Ex. No: 1	**A python program to implement Univariate Regression, Bivariate Regression and Multivariate Regression**
Date:24/01/25	

AIM:

To implement a python program using univariate, bivariate and multivariate regression features for a given iris dataset.

ALGORITHM:

STEP 1: Start the program

STEP 2: Import the necessary libraries: pandas for data manipulation, numpy for numerical operations, matplotlib.pyplot for plotting, and LinearRegression from sklearn.linear_model.

STEP 3: Read the dataset using pandas.read_csv() and store it in a variable such as data.

STEP 4: Prepare the data by extracting the independent variable(s) X and the dependent variable y, and reshape them into 2D arrays if required.

STEP 5: Perform univariate, bivariate, and multivariate linear regression using either numpy.polyfit or sklearn's LinearRegression, then make predictions and calculate the R-squared value to assess model accuracy.

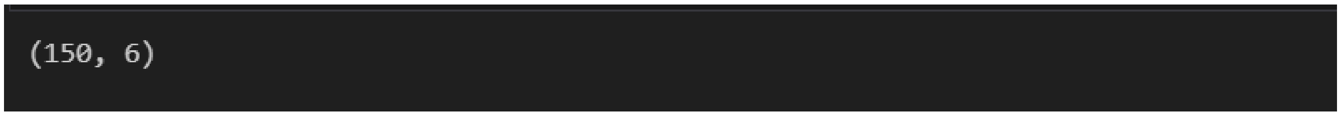
STEP 6: Visualize the results using 2D/3D scatter plots and regression lines or planes, and display the coefficients, intercepts, and R-squared values for each model.

STEP 7: Stop

PROGRAM:

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
df=pd.read_csv(r'C:\Users\221801049\Iris.csv')
df.head(150)
df.shape
```

OUTPUT:

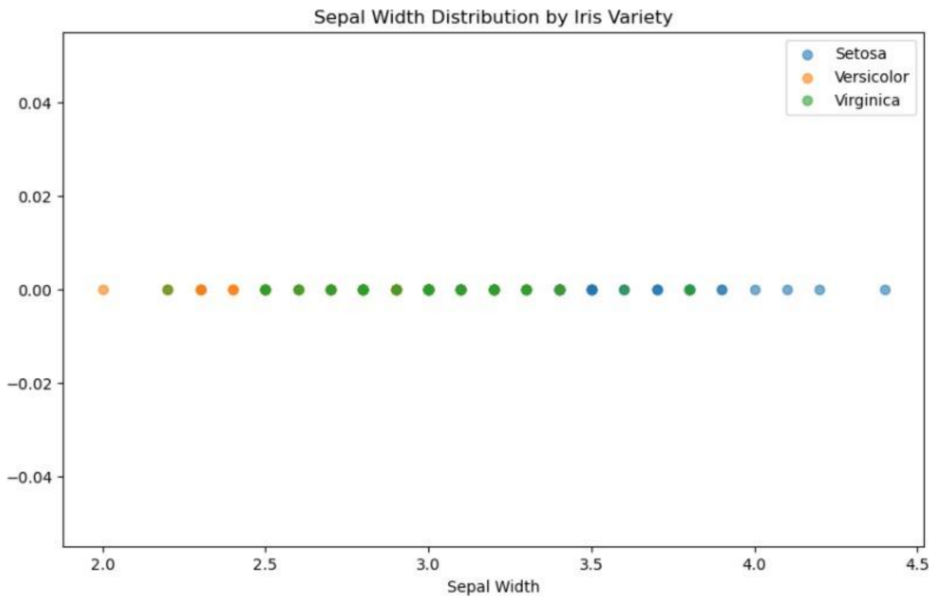


```
(150, 6)
```

UNIVARIATE FOR SEPAL WIDTH

```
df_Setosa = df[df['Species'] == 'Iris-setosa']
df_Virginica = df[df['Species'] == 'Iris-virginica']
df_Versicolor = df[df['Species'] == 'Iris-versicolor']
plt.figure(figsize=(10, 6))
plt.scatter(df_Setosa['SepalWidthCm'], np.zeros_like(df_Setosa['SepalWidthCm']), label='Setosa',
                                                                                        alpha=0.6)
plt.scatter(df_Versicolor['SepalWidthCm'], np.zeros_like(df_Versicolor['SepalWidthCm']),
                                                                   label='Versicolor', alpha=0.6)
plt.scatter(df_Virginica['SepalWidthCm'], np.zeros_like(df_Virginica['SepalWidthCm']),
                                                                    label='Virginica', alpha=0.6)
plt.xlabel('Sepal Width')
plt.title('Sepal Width Distribution by Iris Variety')
plt.legend()
plt.show()
```

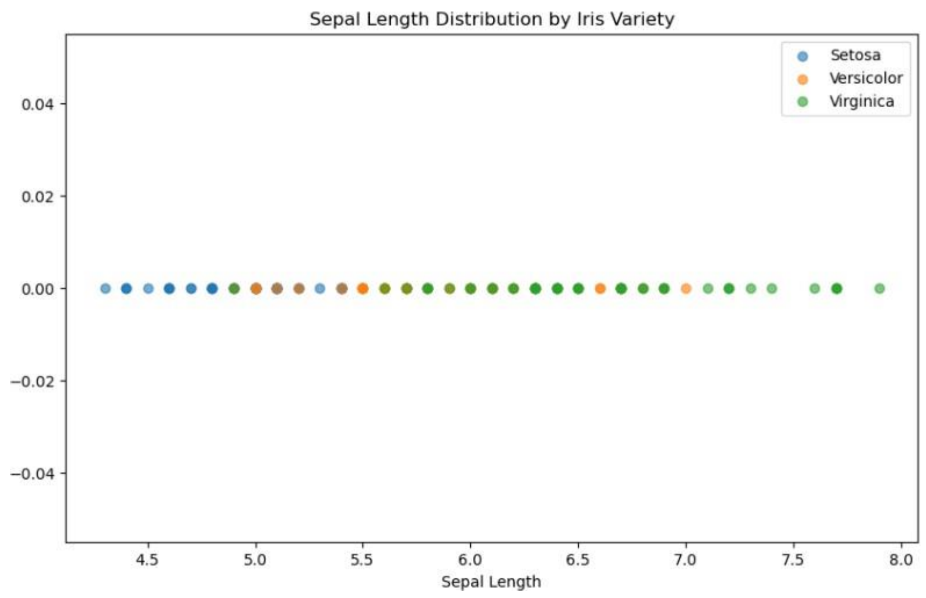
OUTPUT:



UNIVARIATE FOR SEPAL LENGTH

```
plt.figure(figsize=(10, 6))
plt.scatter(df_Setosa['SepalLengthCm'], np.zeros_like(df_Setosa['SepalLengthCm']),
                                                                   label='Setosa',alpha=0.6)
plt.scatter(df_Versicolor['SepalLengthCm'], np.zeros_like(df_Versicolor['SepalLengthCm']),
                                                                   label='Versicolor', alpha=0.6)
plt.scatter(df_Virginica['SepalLengthCm'], np.zeros_like(df_Virginica['SepalLengthCm']),
                                                                   label='Virginica',  alpha=0.6)
plt.xlabel('Sepal Length')
plt.title('Sepal Length Distribution by Iris Variety')
plt.legend()
plt.show()
```

OUTPUT:



UNIVARIATE FOR PETAL WIDTH

plt.figure(figsize=(12, 6))

plt.scatter(df_Setosa['PetalWidthCm'], np.zeros_like(df_Setosa['PetalWidthCm']), label='Setosa', alpha=0.6)
plt.scatter(df_Versicolor['PetalWidthCm'], np.zeros_like(df_Versicolor['PetalWidthCm']), label='Versicolor', alpha=0.6)

plt.scatter(df_Virginica['PetalWidthCm'], np.zeros_like(df_Virginica['PetalWidthCm']), label='Virginica', alpha=0.6)

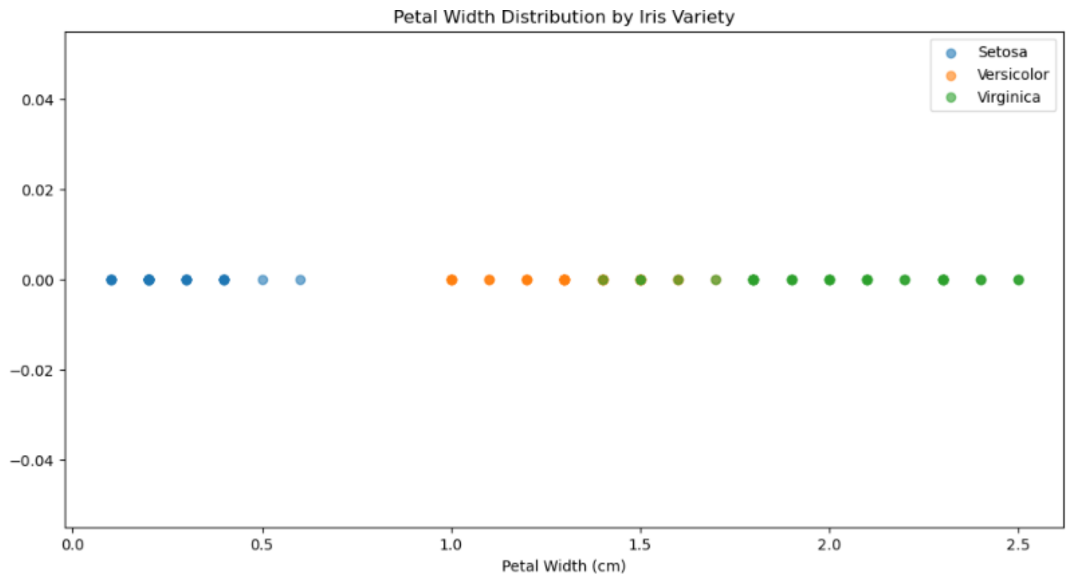
plt.xlabel('Petal Width (cm)')

plt.title('Petal Width Distribution by Iris Variety')

plt.legend()

plt.show()

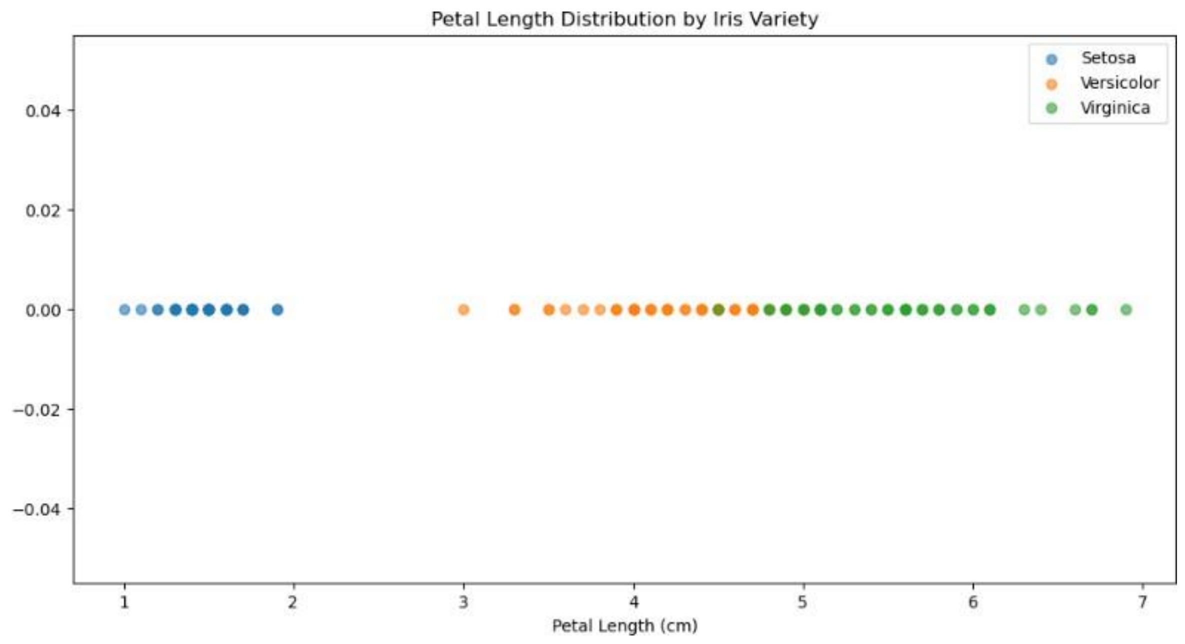
OUTPUT:



UNIVARIATE FOR PETAL LENGTH

```
plt.figure(figsize=(12, 6))
plt.scatter(df_Setosa['PetalLengthCm'], np.zeros_like(df_Setosa['PetalLengthCm']), label='Setosa', alpha=0.6)
plt.scatter(df_Versicolor['PetalLengthCm'], np.zeros_like(df_Versicolor['PetalLengthCm']), label='Versicolor',
                                                                                alpha=0.6)
plt.scatter(df_Virginica['PetalLengthCm'], np.zeros_like(df_Virginica['PetalLengthCm']), label='Virginica',
                                                                                alpha=0.6)
plt.xlabel('Petal Length (cm)')
plt.title('Petal Length Distribution by Iris Variety')
plt.legend()
plt.show()
```

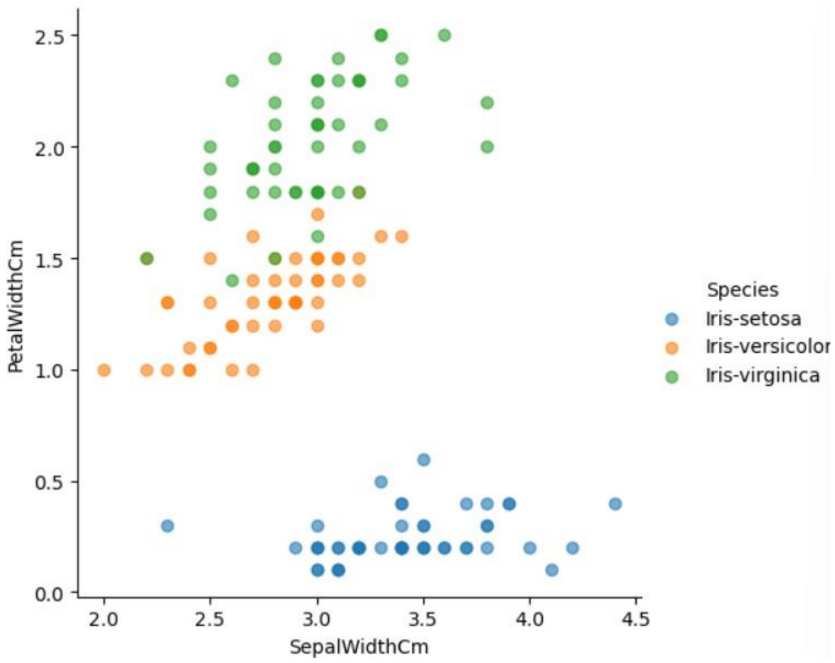
OUTPUT:



BIVARIATE SEPAL.WIDTH VS PETAL.WIDTH

```
g = sns.FacetGrid(df, hue='Species', height=5)
g.map(plt.scatter, 'SepalWidthCm', 'PetalWidthCm', alpha=0.6)
g.add_legend()
plt.show()
```

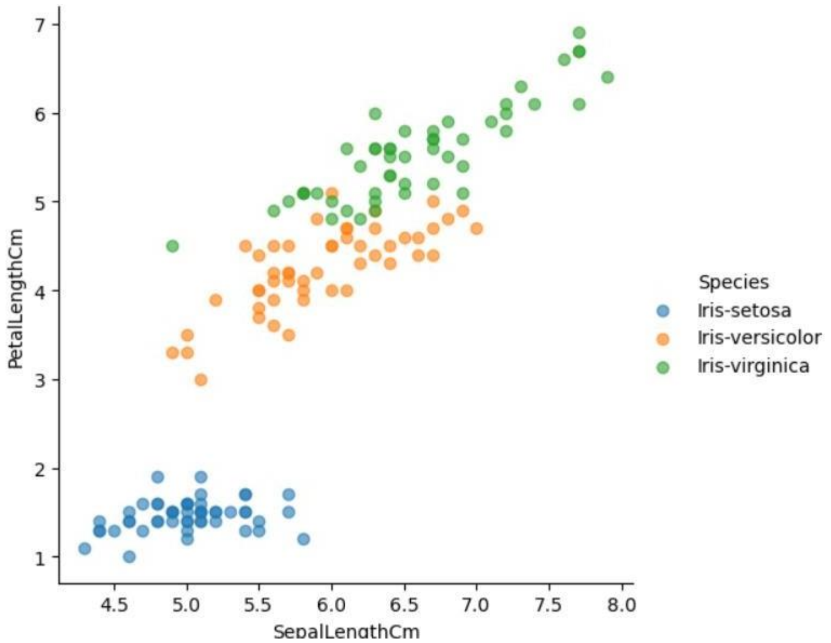
OUTPUT:



BIVARIATE SEPAL.LENGTH VS PETAL.LENGTH

```
g = sns.FacetGrid(df, hue='Species', height=5)
g.map(plt.scatter, 'SepalLengthCm', 'PetalLengthCm', alpha=0.6)
g.add_legend()
plt.show()
```

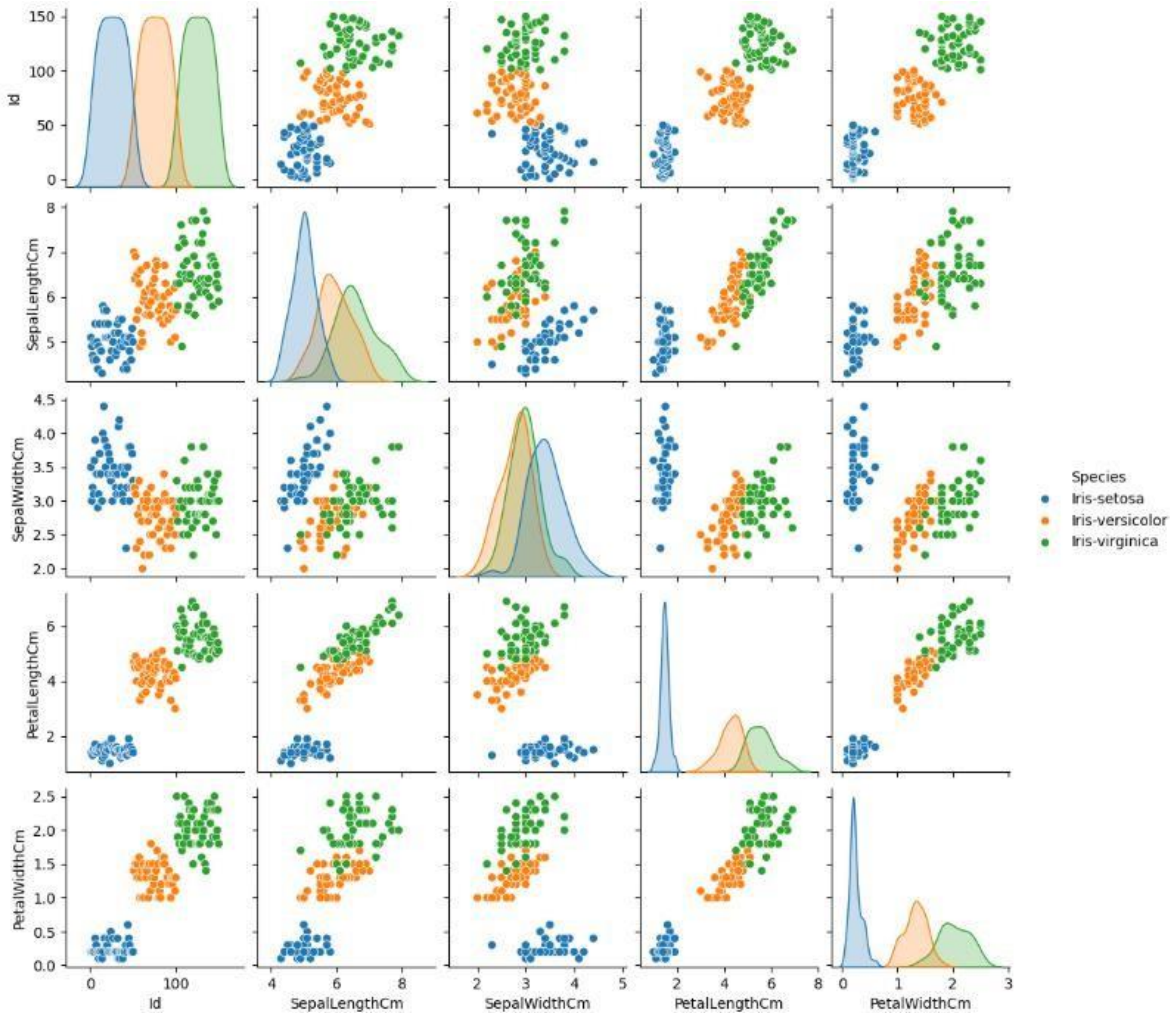
OUTPUT:



MULTIVARIATE ALL THE FEATURES

```
g=sns.pairplot(df,hue='Species',height=2)
plt.show()
```

OUTPUT:



RESULT:

Thus, the python program to implement univariate, bivariate and multivariate regression features for the given iris dataset is analyzed and the features are plotted using scatter plot.

Ex. No: 2	A python program to implement Simple linear regression using Least Square Method
Date:31/01/25	

AIM:

To implement a python program for constructing a simple linear regression using least square method.

ALGORITHM:

STEP 1: Start the program

STEP 2: Import pandas for data handling and matplotlib.pyplot for plotting. Read the dataset using pandas.read_csv() and store it in a variable like data.

STEP 3: Extract the independent variable X and dependent variable y, and reshape them into 2D arrays if necessary.

STEP 4: Calculate the mean of X and y, then compute the slope (m) and intercept (b) using the formulas for simple linear regression.

STEP 5: Use the slope and intercept to make predictions for each X value. Calculate the Total Sum of Squares (TSS), Residual Sum of Squares (RSS), and the R-squared value to assess model accuracy.

STEP 6: Plot the original data points as a scatter plot and the regression line using the predicted values.

STEP 7: Print the calculated slope, intercept, and R-squared value to complete the program.

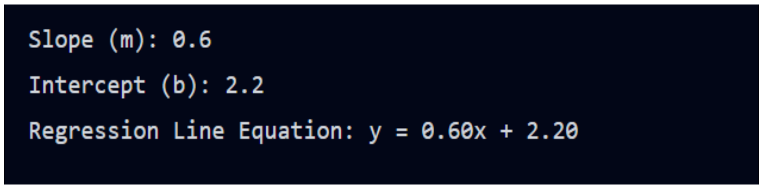
STEP 8: Stop

PROGRAM:

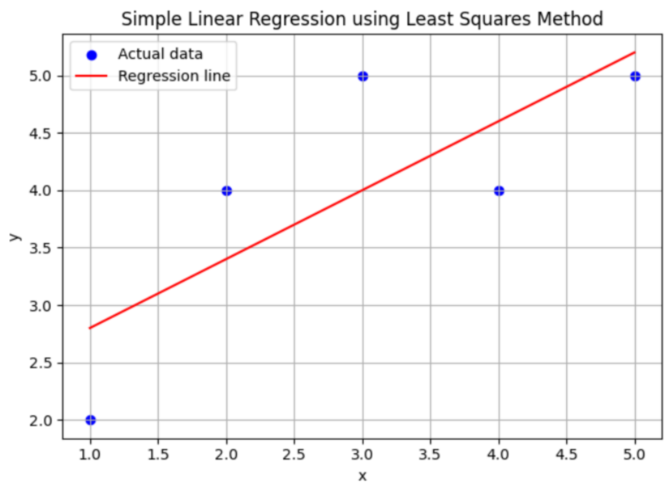
```
import numpy as np
import matplotlib.pyplot as plt
# Sample data
x = np.array([1, 2, 3, 4, 5], dtype=float)
y = np.array([2, 4, 5, 4, 5], dtype=float)
# Calculate means
x_mean = np.mean(x)
y_mean = np.mean(y)
# Calculate coefficients
numerator = np.sum((x - x_mean) * (y - y_mean))
denominator = np.sum((x - x_mean)**2)
slope = numerator / denominator
intercept = y_mean - slope * x_mean
# Prediction function
def predict(x_val):
    return slope * x_val + intercept
# Predict y values
```

```
y_pred = predict(x)
# Print results
print(f"Slope (m): {slope}")
print(f"Intercept (b): {intercept}")
print("Regression Line Equation: y = {:.2f}x + {:.2f}".format(slope, intercept))
# Plotting
plt.scatter(x, y, color='blue', label='Actual data')
plt.plot(x, y_pred, color='red', label='Regression line')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Simple Linear Regression using Least Squares Method')
plt.legend()
plt.grid(True)
plt.show()
```

OUTPUT:



```
Slope (m): 0.6
Intercept (b): 2.2
Regression Line Equation: y = 0.60x + 2.20
```



RESULT:

Thus, the python program for constructing a simple linear regression using least square method has been implemented and executed successfully.

Ex. No: 3	**A python program to implement Logistic Model**
Date:07/02/25	

AIM:

To implement python program for the logistic model using DigitalAd dataset.

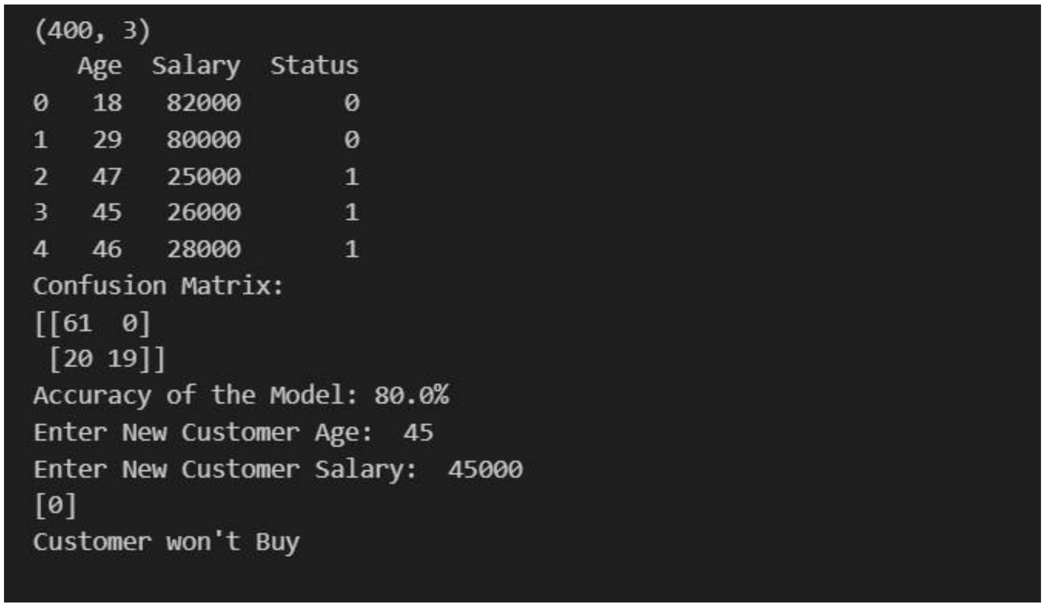
ALGORITHM:

STEP 1: Start the program
STEP 2: Load the dataset
STEP 3: Display the shape and first 5 rows of the dataset to understand its structure
STEP 4: Separate features (X) and target (Y) and split the data into training and testing sets
STEP 5: Train a Logistic Regression model and Make predictions on the test set
STEP 6: Evaluate the model using confusion matrix and Print the accuracy of the model
STEP 7: Predict for a new customer and Output the prediction result
STEP 8: Stop the program

PROGRAM:

```
import pandas as pd
import numpy as np
data = pd.read_csv("DigitalAd_dataset.csv")
print(data.shape)
print(data.head(5))
X = data.iloc[:, :-1].values  # All columns except the last one (features)
Y = data.iloc[:, -1].values   # Last column as target
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.25, random_state=0)
from sklearn.preprocessing import StandardScaler              # Feature scaling (standardizing the features)
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
from sklearn.linear_model import LogisticRegression           # Train a Logistic Regression model
model = LogisticRegression(random_state=0)
model.fit(X_train, y_train)

y_pred = model.predict(X_test)                                # Make predictions on the test set
from sklearn.metrics import confusion_matrix, accuracy_score
cm = confusion_matrix(y_test, y_pred)                         # Evaluate the model using confusion matrix
print("Confusion Matrix: ")
print(cm)
print("Accuracy of the Model: {0}%".format(accuracy_score(y_test, y_pred) * 100))
age = int(input("Enter New Customer Age: "))           # Predict for a new customer
sal = int(input("Enter New Customer Salary: "))
newCust = [[age, sal]]
result = model.predict(sc.transform(newCust))          # Output the prediction result
print(result)
if result == 1:
    print("Customer will Buy")
else:
    print("Customer won't Buy")
```

OUTPUT:



```
(400, 3)
   Age  Salary  Status
0   18   82000       0
1   29   80000       0
2   47   25000       1
3   45   26000       1
4   46   28000       1
Confusion Matrix:
[[61  0]
 [20 19]]
Accuracy of the Model: 80.0%
Enter New Customer Age:  45
Enter New Customer Salary:  45000
[0]
Customer won't Buy
```

RESULT:

Thus, the python program to implement logistic regression for the given dataset is analyzed and the performance of the developed model is measured successfully.

Ex. No: 4	**A python program to implement Single Layer Perceptron**
Date:14/02/25	

AIM:

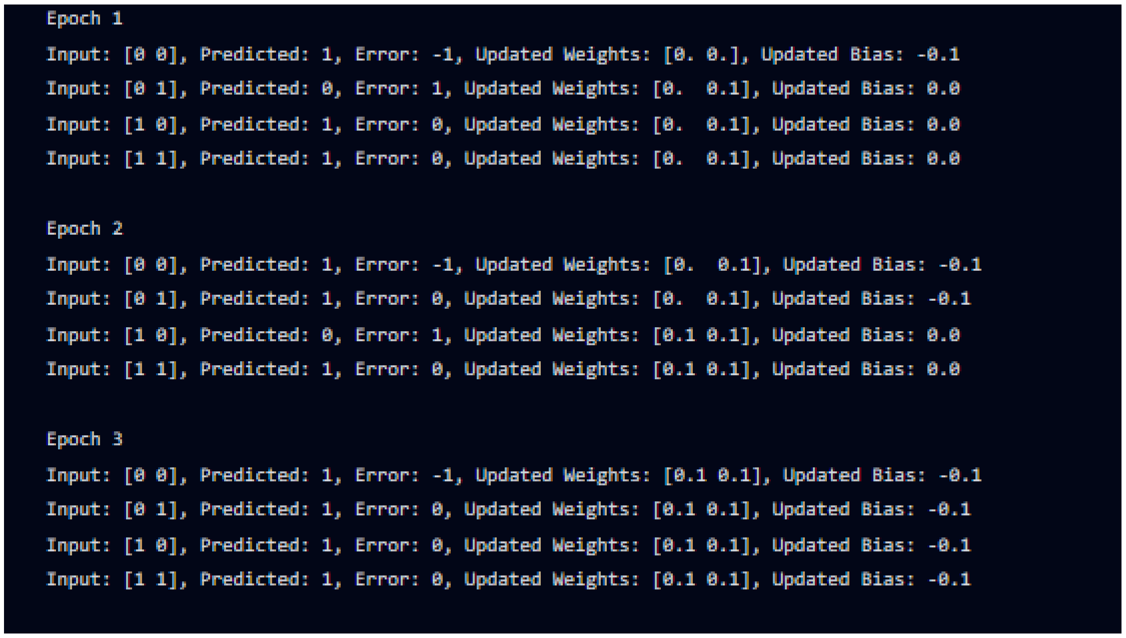
To implement python program for the single layer perceptron.

STEP 1: Start the program.
STEP 2: Import the necessary packages: numpy and matplotlib.
STEP 3: Prepare or read a simple dataset (input features X and labels Y).
STEP 4: Initialize the perceptron parameters: weights and bias.
STEP 5: Define the activation function (step function).
STEP 6: Train the perceptron. For each epoch (iteration),Calculate the output.
STEP 7: After training, display the final weights and bias.
STEP 8: Visualize the decision boundary using matplotlib.
STEP 9: Stop the program.

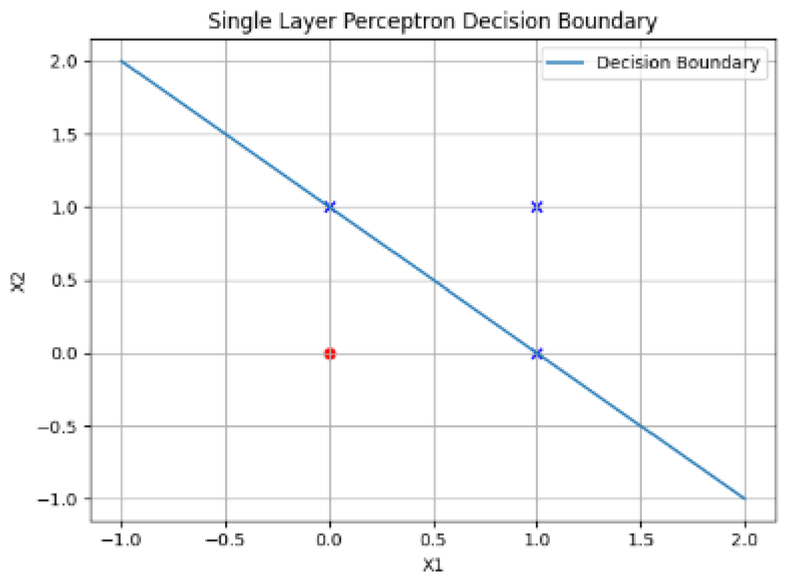
PROGRAM:

```
import numpy as np
import matplotlib.pyplot as plt
X = np.array([
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]])
Y = np.array([0, 1, 1, 1])  # OR gate outputs
weights = np.zeros(X.shape[1])
bias = 0
learning_rate = 0.1
epochs = 10
def activation_fn(x):
    return 1 if x >= 0 else 0
for epoch in range(epochs):
    print(f"\nEpoch {epoch+1}")
    for i in range(len(X)):
        linear_output = np.dot(X[i], weights) + bias
        y_predicted = activation_fn(linear_output)
        error = Y[i] - y_predicted
        weights += learning_rate * error * X[i]
        bias += learning_rate * error
        print(f"Input: {X[i]}, Predicted: {y_predicted}, Error: {error}, Updated Weights: {weights}, Updated
Bias: {bias}")
print("\nFinal Weights:", weights)
print("Final Bias:", bias)
for i in range(len(X)):
    if Y[i] == 0:
        plt.scatter(X[i][0], X[i][1], color='red', marker='o')
    else:
        plt.scatter(X[i][0], X[i][1], color='blue', marker='x')
x_values = [np.min(X[:, 0] - 1), np.max(X[:, 0] + 1)]
y_values = -(weights[0] * np.array(x_values) + bias) / weights[1]
plt.plot(x_values, y_values, label='Decision Boundary')
plt.xlabel('X1')
plt.ylabel('X2')
plt.title('Single Layer Perceptron Decision Boundary')
plt.legend()
plt.grid(True)
plt.show()
```

OUTPUT



```
Epoch 1
Input: [0 0], Predicted: 1, Error: -1, Updated Weights: [0. 0.], Updated Bias: -0.1
Input: [0 1], Predicted: 0, Error: 1, Updated Weights: [0.  0.1], Updated Bias: 0.0
Input: [1 0], Predicted: 1, Error: 0, Updated Weights: [0.  0.1], Updated Bias: 0.0
Input: [1 1], Predicted: 1, Error: 0, Updated Weights: [0.  0.1], Updated Bias: 0.0

Epoch 2
Input: [0 0], Predicted: 1, Error: -1, Updated Weights: [0.  0.1], Updated Bias: -0.1
Input: [0 1], Predicted: 1, Error: 0, Updated Weights: [0.  0.1], Updated Bias: -0.1
Input: [1 0], Predicted: 0, Error: 1, Updated Weights: [0.1 0.1], Updated Bias: 0.0
Input: [1 1], Predicted: 1, Error: 0, Updated Weights: [0.1 0.1], Updated Bias: 0.0

Epoch 3
Input: [0 0], Predicted: 1, Error: -1, Updated Weights: [0.1 0.1], Updated Bias: -0.1
Input: [0 1], Predicted: 1, Error: 0, Updated Weights: [0.1 0.1], Updated Bias: -0.1
Input: [1 0], Predicted: 1, Error: 0, Updated Weights: [0.1 0.1], Updated Bias: -0.1
Input: [1 1], Predicted: 1, Error: 0, Updated Weights: [0.1 0.1], Updated Bias: -0.1
```



RESULT:

Thus, the python program Single Layer Perceptron was successfully implemented using Python.

Ex. No: 5	A python program to implement Multilayer Perceptron with Back Propagation
Date:21/02/25	

AIM:

To implement multilayer perceptron with back propagation using python.

PROCEDURE:

STEP 1: Start the program

STEP 2: Create a class MLP with an __init___method that initializes weights and biases for the input-to-hidden and hidden-to-output layers.

STEP 3: Include activation functions like sigmoid and its derivative and Implement the train() method which performs forward propagation to calculate predictions, backward propagation to compute gradients, and updates weights and biases using gradient descent over a number of epochs.

STEP 4: Set up an XOR dataset with 2 inputs and 1 output. Instantiate the MLP with 2 input nodes, 4 hidden nodes, and 1 output node. Train the model using the XOR data for 10,000 epochs.

STEP 5: After training, pass the input data through the network again using the learned weights to generate predictions.

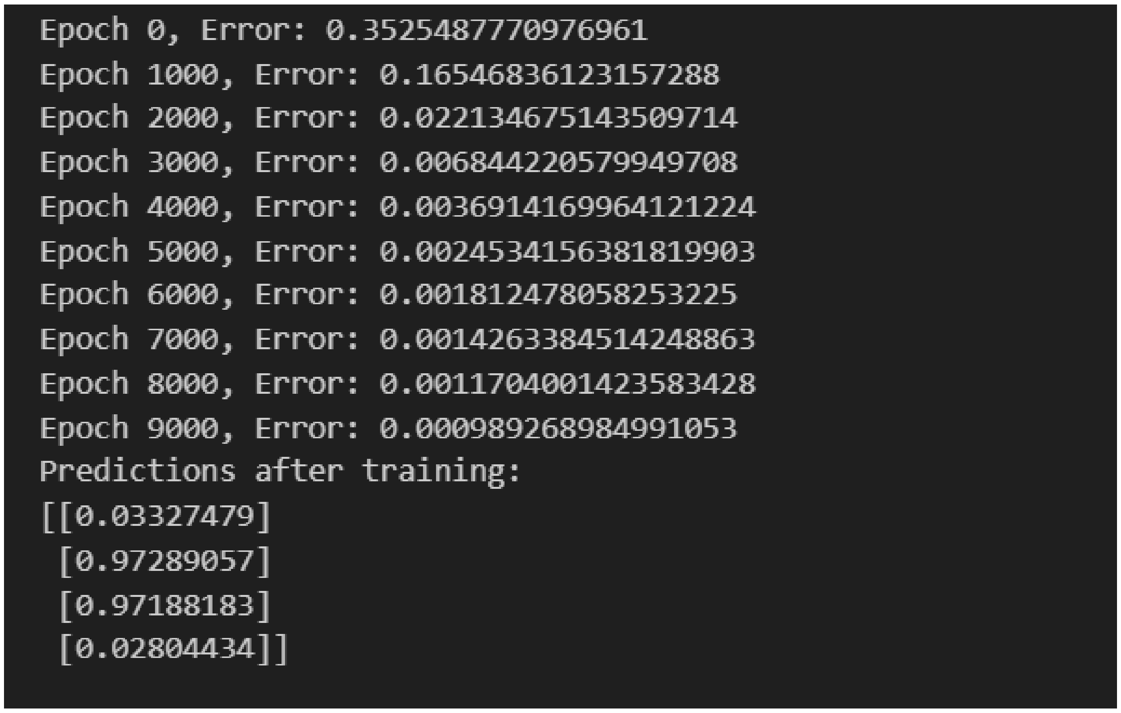
STEP 6: print the outputs to observe how well the model learned the XOR logic.

STEP 7: Stop the program

PROGRAM:

```
import numpy as np
class MLP:
    def __init_(self, input_size, hidden_size, output_size):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size

        # initialize weights matrix and biases
        self.W_input_hidden = np.random.rand(self.input_size, self.hidden_size)
        self.b_input_hidden = np.zeros((1, self.hidden_size))
        self.W_hidden_output = np.random.rand(self.hidden_size, self.output_size)
        self.b_hidden_output = np.zeros((1, self.output_size))
    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))
    def d_sigmoid(self, x):
        return x * (1 - x)
    def train(self, input_data, target, epochs=1000, lr=0.2):
        for epoch in range(epochs):
            # Forward propagation
            hidden_layer_input = np.dot(input_data, self.W_input_hidden) + self.b_input_hidden
            hidden_layer_output = self.sigmoid(hidden_layer_input)
            output_layer_input = np.dot(hidden_layer_output, self.W_hidden_output) +
                                                                         self.b_hidden_output
            output = self.sigmoid(output_layer_input)

            # Backward propagation
            output_error = target - output
            output_grad = output_error * self.d_sigmoid(output)
            hidden_error = np.dot(output_grad, self.W_hidden_output.T)
            hidden_grad = hidden_error * self.d_sigmoid(hidden_layer_output)

            # Update weights and biases using gradient descent
            self.W_hidden_output += np.dot(hidden_layer_output.T, output_grad) * lr
            self.b_hidden_output += np.sum(output_grad, axis=0, keepdims=True) * lr
            self.W_input_hidden += np.dot(input_data.T, hidden_grad) * lr
            self.b_input_hidden += np.sum(hidden_grad, axis=0, keepdims=True) * lr

            # Optionally, print error every 1000 epochs
            if epoch % 1000 == 0:
                error = np.mean(np.square(target - output))  # Mean Squared Error
                print(f'Epoch {epoch}, Error: {error}')

# Example usage:
if __name___== "__main__":
    # XOR problem: 4 samples, 2 input features, 1 output
    X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
    y = np.array([[0], [1], [1], [0]])

    # Create MLP with 2 input nodes, 4 hidden nodes, and 1 output node
    mlp = MLP(input_size=2, hidden_size=4, output_size=1)

    mlp.train(X, y, epochs=10000)                # Train the model

    # Test the model after training
    print("Predictions after training:")
    hidden_layer_input = np.dot(X, mlp.W_input_hidden) + mlp.b_input_hidden
    hidden_layer_output = mlp.sigmoid(hidden_layer_input)
    output_layer_input = np.dot(hidden_layer_output, mlp.W_hidden_output) + mlp.b_hidden_output
    predictions = mlp.sigmoid(output_layer_input)
    print(predictions)
```

OUTPUT:



```
Epoch 0, Error: 0.3525487770976961
Epoch 1000, Error: 0.16546836123157288
Epoch 2000, Error: 0.022134675143509714
Epoch 3000, Error: 0.006844220579949708
Epoch 4000, Error: 0.0036914169964121224
Epoch 5000, Error: 0.0024534156381819903
Epoch 6000, Error: 0.001812478058253225
Epoch 7000, Error: 0.0014263384514248863
Epoch 8000, Error: 0.0011704001423583428
Epoch 9000, Error: 0.000989268984991053
Predictions after training:
[[0.03327479]
 [0.97289057]
 [0.97188183]
 [0.02804434]]
```

RESULT:

Thus, the Python program to implement a simple multi-layer perceptron (MLP) for the XOR problem is analyzed, and the model successfully predicts the output after training by learning the non-linear relationships in the data.

Ex. No: 6	A python program to do face recognition using SVM classifier
Date: 28/02/25	

AIM:

To implement a SVM classifier model using python and determine its accuracy.

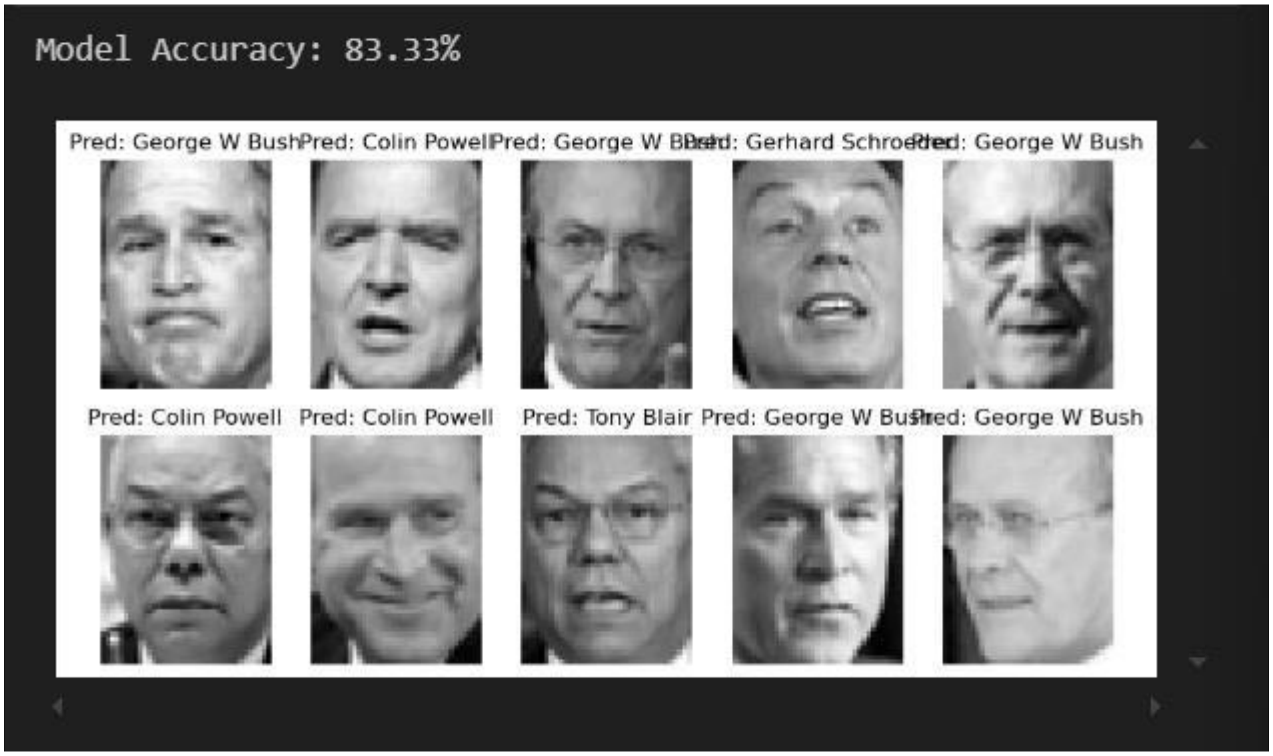
PROCEDURE:

STEP 1: Start the program.
STEP 2: Import the necessary libraries and Load the dataset.
STEP 3: Flatten the images and apply PCA
STEP 4: Split the dataset into train & test sets and train SVM model
STEP 5: Make prediction and evaluate the accuracy.
STEP 6: Print the model accuracy and visualize some test images with predictions.
STEP 7: Stop the program.

PROGRAM:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
from sklearn.svm import SVC
from sklearn.datasets import fetch_lfw_people
from sklearn.model_selection import train_test_split
from sklearn.decomposition import PCA
from sklearn.metrics import accuracy_score
lfw = fetch_lfw_people(min_faces_per_person=100, resize=0.4, download_if_missing=True)
X, y = lfw.images, lfw.target  # Images and labels
X_flat = X.reshape(X.shape[0], -1)  # Convert images to 1D array
pca = PCA(n_components=100).fit(X_flat)  # Reduce dimensions to 100 principal components
X_pca = pca.transform(X_flat)
X_train, X_test, y_train, y_test = train_test_split(X_pca, y, test_size=0.2, random_state=42)
svm = SVC(kernel='linear')
svm.fit(X_train, y_train)
y_pred = svm.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f"Model Accuracy: {accuracy * 100:.2f}%")
fig, axes = plt.subplots(2, 5, figsize=(10, 5))  # Create a grid of 2 rows, 5 columns
for i, ax in enumerate(axes.ravel()):
    ax.imshow(X[i], cmap='gray')  # Show actual face image
    ax.set_title(f"Pred: {lfw.target_names[y_pred[i]]}")  # Predicted name
    ax.axis('off')
plt.show()
```

OUTPUT:



```
Model Accuracy: 83.33%
```

RESULT:

Thus the python program to implement SVM classifier model has been executed successfully and the classified output has been analyzed for the given dataset.\\

Ex. No: 7	**A python program to implement Decision Tree**
Date:07/03/2025	

AIM:

To build, train, and visualize a Decision Tree Classifier using the Iris dataset in Python, and to evaluate the model's performance based on its accuracy on the test data.

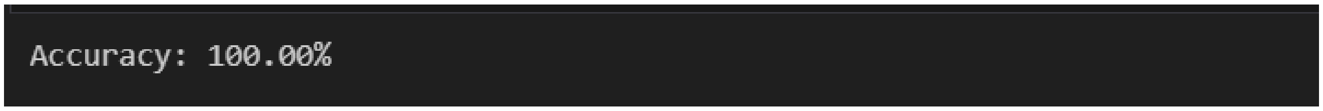
ALGORITHM:

STEP 1: Start the program.
STEP 2: Import necessary modules: datasets, train_test_split, DecisionTreeClassifier, accuracy_score, and matplotlib.pyplot.
STEP 3: Load the Iris dataset and separate it into features (X) and target labels (y).
STEP 4: Split the dataset into training and testing sets using train_test_split().
STEP 5: Create an instance of DecisionTreeClassifier, train (fit) it using the training data (X_train, y_train).
STEP 6: Predict the target labels for the test set using the trained classifier.
STEP 7: Calculate and print the model's accuracy using accuracy_score().
STEP 8: Visualize the trained Decision Tree using plot_tree() and display it with plt.show().
STEP 9: Stop the program.

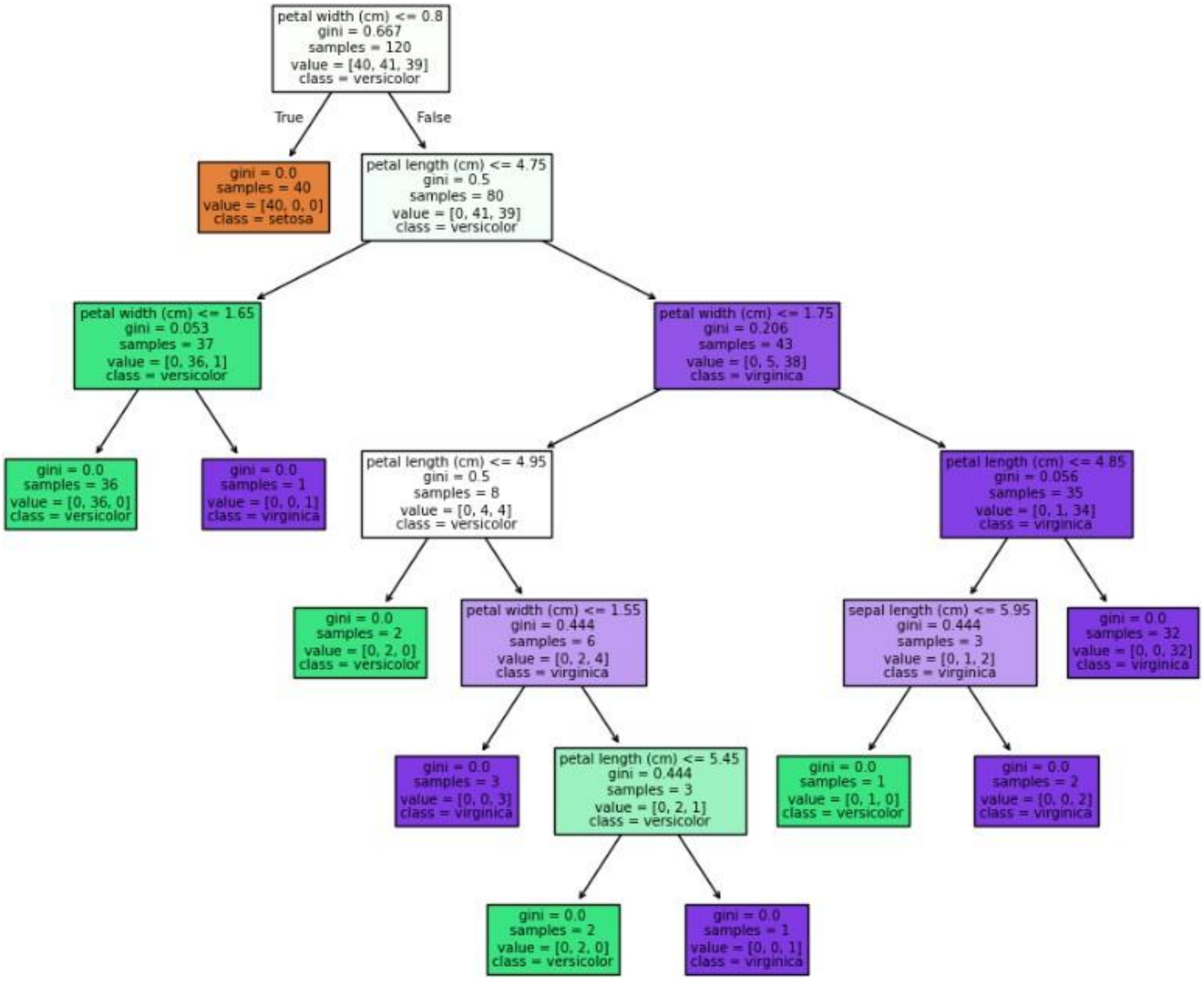
PROGRAM:

```
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt
iris = datasets.load_iris()
X, y = iris.data, iris.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
clf = DecisionTreeClassifier()
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
print(f"Accuracy: {accuracy_score(y_test, y_pred) * 100:.2f}%")
plt.figure(figsize=(12, 10))
plot_tree(clf, filled=True, feature_names=iris.feature_names, class_names=iris.target_names)
plt.show()
```

OUTPUT:



```
Accuracy: 100.00%
```



RESULT:

Thus, the Python program to build, train, and visualize a Decision Tree Classifier on the Iris dataset was successfully implemented.

Ex. No: 8	**A python program to implement Boosting**
Date:28/03/25	

AIM:

To build and evaluate a python program to implement boosting.

ALGORITHM:

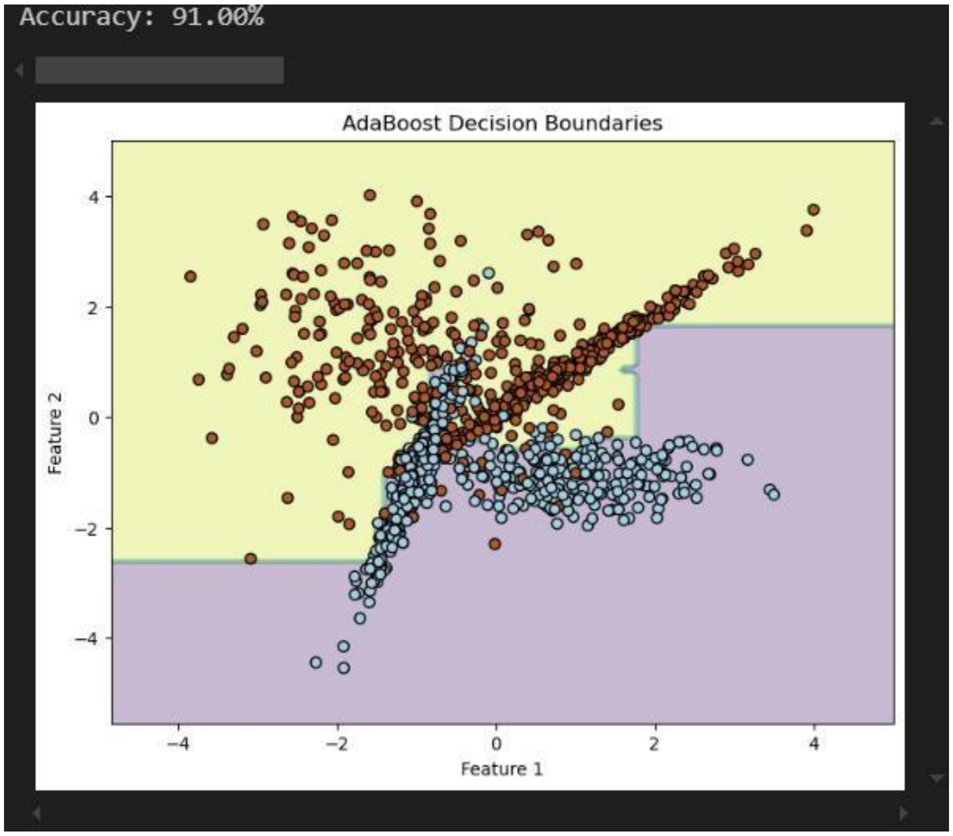
STEP 1: Start the program.
STEP 2: Import required libraries: sklearn, numpy, and matplotlib.
STEP 3: Generate a synthetic binary classification dataset using make_classification().
STEP 4: Split the dataset into training and testing sets using train_test_split().
STEP 5: Initialize the AdaBoostClassifier with a base estimator (DecisionTreeClassifier) and train it on the training data.
STEP 6: Use the trained model to make predictions on the test set.
STEP 7: Calculate and print the accuracy of the model using accuracy_score().
STEP 8: Visualize the decision boundaries by predicting over a meshgrid and plotting the results.
STEP 9: Stop the program.

PROGRAM:

```
        from sklearn.ensemble import AdaBoostClassifier
        from sklearn.tree import DecisionTreeClassifier
        from sklearn.datasets import make_classification
        from sklearn.model_selection import train_test_split
        from sklearn.metrics import accuracy_score
        import matplotlib.pyplot as plt
        import numpy as np
        X, y = make_classification(n_samples=1000, n_features=2, n_informative=2, n_redundant=0,
random_state=42)
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
        boost = AdaBoostClassifier(DecisionTreeClassifier(max_depth=1), n_estimators=50,
random_state=42)
        boost.fit(X_train, y_train)
        y_pred = boost.predict(X_test)              # Predict and evaluate model performance
        accuracy = accuracy_score(y_test, y_pred)
        print(f"Accuracy: {accuracy * 100:.2f}%")
        x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1              # Visualization - Decision Boundaries
        y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
        xx, yy = np.meshgrid(np.linspace(x_min, x_max, 100), np.linspace(y_min, y_max, 100))
        Z = boost.predict(np.c_[xx.ravel(), yy.ravel()])
        Z = Z.reshape(xx.shape)
```

```
plt.figure(figsize=(8, 6))
plt.contourf(xx, yy, Z, alpha=0.3)
plt.scatter(X[:, 0], X[:, 1], c=y, edgecolors="k", cmap=plt.cm.Paired)
plt.title("AdaBoost Decision Boundaries")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.show()
```

OUTPUT:



```
Accuracy: 91.00%
```

RESULT:

Thus, the Python program for Boosting using AdaBoostClassifier was successfully executed.

Ex. No: 9	**A python program to implement KNN and K-Means**
Date:04/04/25	

AIM:

To implement the K-Nearest Neighbors (KNN) and K Means algorithm on the Iris dataset for classification and evaluate its accuracy.

ALGORITHM:

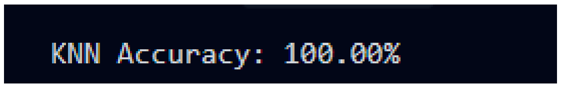
STEP 1: Start the program.
STEP 2: Import necessary libraries such as sklearn.datasets, train_test_split, KNeighborsClassifier, and accuracy_score.
STEP 3: Load the Iris dataset and extract the features (X) and target labels (y).
STEP 4: Split the dataset into training and testing sets.
STEP 5: Create a KNeighborsClassifier model with a defined value of k (e.g., k = 3).
STEP 6: Train the model using the training data.
STEP 7: Predict the labels for the test data.
STEP 8: Evaluate the model by calculating the accuracy score.
STEP 9: Display the results.
STEP 10: Stop the program.
STEP 11: Import necessary libraries such as sklearn.datasets, KMeans, matplotlib.pyplot, and PCA from sklearn.decomposition.
STEP 12: Load the Iris dataset and extract the feature matrix (X).
STEP 13: Choose the number of clusters (k = 3 for Iris)
STEP 14: Create a KMeans object.
STEP 15: Fit the KMeans model on the dataset to form clusters.
STEP 16: Retrieve and store the cluster labels.
STEP 17: Reduce dimensionality using PCA for visualization (2 components).
STEP 18: Plot the clustered data on a 2D scatter plot using the PCA-reduced values.
STEP 19: Display the visualization.
STEP 20: Stop the program.

PROGRAM:

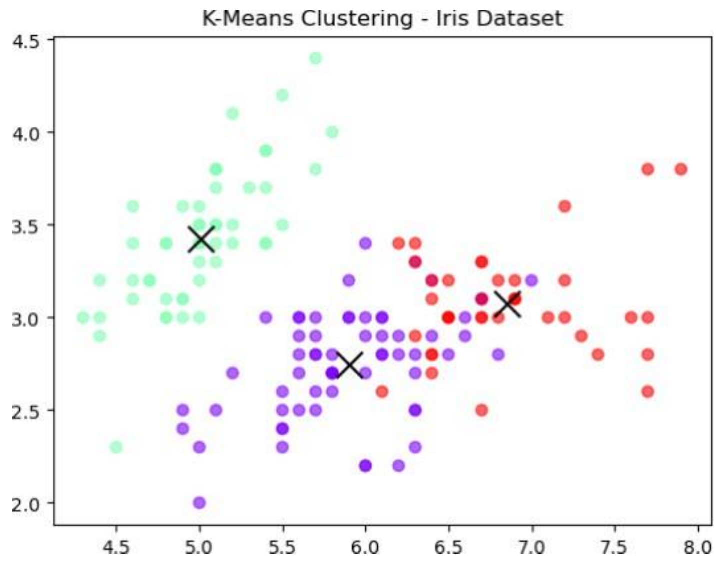
```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
# Load dataset
iris = load_iris()
X, y = iris.data, iris.target

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
knn = KNeighborsClassifier(n_neighbors=3)
```

```
knn.fit(X_train, y_train)
y_pred = knn.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f"KNN Accuracy: {accuracy * 100:.2f}%")
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
iris = load_iris()
X, y = iris.data, iris.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train, y_train)
y_pred = knn.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f"KNN Accuracy: {accuracy * 100:.2f}%")
```

OUTPUT:



```
KNN Accuracy: 100.00%
```



RESULT:

Thus the python program for building classification model using KNN algorithm is executed and verified successfully.

Ex. No: 10	**A python program to implement Dimensionality Reduction – PCA**
Date:11/04/25	

AIM:

To develop a Python program to implement Principal Component Analysis (PCA) for dimensionality reduction using the Digits dataset.

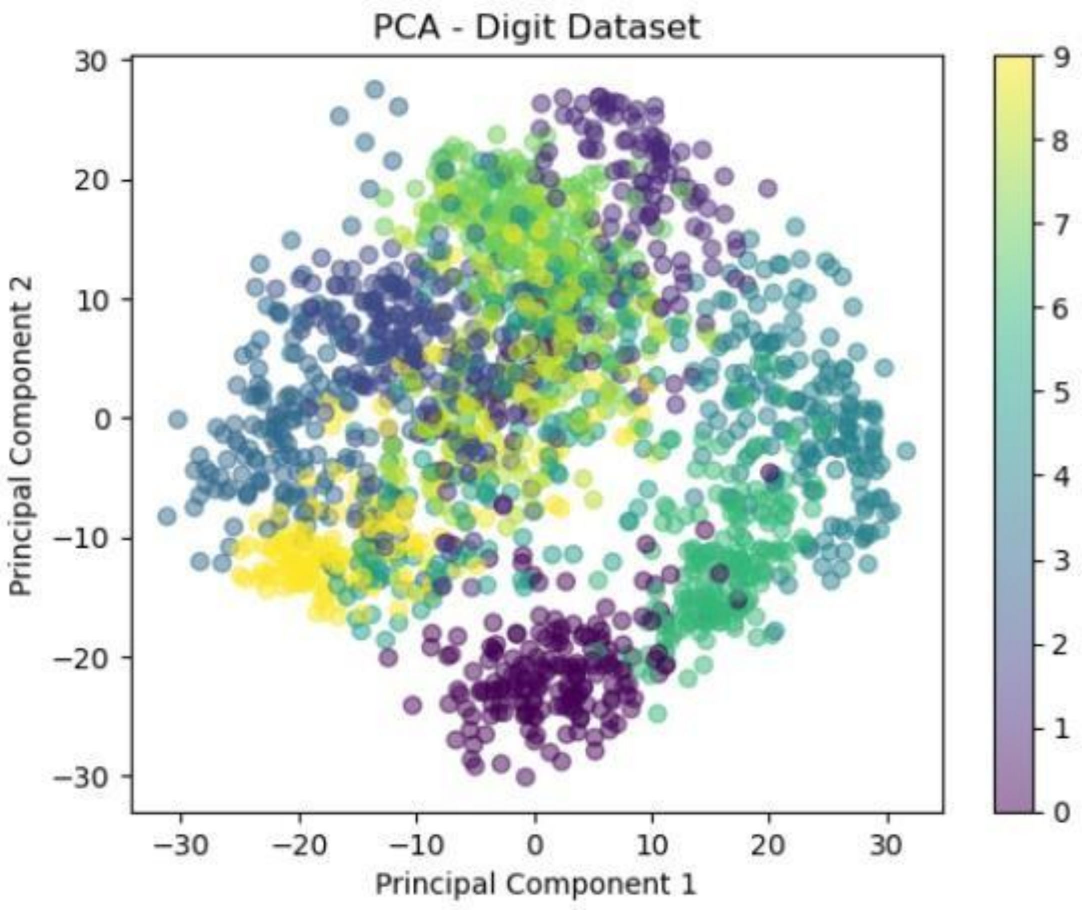
ALGORITHM:

STEP 1: Start the program.
STEP 2: Import the required libraries: numpy, matplotlib.pyplot, sklearn.decomposition.PCA, and sklearn.datasets.load_digits.
STEP 3: Load the Digits dataset using load_digits() and store the data in variable X.
STEP 4: Apply PCA with n_components=2 using the PCA() class and fit-transform the dataset.
STEP 5: Store the result of PCA in X_pca and visualize the 2D projection of the data using a scatter plot.
STEP 6: Color the points in the plot based on their digit labels using a color map for better interpretation.
STEP 7: Add labels, a colorbar, and a title to the plot for clarity.
STEP 8: Stop the program.

PROGRAM:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.datasets import load_digits
# Load dataset
digits = load_digits()
X = digits.data
y = digits.target
# Apply PCA
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)
# Plot PCA results
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=y, cmap='viridis', alpha=0.5)
plt.colorbar()
plt.title("PCA - Digit Dataset")
plt.xlabel("Principal Component 1")
plt.ylabel("Principal Component 2")
plt.show()
```

OUTPUT:



RESULT:

Thus, the Python program to perform dimensionality reduction using PCA was successfully executed and visualized using the Digits dataset.

Ex. No: 11	**Mini project – Develop a simple application using TensorFlow / Keras**
Date:11/04/25	

AIM:

To develop a Removing Noise from Images Using the TensorFlow.

ALGORITHM:

STEP 1: Start the program.
STEP 2: Add random Gaussian noise to the images and clip values to stay between 0 and 1.
STEP 3: Load the MNIST dataset and split it into training and testing sets..
STEP 4: Build a Convolutional autoencoder model:

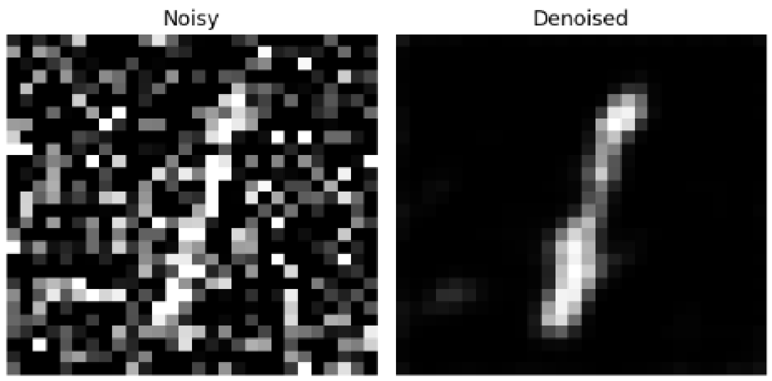
- Use Conv2D and Maxpooling layers for encoding.
- Use Con2D Transpose and Conv2D Layers for Decoding.

STEP 5: Compile the model using the adam optimizer and binary crossentropy loss.
STEP 6: Train the model with noisy images as input and original images as the output
STEP 7: Predict and Display the denoised images alongside their noisy versions
STEP 8: Stop the program.

PROGRAM:

```
import tensorflow as tf, matplotlib.pyplot as plt
(x,*), (xt,*) = tf.keras.datasets.mnist.load_data()
x = xt[..., None] / 255.
xn = tf.clip_by_value(x + .5*tf.random.normal(tf.shape(x)), 0, 1)
m = tf.keras.Sequential([
   tf.keras.layers.Input((28,28,1)),
   tf.keras.layers.Conv2D(16, 3, padding='same', activation='relu'),
   tf.keras.layers.MaxPool2D(),
   tf.keras.layers.Conv2DTranspose(16, 3, strides=2, padding='same', activation='relu'),
   tf.keras.layers.Conv2D(1, 3, padding='same', activation='sigmoid')
])
m.compile('adam', 'binary_crossentropy')
m.fit(xn, x, epochs=3, batch_size=128, verbose=0)

for i in range(5):
   noisy_img = xn[i].numpy().squeeze()
   denoised_img = m.predict(xn[i:i+1], verbose=0).squeeze()
   plt.subplot(1,2,1)
 plt.imshow(noisy_img, cmap='gray')
   plt.axis('off')
   plt.title('Noisy')
   plt.subplot(1,2,2)
   plt.imshow(denoised_img, cmap='gray')
   plt.axis('off')
   plt.title('Denoised')
   plt.show()
```

OUTPUT:



RESULT:

Thus, the TensorFlow based application for Removing Noise from Images Using the TensorFlow is developed, executed successfully, and the model’s performance is visualized and verified.