to find probability of event A, given the event B is true. Event B is also termed as **evidence**.
- P(A) is the **priori** of A (the prior probability, i.e. Probability of event before evidence is seen). The evidence is an attribute value of an unknown instance (here, it is event B).
- P(A|B) is a posteriori probability of B, i.e. probability of event after evidence is seen.

Now, with regards to our dataset, we can apply Bayes' theorem in following way:

$$P(y|X) = \frac{P(X|y) \, . \, P(y)}{P(X)}$$

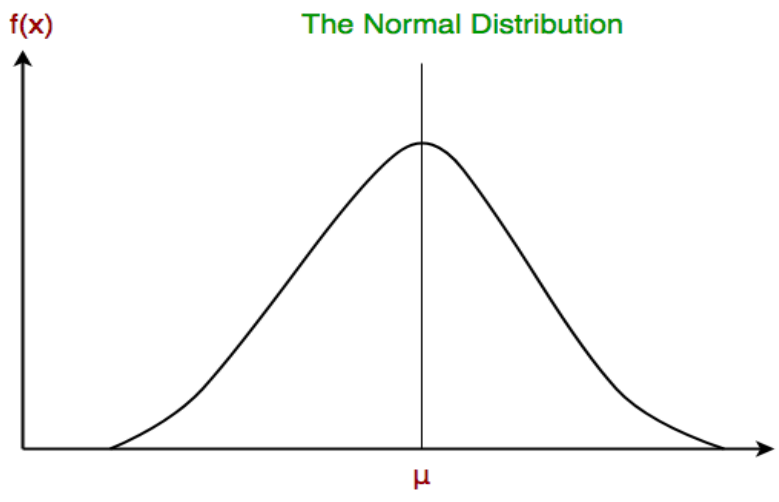
where, y is class variable and X is a dependent feature vector (of size *n*) where:

$$X = \{x_1, x_2, x_3, x_4, \ldots x_n\}$$

Just to clear, an example of a feature vector and corresponding class variable can be: (refer 1st row of dataset)
X = (Rainy, Hot, High, False)
y = No
So basically, P(y|X) here means, the probability of "Not playing golf" given that the weather conditions are "Rainy outlook", "Temperature is hot", "high humidity" and "no wind".

Gaussian Naive Bayes classifier:-

In Gaussian Naive Bayes, continuous values associated with each feature are assumed to be distributed according to a **Gaussian distribution**. A Gaussian distribution is also called Normal distribution. When plotted, it gives a bell-shaped curve which is symmetric about the mean of the feature values as shown below:



Implementation of Naïve Bayes Classifier and K-Nearest Neighbor Classifier:-

In [1]:
```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, precision_score
```

In [2]:
```
data=pd.read_csv('IMDB Dataset.csv')
print(data.shape)
data.head()
```

```
(50000, 2)
```

Out[2]:

	review	sentiment
0	One of the other reviewers has mentioned that ...	positive
1	A wonderful little production. The...	positive
2	I thought this was a wonderful way to spend ti...	positive
3	Basically there's a family where a little boy ...	negative
4	Petter Mattei's "Love in the Time of Money" is...	positive

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

Out[3]:

	review	sentiment
count	50000	50000
unique	49582	2
top	Loved today's show!!! It was a variety and not...	positive
freq	5	25000

In [4]:
```
data['sentiment'].value_counts()
```

Out[4]:
```
positive    25000
negative    25000
Name: sentiment, dtype: int64
```

In [5]:
```
X = data['review']
y = data['sentiment']
```

In [6]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, rando
```

In [7]:
```
vectorizer = CountVectorizer()
X_train_vec = vectorizer.fit_transform(X_train)
X_test_vec = vectorizer.transform(X_test)
```

In [8]:
```
nb_classifier = MultinomialNB()
nb_classifier.fit(X_train_vec, y_train)

# Predict on the test data
nb_predictions = nb_classifier.predict(X_test_vec)

# Calculate accuracy and precision for Naïve Bayes
nb_accuracy = accuracy_score(y_test, nb_predictions)
nb_precision = precision_score(y_test, nb_predictions, average='weighted')
```

In [9]:
```
knn_classifier = KNeighborsClassifier(n_neighbors=5)
knn_classifier.fit(X_train_vec, y_train)

# Predict on the test data
knn_predictions = knn_classifier.predict(X_test_vec)

# Calculate accuracy and precision for KNN
knn_accuracy = accuracy_score(y_test, knn_predictions)
knn_precision = precision_score(y_test, knn_predictions, average='weighted')
```

In [10]:
```
print("Naïve Bayes Classifier:")
print(f"Accuracy: {nb_accuracy:.2f}")
print(f"Precision: {nb_precision:.2f}")

print("\nK-Nearest Neighbors (KNN) Classifier:")
print(f"Accuracy: {knn_accuracy:.2f}")
print(f"Precision: {knn_precision:.2f}")
```

```
Naïve Bayes Classifier:
Accuracy: 0.85
Precision: 0.85

K-Nearest Neighbors (KNN) Classifier:
Accuracy: 0.64
Precision: 0.65
```

Experiment:4

AIM:-

Unsupervised Learning Implement K-Means Clustering and Hierarchical Clustering on proper data set of your choice. Compare their Convergence.

Solution:

K-Means Clustering Algorithm

K-Means Clustering is an unsupervised learning algorithm that is used to solve the clustering problems in machine learning or data science. In this topic, we will learn what is K-means clustering algorithm, how the algorithm works, along with the Python implementation of k-means clustering.

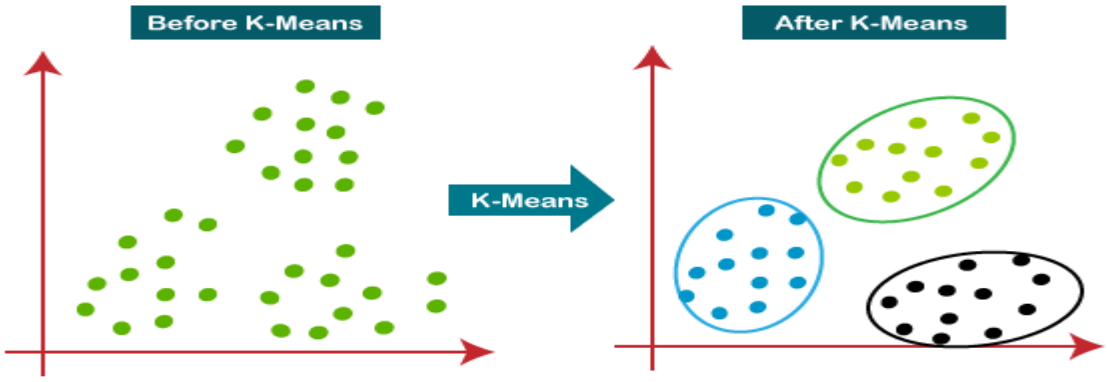
What is K-Means Algorithm?

- K-Means Clustering is an Unsupervised Learning algorithm, which groups the unlabeled dataset into different clusters. Here K defines the number of pre-defined clusters that need to be created in the process, as if K=2, there will be two clusters, and for K=3, there will be three clusters, and so on.
- It is an iterative algorithm that divides the unlabeled dataset into k different clusters in such a way that each dataset belongs only one group that has similar properties.
- It allows us to cluster the data into different groups and a convenient way to discover the categories of groups in the unlabeled dataset on its own without the need for any training.
- It is a centroid-based algorithm, where each cluster is associated with a centroid. The main aim of this algorithm is to minimize the sum of distances between the data point and their corresponding clusters.

The k-means clustering algorithm mainly performs two tasks:

- Determines the best value for K center points or centroids by an iterative process.
- Assigns each data point to its closest k-center. Those data points which are near to the particular k-center, create a cluster.

Hence each cluster has datapoints with some commonalities, and it is away from other clusters.
The below diagram explains the working of the K-means Clustering Algorithm:



IMPLEMENTATION:-

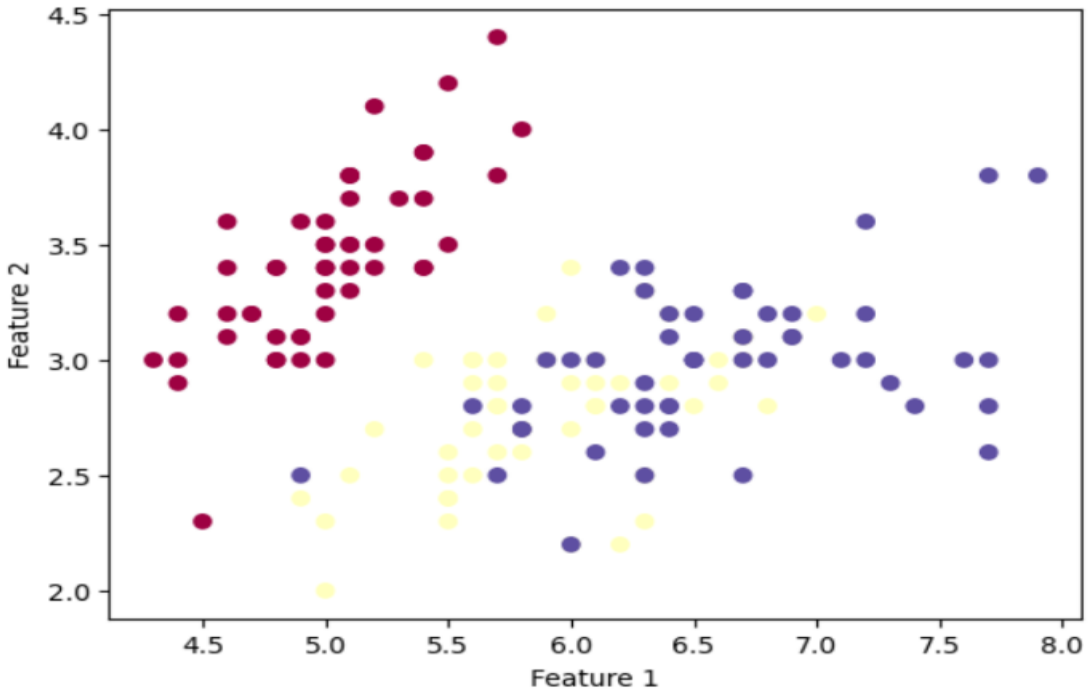
Step 1: Import Necessary Libraries

In [1]:
```
import numpy as np
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
import matplotlib.pyplot as plt
```

Step 2: Load and Visualize the Dataset

In [9]:
```
# Load a non-separable dataset (e.g., Iris)
iris = datasets.load_iris()
X = iris.data[:, :2]  # Use only the first two features
y = iris.target
```

In [10]:
```
# Visualize the dataset
plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Spectral)
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.show()
```



Step 3: Train-Test Split

In [3]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, rando
```

Step 4: SVM Model Selection

In [4]:
```
# Create an SVM classifier with an RBF kernel
clf = SVC(kernel='rbf', C=1.0)
```

Step 5: Model Training

In [5]:
```
clf.fit(X_train, y_train)
```

Out[5]:

▾ SVC

SVC()

Step 7: Model Evaluation

In [6]:
```
accuracy = clf.score(X_test, y_test)
print(f"Accuracy: {accuracy}")
```

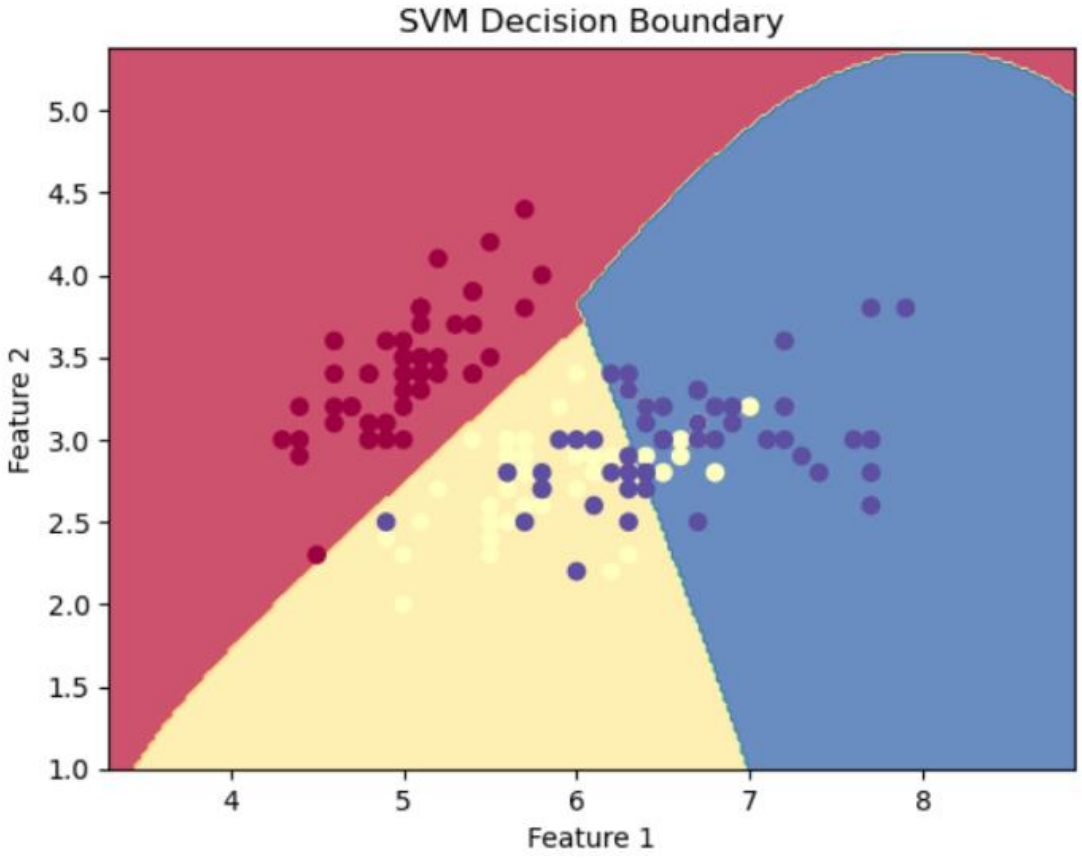
```
Accuracy: 0.9
```

Step 7: Visualization

In [7]:
```
# Create a mesh to plot decision boundaries
h = .02  # Step size in the mesh
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
```

```
# Predict the decision boundary
Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
```

```
# Scatter plot of the dataset
plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Spectral)
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.title('SVM Decision Boundary')
plt.show()
```



Experiment:5

AIM:-

Dimensionality Reduction Principal Component Analysis-Finding Principal Components, Variance and Standard Deviation calculations of principal components.

Solution:

Principal Component Analysis is basically a statistical procedure to convert a set of observation of possibly correlated variables into a set of values of linearly uncorrelated variables. Each of the principal components is chosen in such a way so that it would describe most of the still available variance and all these principal components are orthogonal to each other. In all principal components first, principal component has maximum variance.

Uses of PCA:

- It is used to find inter-relation between variables in the data.
- It is used to interpret and visualize data.
- As number of variables are decreasing it makes further analysis simpler.
- It's often used to visualize genetic distance and relatedness between populations.

These are basically performed on square symmetric matrix. It can be a pure sums of squares and cross products matrix or Covariance matrix or Correlation matrix. A correlation matrix is used if the individual variance differs much.

Objectives of PCA:

- It is basically a non-dependent procedure in which it reduces attribute space from a large number of variables to a smaller number of factors.
- PCA is basically a dimension reduction process but there is no guarantee that the dimension is interpretable.
- Main task in this PCA is to select a subset of variables from a larger set, based on which original variables have the highest correlation with the principal amount.

Principal Axis Method:

PCA basically search a linear combination of variables so that we can extract maximum variance from the variables. Once this process completes it removes it and search for another linear combination which gives an explanation about the maximum proportion of remaining variance which basically leads to orthogonal factors. In this method, we analyze total variance.

Eigenvector:

It is a non-zero vector that stays parallel after matrix multiplication. Let's suppose x is eigen vector of dimension r of matrix M with dimension r*r if Mx and x are parallel. Then we need to solve Mx=Ax where both x and A are unknown to get eigen vector and eigen values.
Under Eigen-Vectors we can say that Principal components show both common and unique variance of the variable. Basically, it is variance focused approach seeking to reproduce total variance and correlation with all components. The principal components are basically the linear combinations of the original variables

weighted by their contribution to explain the variance in a particular orthogonal dimension.

Eigen Values:

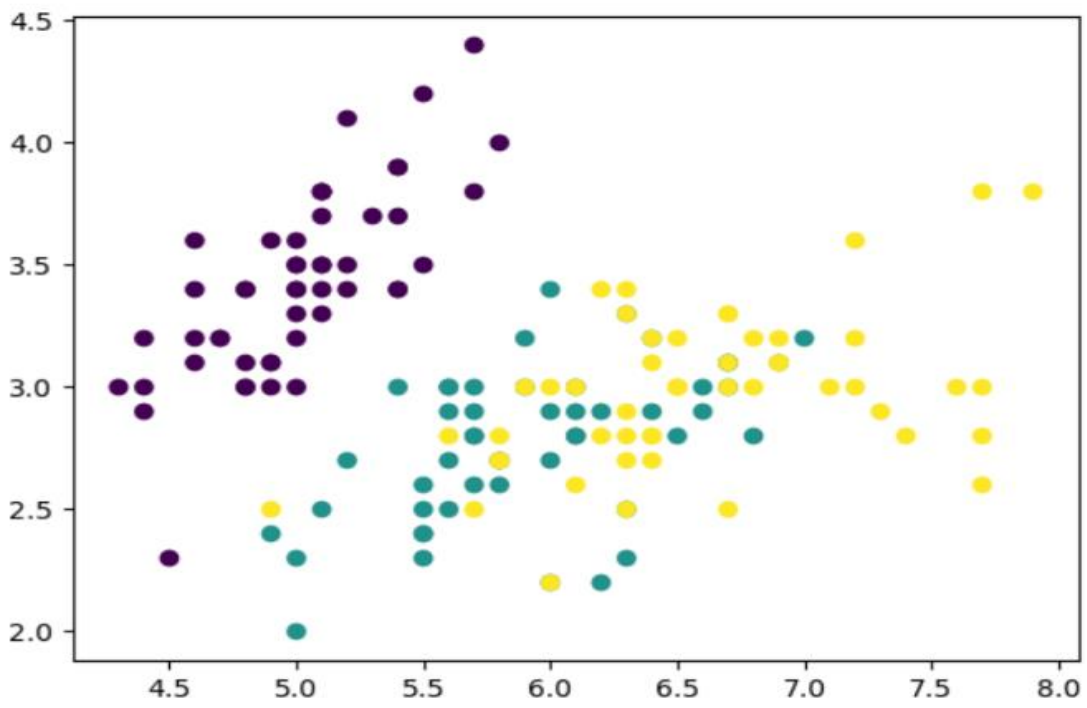
It is basically known as characteristic roots. It basically measures the variance in all variables which is accounted for by that factor. The ratio of eigenvalues is the ratio of explanatory importance of the factors with respect to the variables. If the factor is low then it is contributing less in explanation of variables. In simple words, it measures the amount of variance in the total given database accounted by the factor. We can calculate the factor's eigen value as the sum of its squared factor loading for all the variables.

1. Import Libraries:

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
```

2. Load Your Dataset:

In [33]:
```
iris = datasets.load_iris()
X = iris.data[:, :2]  # Use only the first two features
y = iris.target
plt.scatter(X[:, 0], X[:, 1], c=y)
plt.show()
```



3. Standardize the Data:

In [4]:
```
scaler = StandardScaler()
X_std = scaler.fit_transform(X)
```

4. Fit PCA:

In [5]:
```
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_std)
```

5. Mean of the original data:

In [6]:
```
original_mean = np.mean(X_std, axis=0)
print("Original Data Mean:\n", original_mean)
```

```
Original Data Mean:
 [-1.69031455e-15 -1.84297022e-15]
```

6. Covariance of the original data:

In [7]:
```
original_covariance = np.cov(X_std, rowvar=False)
print("Original Data Covariance:\n", original_covariance)
```

```
Original Data Covariance:
 [[ 1.00671141 -0.11835884]
 [-0.11835884  1.00671141]]
```

7. Mean of the PCA-transformed data:

In [8]:
```
pca_mean = np.mean(X_pca, axis=0)
print("PCA Data Mean:\n", pca_mean)
```

```
PCA Data Mean:
 [9.47390314e-17 1.53950926e-16]
```

8. Covariance of the PCA-transformed data:

In [9]:
```
pca_covariance = np.cov(X_pca, rowvar=False)
print("PCA Data Covariance:\n", pca_covariance)
```

```
PCA Data Covariance:
 [[1.12507025e+00 3.38020765e-16]
 [3.38020765e-16 8.88352566e-01]]
```

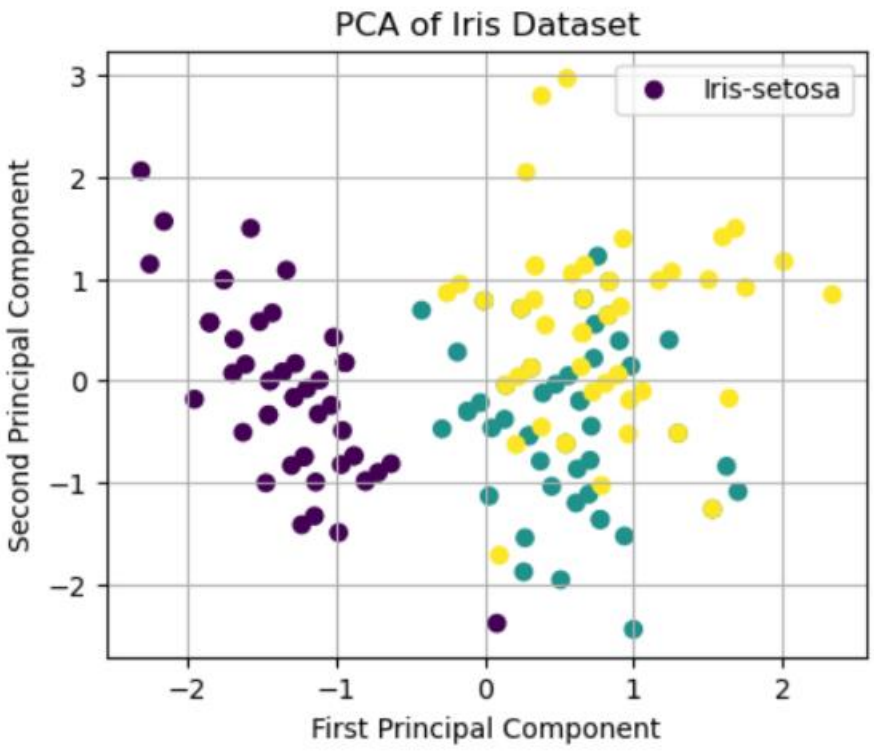
9. Explained Variance Ratio:

In [10]:
```
explained_variance = pca.explained_variance_ratio_
print("Explained Variance Ratio:", explained_variance)
```

```
Explained Variance Ratio: [0.55878489 0.44121511]
```

10. Visualize the Data:

In [32]:
```
plt.figure(figsize=(5, 4))
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=iris.target, cmap='viridis')
targets = ['Iris-setosa', 'Iris-versicolor', 'Iris-virginica']
colors = ['r', 'g', 'b']
plt.xlabel('First Principal Component')
plt.ylabel('Second Principal Component')
plt.title('PCA of Iris Dataset')
plt.legend(targets)
plt.grid()
plt.show()
```



Experiment:6

AIM:-

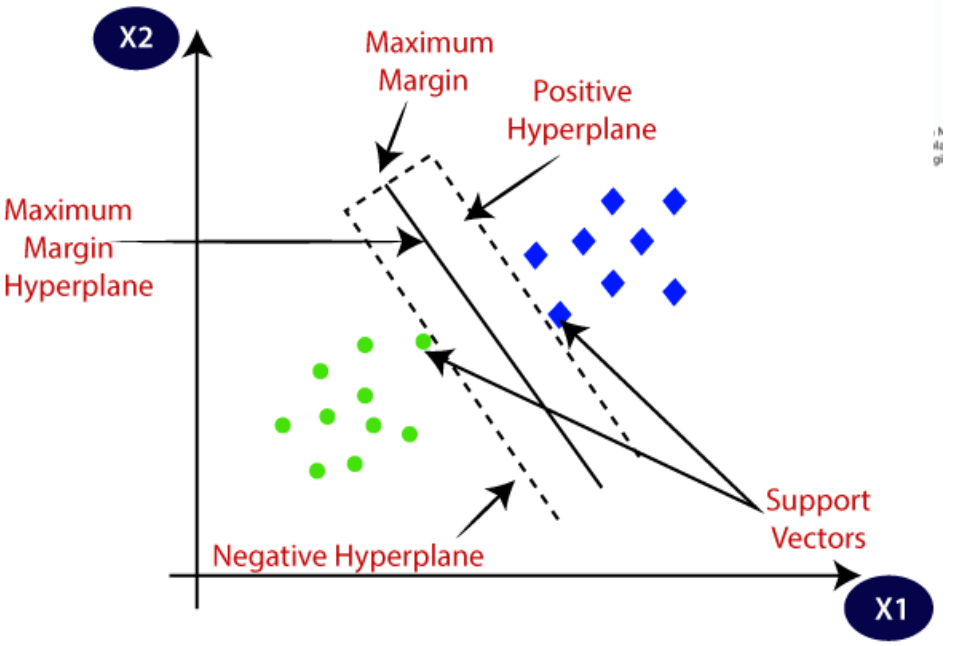
Supervised Learning and Kernel Methods Design, Implement SVM for classification with proper data set of your choice. Comment on Design and Implementation for Linearly non-separable Dataset.

Solution:

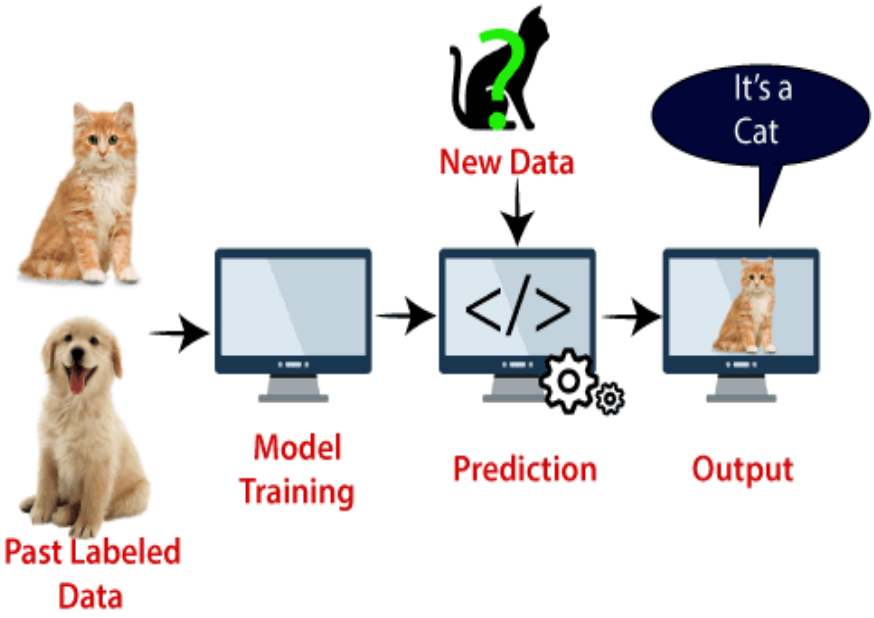
Support Vector Machine or SVM is one of the most popular Supervised Learning algorithms, which is used for Classification as well as Regression problems. However, primarily, it is used for Classification problems in Machine Learning.

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

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



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



SVM algorithm can be used for **Face detection, image classification, text categorization,** etc.

Types of SVM:-

SVM can be of two types:

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

Hyperplane and Support Vectors in the SVM algorithm:

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

The dimensions of the hyperplane depend on the features present in the dataset, which means if there are 2 features (as shown in image), then hyperplane will be a straight line. And if there are 3 features, then hyperplane will be a 2-dimension plane.

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

Support Vectors:

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

Different Kernel Functions

Kernels play a crucial role in Support Vector Machines (SVMs) by transforming the input data into a higher-dimensional space, making it easier to find a separating hyperplane. Different kernels capture different types of relationships in the data. Here are some common kernels used in SVMs:

1. Linear Kernel:

$$K(x, y) = x^T \cdot y$$

It represents a linear relationship between features. The decision boundary is a hyperplane in the original feature space. It can be used for text classification, where features represent word frequencies.

2. Polynomial Kernel:

$$K(x, y) = (x^T \cdot y + c)^d$$

- It introduces non-linearity through polynomial transformations of the input features.
- Apply when the decision boundary is expected to be a polynomial curve.
- It can be used for image recognition, where pixel values are transformed to capture non-linear relationships.

3. Radial Basis Function (RBF) or Gaussian Kernel:

$$K(x, y) = \exp\left(-\frac{\|x-y\|^2}{2\sigma^2}\right)$$

- It models complex, non-linear relationships by transforming data into an infinite-dimensional space.
- Suitable for datasets with circular or complex-shaped clusters.
- It can be used for financial forecasting, where the RBF kernel captures patterns in historical stock prices.

4. Sigmoid Kernel:

$$K(x, y) = \tanh(\alpha x^T \cdot y + c)$$

- It is similar to the hyperbolic tangent function and can handle non-linear relationships.

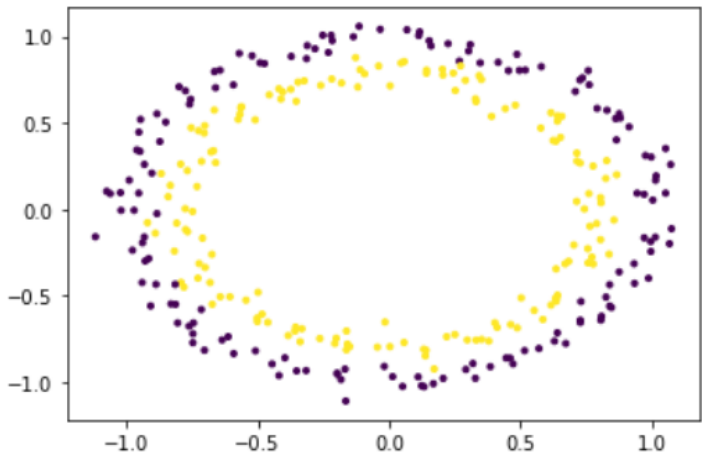
IMPLEMENTATION:-

Non-Linear SVM

In [12]:
```
# Importing Libraries

import matplotlib.pyplot as plt
import numpy as np
from sklearn import datasets
from sklearn import svm
```

In [13]:
```
# non-linear data
circle_X, circle_y = datasets.make_circles(n_samples=300, noise=0.05)
```

In [14]:
```
# show raw non-linear data
plt.scatter(circle_X[:, 0], circle_X[:, 1], c=circle_y, marker='.')
plt.show()
```



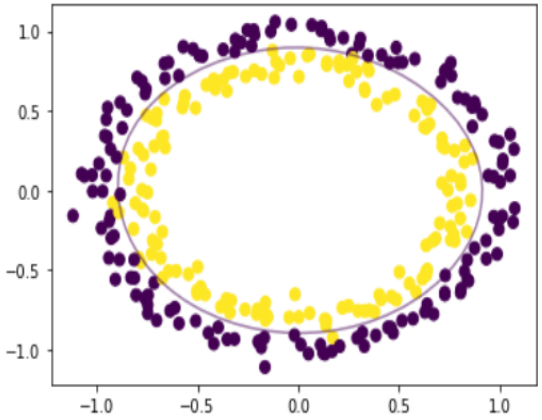
In [15]:
```
# make non-linear algorithm for model & using RBF
nonlinear_clf = svm.SVC(kernel='rbf', C=1.0)
```

In [16]:
```
# training non-linear model
nonlinear_clf.fit(circle_X, circle_y)
```

Out[16]:
```
SVC()
```

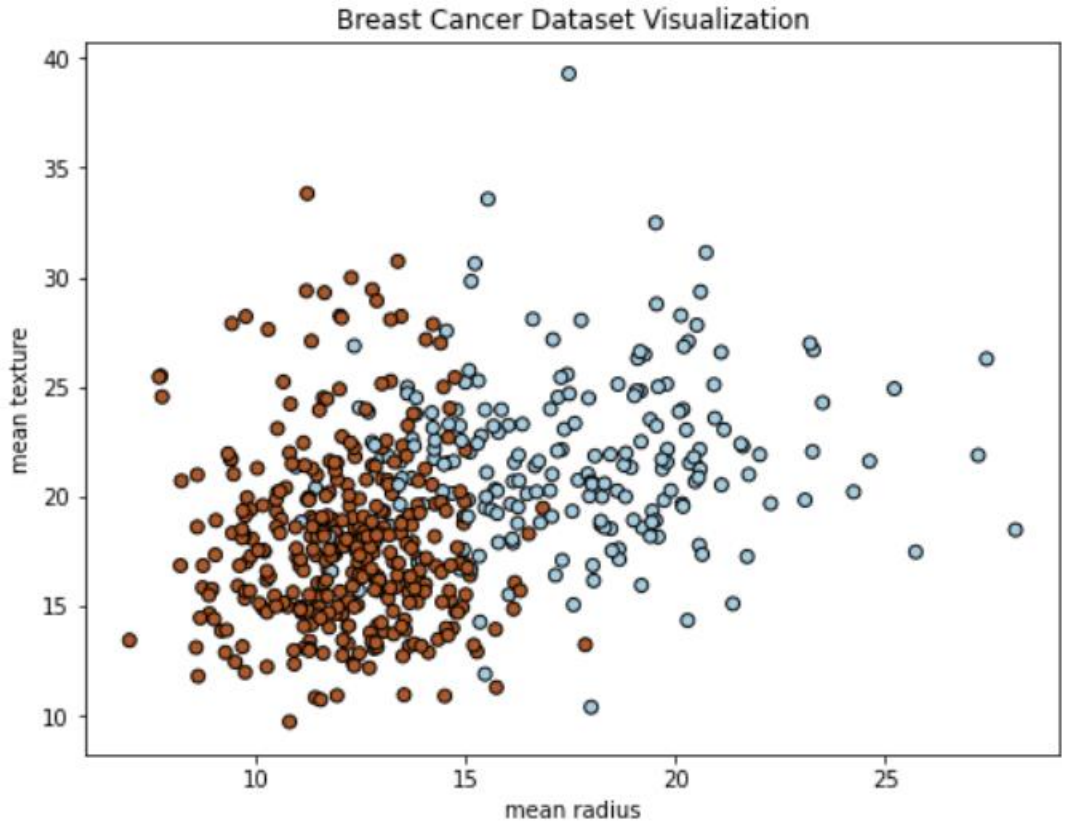
In [18]:
```
# Plot the decision boundary for a non-linear SVM problem
def plot_decision_boundary(model, ax=None):
    if ax is None:
        ax = plt.gca()

    xlim = ax.get_xlim()
    ylim = ax.get_ylim()

    # create grid to evaluate model
    x = np.linspace(xlim[0], xlim[1], 30)
    y = np.linspace(ylim[0], ylim[1], 30)
    Y, X = np.meshgrid(y, x)
```

```
    # shape data
    xy = np.vstack([X.ravel(), Y.ravel()]).T

    # get the decision boundary based on the model
    P = model.decision_function(xy).reshape(X.shape)

    # plot decision boundary
    ax.contour(X, Y, P,
               levels=[0], alpha=0.5,
               linestyles=['-'])

# plot data and decision boundary
plt.scatter(circle_X[:, 0], circle_X[:, 1], c=circle_y, s=50)
plot_decision_boundary(nonlinear_clf)
plt.scatter(nonlinear_clf.support_vectors_[:, 0], nonlinear_clf.support_vectors_[:, 1], s=50, lw=1, facecolors='non
plt.show()
```



SVM Kernels On Breast Cancer Dataset

In [22]:
```
import matplotlib.pyplot as plt
from sklearn import datasets

# Load Breast Cancer dataset
cancer = datasets.load_breast_cancer()
X = cancer.data[:, :2]  # Using only two features for visualization
y = cancer.target

# Extract feature names
feature_names = cancer.feature_names[:2]

# Plot the dataset with labeled axes
plt.figure(figsize=(8, 6))
plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Paired, edgecolors='k', marker='o')
plt.title("Breast Cancer Dataset Visualization")
plt.xlabel(feature_names[0])
plt.ylabel(feature_names[1])
plt.show()
```

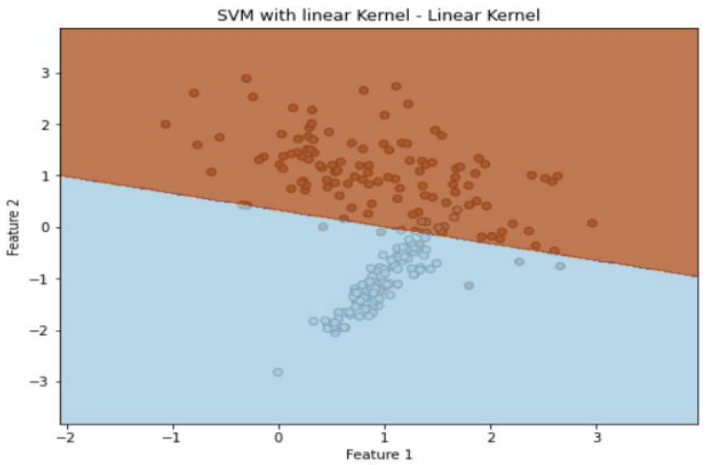


In [28]:
```python
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification, make_circles, make_moons
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

# Function to train and plot SVM with different kernels
def plot_svm_kernel(X, y, kernel, title):
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

    svm_classifier = SVC(kernel=kernel)
    svm_classifier.fit(X_train, y_train)
    predictions = svm_classifier.predict(X_test)

    accuracy = accuracy_score(y_test, predictions)
    print(f"Accuracy with {kernel} kernel: {accuracy}")

    # Plot decision boundary
    plt.figure(figsize=(8, 6))
    plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=plt.cm.Paired, edgecolors='k', marker='o')

    h = .02  # step size in the mesh
    x_min, x_max = X_train[:, 0].min() - 1, X_train[:, 0].max() + 1
    y_min, y_max = X_train[:, 1].min() - 1, X_train[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
    Z = svm_classifier.predict(np.c_[xx.ravel(), yy.ravel()])
```

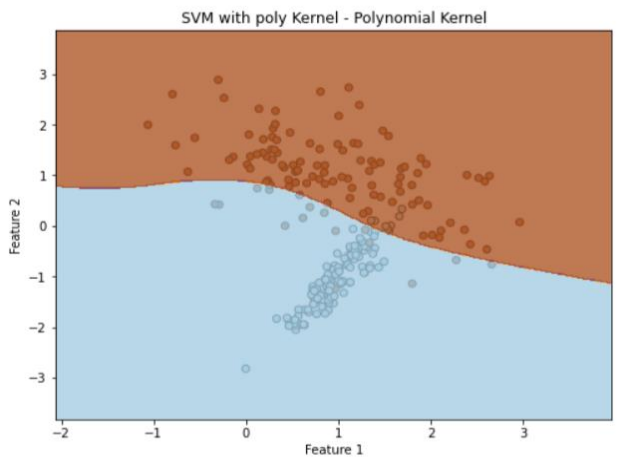
```python
    # Put the result into a color plot
    Z = Z.reshape(xx.shape)
    plt.contourf(xx, yy, Z, cmap=plt.cm.Paired, alpha=0.8)

    plt.title(f"SVM with {kernel} Kernel - {title}")
    plt.xlabel("Feature 1")
    plt.ylabel("Feature 2")
    plt.show()
```

```
# Linear Kernel
plot_svm_kernel(X_linear, y_linear, 'linear', 'Linear Kernel')

# Polynomial Kernel
plot_svm_kernel(X_poly, y_poly, 'poly', 'Polynomial Kernel')

# RBF Kernel (Same dataset as Circles)
plot_svm_kernel(X_circles, y_circles, 'rbf', 'RBF Kernel')

# Sigmoid Kernel
plot_svm_kernel(X_linear, y_linear, 'sigmoid', 'Sigmoid Kernel')
```

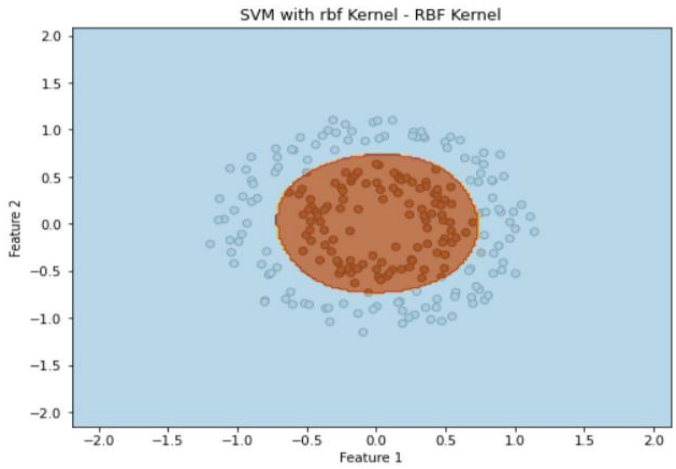
```
Accuracy with linear kernel: 0.9666666666666667
```




```
Accuracy with poly kernel: 0.9166666666666666
```




```
Accuracy with rbf kernel: 1.0
```




```
Accuracy with sigmoid kernel: 0.9166666666666666
```