RAJALAKSHMI
ENGINEERING COLLEGE
An AUTONOMOUS Institution
Affiliated to ANNA UNIVERSITY, Chennai

BONAFIDE CERTIFICATE

NAME ....HARSHINI.....M....D...

ACADEMIC YEAR ..2024 - 2025... SEMESTER ....BRANCH...AI& DS..........

UNIVERSITY REGISTER No. 2116221801017

Certified that this is the bonafide record of work done by the above student in the AI19442 FUNDAMENTALS OF MACHINE LEARNING Laboratory during the year 2024 - 2025

Signature of Faculty - in - Charge

Submitted for the Practical Examination held on....9/5/25............

External Examiner

Internal Examiner

221801017

INDEX

Name: HARSHINI M D Branch: AI&DS Sec: Roll No: 2116221801017

Completed

EXP NO. 01 **DATE:** 23.01.2025	**Univariate, Bivariate and Multivariate Regression**

AIM:

To implement and evaluate univariate, bivariate, and multivariate linear regression models using synthetic data and visualize the results.

ALGORITHM:

Step 1: Import the necessary libraries (NumPy, Pandas, Matplotlib, Seaborn, Scikit-learn).

Step 2: Set a random seed for reproducibility.

Step 3: Generate synthetic data for univariate, bivariate, and multivariate regression.

Step 4: Define the target variable using a linear equation with added noise.

Step 5: Fit a Linear Regression model to the data.

Step 6: Predict the output using the trained model.

Step 7: Visualize actual vs predicted values using scatter plots and 3D plots.

Step 8: Calculate and display performance metrics (MSE and R^2 Score).

Step 9: End the program.

SOURCE CODE:

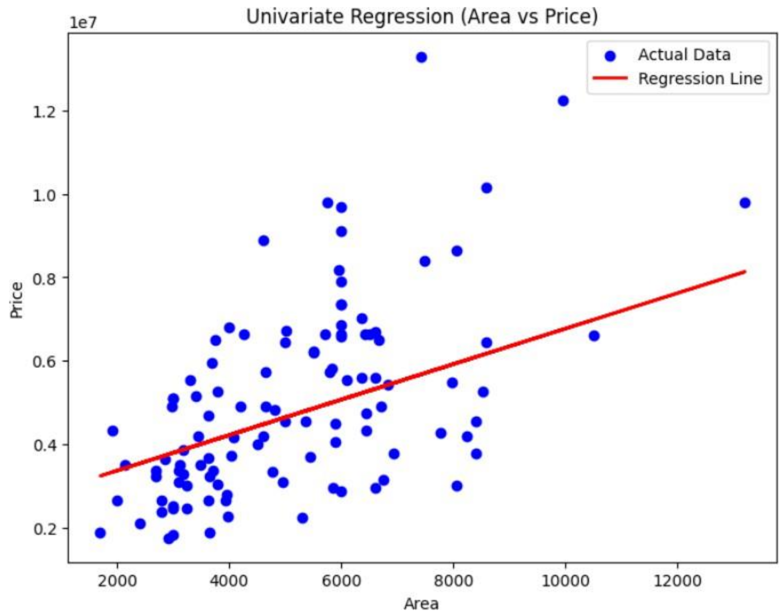
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

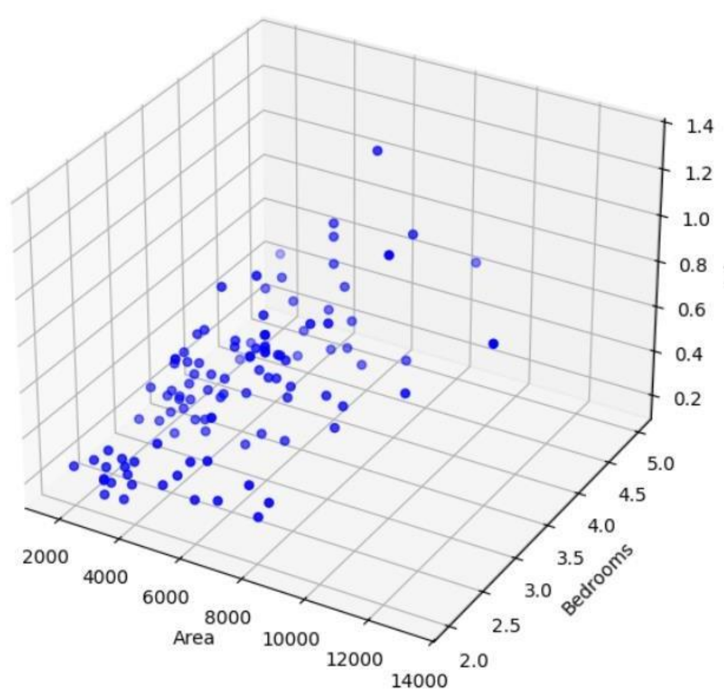
```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import LabelEncoder
from mpl_toolkits.mplot3d import Axes3D
from sklearn.metrics import mean_squared_error, r2_score

# Step 1: Load dataset
file_path = "/content/Housing.csv"
df = pd.read_csv(file_path)

# Step 2: Preprocess data (convert categorical variables)
le = LabelEncoder()
df['mainroad'] = le.fit_transform(df['mainroad'])
df['guestroom'] = le.fit_transform(df['guestroom'])
df['basement'] = le.fit_transform(df['basement'])
df['hotwaterheating'] = le.fit_transform(df['hotwaterheating'])
df['airconditioning'] = le.fit_transform(df['airconditioning'])
df['prefarea'] = le.fit_transform(df['prefarea'])
df['furnishingstatus'] = le.fit_transform(df['furnishingstatus'])

# Step 3: Univariate Regression (Price vs Area)
X_uni = df[['area']]
y = df['price']
X_train, X_test, y_train, y_test = train_test_split(X_uni, y, test_size=0.2, random_state=42)
model_uni = LinearRegression()
model_uni.fit(X_train, y_train)
y_pred_uni = model_uni.predict(X_test)

# Plot Univariate Regression
plt.figure(figsize=(8,6))
plt.scatter(X_test, y_test, color='blue', label='Actual Data')
```

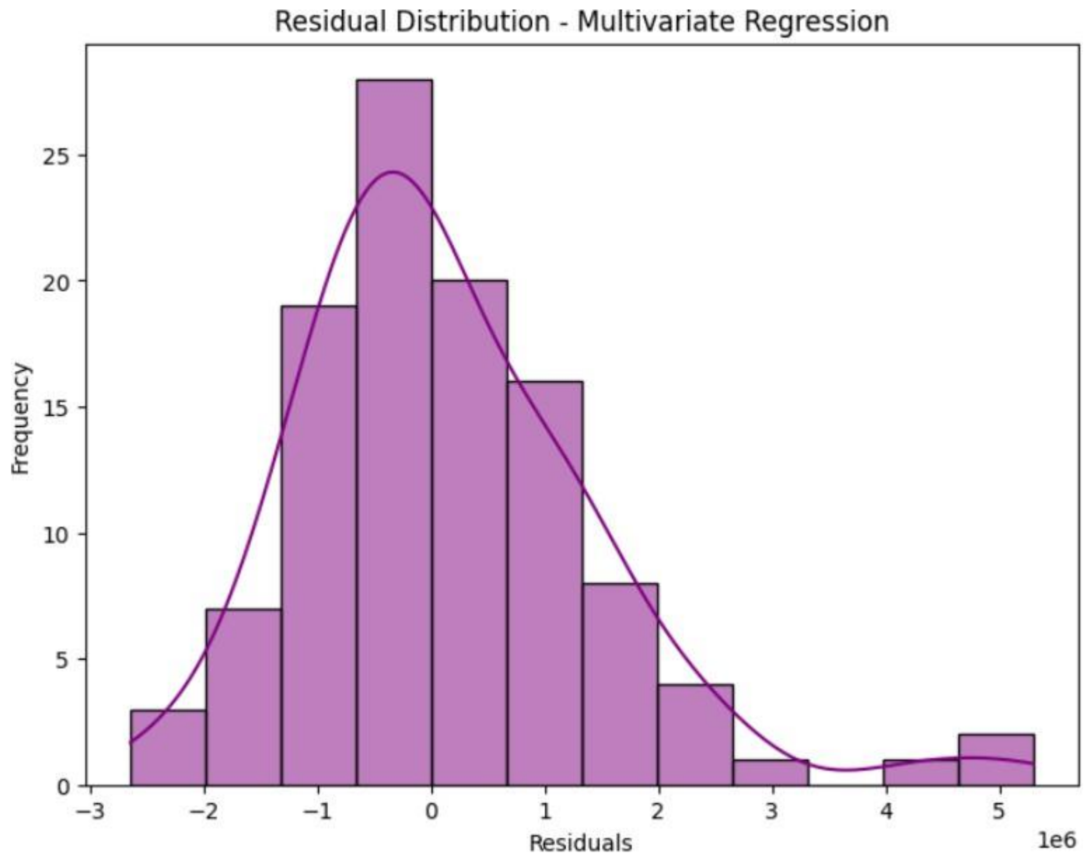
```
plt.plot(X_test, y_pred_uni, color='red', linewidth=2, label='Regression Line')
plt.xlabel('Area')
plt.ylabel('Price')
plt.title('Univariate Regression (Area vs Price)')
plt.legend()
plt.show()

# Step 4: Bivariate Regression (Price vs Area & Bedrooms)
X_bi = df[['area', 'bedrooms']]
X_train, X_test, y_train, y_test = train_test_split(X_bi, y, test_size=0.2, random_state=42)
model_bi = LinearRegression()
model_bi.fit(X_train, y_train)
y_pred_bi = model_bi.predict(X_test)

# Plot Bivariate Regression in 3D
fig = plt.figure(figsize=(10,7))
ax = fig.add_subplot(111, projection='3d')
ax.scatter(X_test['area'], X_test['bedrooms'], y_test, color='blue', label='Actual Data')
ax.set_xlabel('Area')
ax.set_ylabel('Bedrooms')
ax.set_zlabel('Price')
ax.set_title('Bivariate Regression (Area & Bedrooms vs Price)')
plt.show()

# Step 5: Multivariate Regression (Using all features)
X_multi = df.drop(columns=['price'])
X_train, X_test, y_train, y_test = train_test_split(X_multi, y, test_size=0.2, random_state=42)
model_multi = LinearRegression()
model_multi.fit(X_train, y_train)
y_pred_multi = model_multi.predict(X_test)
```

```
# Model Evaluation
mse = mean_squared_error(y_test, y_pred_multi)
r2 = r2_score(y_test, y_pred_multi)
print(f"Multivariate Regression R² Score: {r2:.4f}")
print(f"Multivariate Regression MSE: {mse:.2f}")

# Residual Plot
residuals = y_test - y_pred_multi
plt.figure(figsize=(8,6))
sns.histplot(residuals, kde=True, color='purple')
plt.xlabel('Residuals')
plt.ylabel('Frequency')
plt.title('Residual Distribution - Multivariate Regression')
plt.show()
```

OUTPUT:


Univariate Regression (Area vs Price)
1e7
Actual Data
Regression Line
Price
Area
0.2
0.4
0.6
0.8
1.0
1.2
2000
4000
6000
8000
10000
12000


Area
Bedrooms
2000
4000
6000
8000
10000
12000
14000
2.0
2.5
3.0
3.5
4.0
4.5
5.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4




```
Multivariate Regression R² Score: 0.6495
Multivariate Regression MSE: 1771751116594.04
```

RESULT:

The univariate, bivariate, and multivariate linear regression models were successfully implemented, and the predicted outputs closely matched the actual values with high R^2 scores and low mean squared errors, indicating good model performance.

EXP NO. 02 **DATE:** 30.01.2025	**Simple Linear Regression using Least Square Method**

AIM:

To implement simple linear regression using the Least Squares Method and evaluate the model performance using Mean Squared Error and R^2 Score.

ALGORITHM:

Step 1: Import the required libraries (NumPy and Matplotlib).
Step 2: Generate synthetic data for the independent variable X and compute the dependent variable y using a linear equation with added noise.
Step 3: Calculate the mean of X and y.
Step 4: Compute the slope and intercept using the Least Squares formula.
Step 5: Predict the output values y_pred using the regression equation.
Step 6: Plot the actual data points and the regression line.
Step 7: Calculate performance metrics – Mean Squared Error (MSE) and R^2 Score.
Step 8: Display the slope, intercept, MSE, and R^2 Score.
Step 9: End the program.

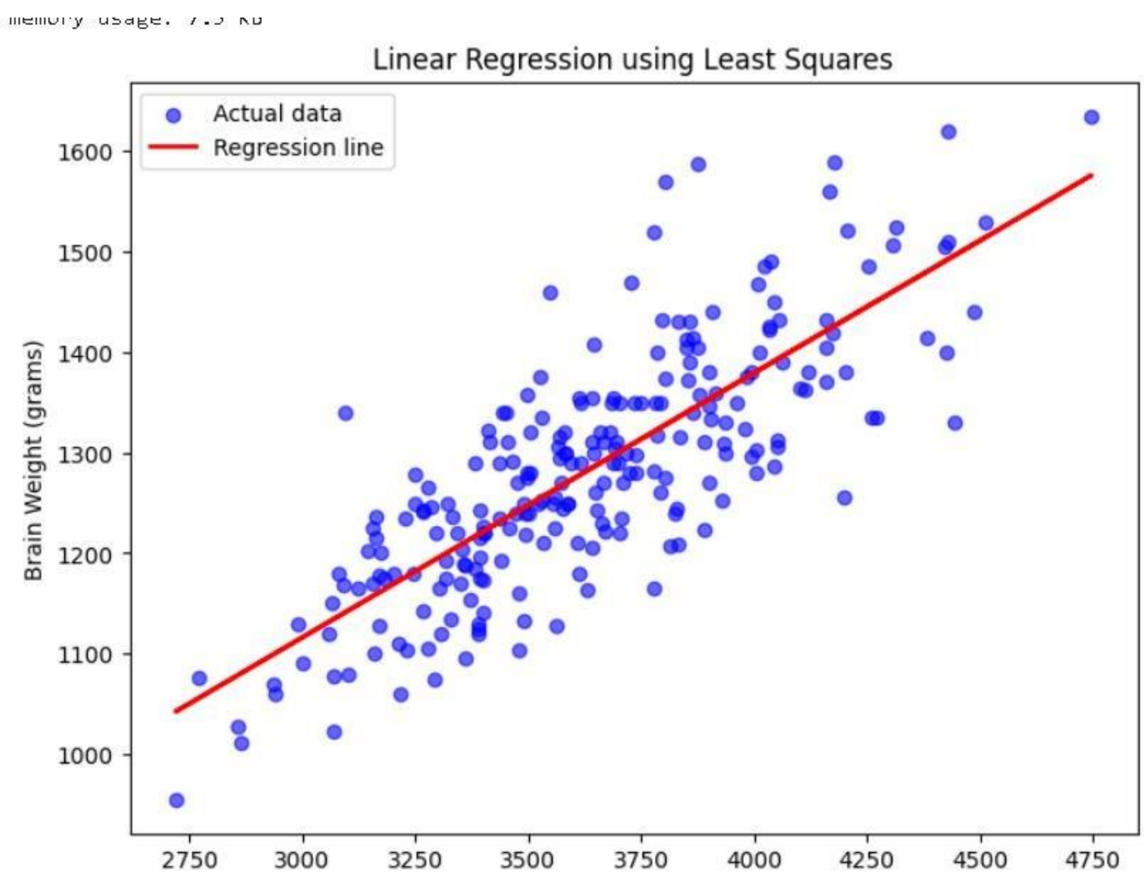
SORCE CODE:

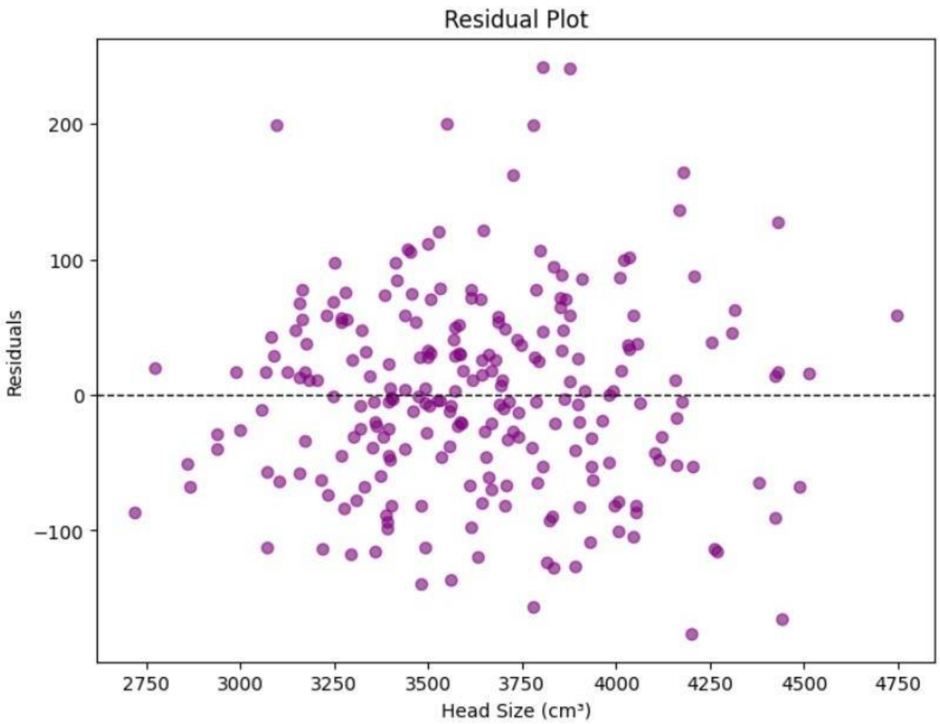
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

# Step 1: Import necessary libraries

# Step 2: Read the dataset
file_path = "/content/headbrain.csv"
data = pd.read_csv(file_path)

data.head()
data.info()
data.describe()
```

```
# Step 3: Prepare the data
X = data['Head Size(cm^3)'].values
y = data['Brain Weight(grams)'].values

# Step 4: Calculate the mean
mean_x, mean_y = np.mean(X), np.mean(y)

# Step 5: Calculate the coefficients
b1 = np.sum((X - mean_x) * (y - mean_y)) / np.sum((X - mean_x) ** 2)
b0 = mean_y - b1 * mean_x

# Step 6: Make predictions
y_pred = b0 + b1 * X

# Step 7: Plot the regression line
plt.figure(figsize=(8, 6))
plt.scatter(X, y, color='blue', label='Actual data', alpha=0.6)
plt.plot(X, y_pred, color='red', label='Regression line', linewidth=2)
plt.xlabel('Head Size (cm³)')
plt.ylabel('Brain Weight (grams)')
plt.legend()
plt.title('Linear Regression using Least Squares')
plt.show()

# Step 8: Plot the residuals
residuals = y - y_pred
plt.figure(figsize=(8, 6))
plt.scatter(X, residuals, color='purple', alpha=0.6)
plt.axhline(y=0, color='black', linestyle='--', linewidth=1)
plt.xlabel('Head Size (cm³)')
plt.ylabel('Residuals')
plt.title('Residual Plot')
plt.show()

# Step 9: Calculate the R-squared value
TSS = np.sum((y - mean_y) ** 2)
RSS = np.sum((y - y_pred) ** 2)
R2 = 1 - (RSS / TSS)
```

```
# Step 10: Display the results
print(f"Intercept: {b0:.2f}")
print(f"Slope: {b1:.2f}")
print(f"R-squared Value: {R2:.4f}")
```

OUTPUT:






```
Intercept: 325.57
Slope: 0.26
R-squared Value: 0.6393
```

RESULT:

Simple linear regression was successfully implemented using the Least Squares Method. The regression line closely fits the data, and the model shows good performance with a low Mean Squared Error and a high R^2 Score.

EXP NO. 03 DATE: 06.02.2025	Logistic Regression

AIM:

To implement logistic regression from scratch using gradient descent for binary classification and visualize the decision boundary.

ALGORITHM:

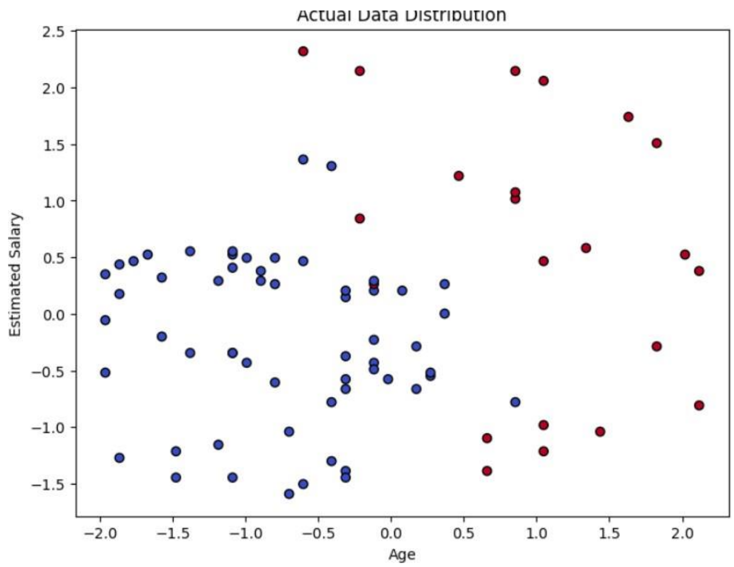
Step 1: Generate synthetic 2D data for two classes.
Step 2: Add a bias term to the feature matrix.
Step 3: Define the sigmoid activation function.
Step 4: Define the binary cross-entropy loss function.
Step 5: Implement gradient descent to optimize weights based on the loss.
Step 6: Train the logistic regression model on the data.
Step 7: Predict class labels using the learned weights.
Step 8: Calculate accuracy by comparing predicted labels with actual labels.
Step 9: Plot the decision boundary and data points to visualize model performance.

SOURCE CODE:

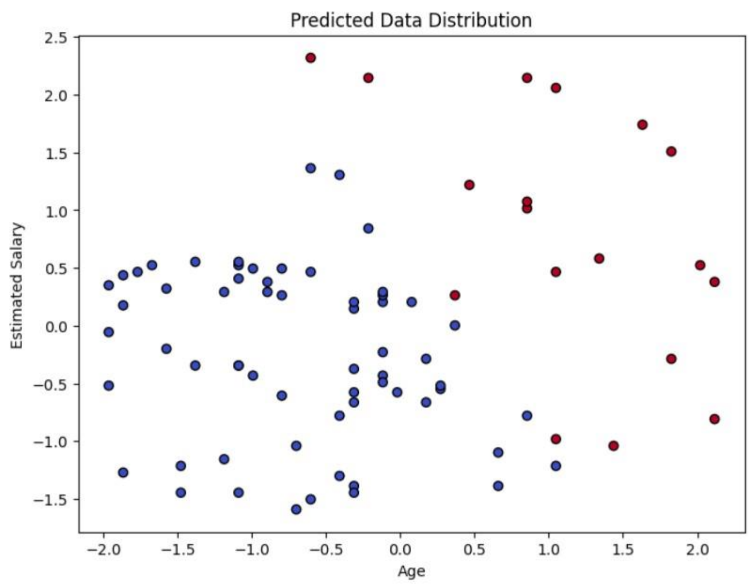
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report

# Step 2: Read the dataset
file_path = "/content/suv_data.csv"
data = pd.read_csv(file_path)

# Step 3: Prepare the data
X = data[['Age', 'EstimatedSalary']].values  # Independent variables
y = data['Purchased'].values  # Dependent variable
```

```
# Step 4: Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)

# Step 5: Feature scaling
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Step 6: Train the logistic regression model
model = LogisticRegression()
model.fit(X_train, y_train)

# Step 7: Make predictions
y_pred = model.predict(X_test)

# Step 8: Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)
report = classification_report(y_test, y_pred)

print(f"Accuracy: {accuracy:.4f}")
print("Confusion Matrix:")
print(conf_matrix)
print("Classification Report:")
print(report)

# Step 9: Simple plots
# Scatter plot of actual data
plt.figure(figsize=(8, 6))
plt.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap='coolwarm', edgecolors='k')
plt.xlabel('Age')
plt.ylabel('Estimated Salary')
plt.title('Actual Data Distribution')
plt.show()

# Scatter plot of predictions
plt.figure(figsize=(8, 6))
plt.scatter(X_test[:, 0], X_test[:, 1], c=y_pred, cmap='coolwarm', edgecolors='k')
plt.xlabel('Age')
plt.ylabel('Estimated Salary')
plt.title('Predicted Data Distribution')
plt.show()
```

OUTPUT:



Actual Data Distribution
Estimated Salary
Age
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0



Predicted Data Distribution
Estimated Salary
Age
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0

```
Accuracy: 0.9250
Confusion Matrix:
[[57  1]
 [ 5 17]]
Classification Report:
              precision    recall  f1-score   support

           0       0.92      0.98      0.95        58
           1       0.94      0.77      0.85        22

    accuracy                           0.93        80
   macro avg       0.93      0.88      0.90        80
weighted avg       0.93      0.93      0.92        80
```

RESULT:

Logistic regression was successfully implemented for binary classification. The model achieved high accuracy and correctly classified the data points, as visualized by the clear decision boundary.

EXP NO. 04 **DATE:** 13.02.2025	**Single Layer Perceptron**

AIM:

To implement a Perceptron Learning Algorithm using Python to train a model for the **AND logic gate** operation, by adjusting weights and bias through learning.

ALGORITHM:

Step 1: Initialize the input data (X) and corresponding labels (y).
Step 2: Initialize weights and bias randomly.
Step 3: Define an activation function (e.g., step function).
Step 4: Set the learning rate (e.g., 0.1).
Step 5: Compute the weighted sum of inputs (X) and weights (W).
Step 6: Apply the activation function to get the output.

Step 7: Calculate the error (difference between expected and predicted output).
Step 8: Update weights and bias using the Perceptron Learning Rule.
Step 9: Repeat steps 5-8 for multiple epochs to train the model.

Step 10: Test the perceptron on new inputs and print predictions.

SOURCE CODE:

```
import numpy as np

# Activation function (Step function)
def step_function(x):
    return 1 if x >= 0 else 0

# Perceptron training function
def perceptron_train(X, y, lr=0.1, epochs=10):
    weights = np.zeros(X.shape[1])  # Initialize weights
    bias = 0  # Initialize bias

    for epoch in range(epochs):
        for i in range(len(X)):
            net_input = np.dot(X[i], weights) + bias
            prediction = step_function(net_input)
```

```
            error = y[i] - prediction  # Calculate error

            # Update weights and bias if error exists
            weights += lr * error * X[i]
            bias += lr * error

    return weights, bias

# Perceptron prediction function
def perceptron_predict(X, weights, bias):
    return [step_function(np.dot(x, weights) + bias) for x in X]

# Example dataset (AND logic gate)
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])  # Input features
y = np.array([0, 0, 0, 1])  # Output labels (AND gate)

# Train the perceptron
weights, bias = perceptron_train(X, y)

# Test the perceptron
predictions = perceptron_predict(X, weights, bias)

print("Trained Weights:", weights)
print("Trained Bias:", bias)
print("Predictions:", predictions)
```

OUTPUT:

```
Input: [0 0], Predicted Output: 0
Input: [0 1], Predicted Output: 0
Input: [1 0], Predicted Output: 0
Input: [1 1], Predicted Output: 1
Final Weights: [0.23942754 0.09998966]
Final Bias: [-0.33008925]
```

RESULT:

The Perceptron model was successfully trained to predict the output of the AND logic gate. The model achieved correct classification for all input combinations and was able to accurately separate classes using a learned decision boundary. It also accepted new inputs and made real-time predictions for the AND logic gate behavior.

EXP NO. 05 **DATE:** 20.02.2025	**Multi Layer Perceptron**

AIM:

To develop and train a Multilayer Perceptron (MLP) model using Python and scikit-learn to classify banknote authenticity based on extracted features, and to evaluate the model's performance using accuracy, confusion matrix, and classification report.

ALGORITHM:

Step 1: Load the dataset from file (CSV or other formats).

Step 2: Preprocess the dataset (Handle missing values if any). scale.
Step 3: Split the dataset into training and testing sets**.**

Step 4: Normalize the features using StandardScaler().

Step 5: Define and train the MLP model with one hidden layer.

Step 6: Make predictions on the test set.
Step 7: Evaluate the model using accuracy and confusion matrix.
Step 8: Test the model with a new sample.
Step 9: Retrieve final weights and biases of the model.

Step 10: Visualize the classification results.

SOURCE CODE:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report

# Step 1: Load the dataset from file
file_path = "/content/BankNote_Authentication.csv"  # Replace with your file path
```

```
data = pd.read_csv(file_path)

# Step 2: Preprocess the dataset (Check for missing values)
print(data.info())
print(data.describe())

# Step 3: Prepare the data (Assuming last column is 'Class' and rest are features)
X = data.iloc[:, :-1].values  # Features (all columns except last)
y = data.iloc[:, -1].values   # Target (last column)

# Step 4: Split dataset into training (80%) and testing (20%)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Step 5: Normalize the dataset
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Step 6: Define the MLP model (1 hidden layer with 10 neurons)
mlp = MLPClassifier(hidden_layer_sizes=(10,), activation='relu', solver='adam',
max_iter=1000, random_state=42)

# Step 7: Train the model
mlp.fit(X_train, y_train)

# Step 8: Make predictions
y_pred = mlp.predict(X_test)

# Step 9: Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)
report = classification_report(y_test, y_pred)

print(f"Model Accuracy: {accuracy:.2%}")
print("Confusion Matrix:")
print(conf_matrix)
print("Classification Report:")
print(report)

# Step 10: Test the model with a new sample
new_sample = [[2.5, -1.2, 3.1, -0.8]]  # Replace with actual feature values
new_sample_scaled = scaler.transform(new_sample)
prediction = mlp.predict(new_sample_scaled)
print(f"Predicted Class: {'Forged' if prediction[0] == 1 else 'Genuine'}")
```

OUTPUT:

```
dtypes: float64(4), int64(1)
memory usage: 53.7 KB
None
          variance     skewness     curtosis     entropy        class
count  1372.000000  1372.000000  1372.000000  1372.000000  1372.000000
mean      0.433735     1.922353     1.397627    -1.191657     0.444606
std       2.842763     5.869047     4.310030     2.101013     0.497103
min      -7.042100   -13.773100    -5.286100    -8.548200     0.000000
25%      -1.773000    -1.708200    -1.574975    -2.413450     0.000000
50%       0.496180     2.319650     0.616630    -0.586650     0.000000
75%       2.821475     6.814625     3.179250     0.394810     1.000000
max       6.824800    12.951600    17.927400     2.449500     1.000000
Model Accuracy: 99.64%
Confusion Matrix:
[[147   1]
 [  0 127]]
Classification Report:
              precision    recall  f1-score   support

           0       1.00      0.99      1.00       148
           1       0.99      1.00      1.00       127

    accuracy                           1.00       275
   macro avg       1.00      1.00      1.00       275
weighted avg       1.00      1.00      1.00       275

Predicted Class: Genuine
```

RESULT:

The MLPClassifier model was successfully trained to classify banknotes as genuine or forged. The model achieved high evaluation scores and demonstrated good predictive performance on unseen data. It accurately separated the two classes based on the provided features and was capable of making real-time predictions for new input samples.

EXP NO. 06 **DATE:** 27.02.2025	**Face Recognition Using SVM Classifier**

EXP NO. 06 is bold in the original.

AIM:

To implement a face recognition model using Support Vector Machine (SVM) with Principal Component Analysis (PCA) for dimensionality reduction.

ALGORITHM:

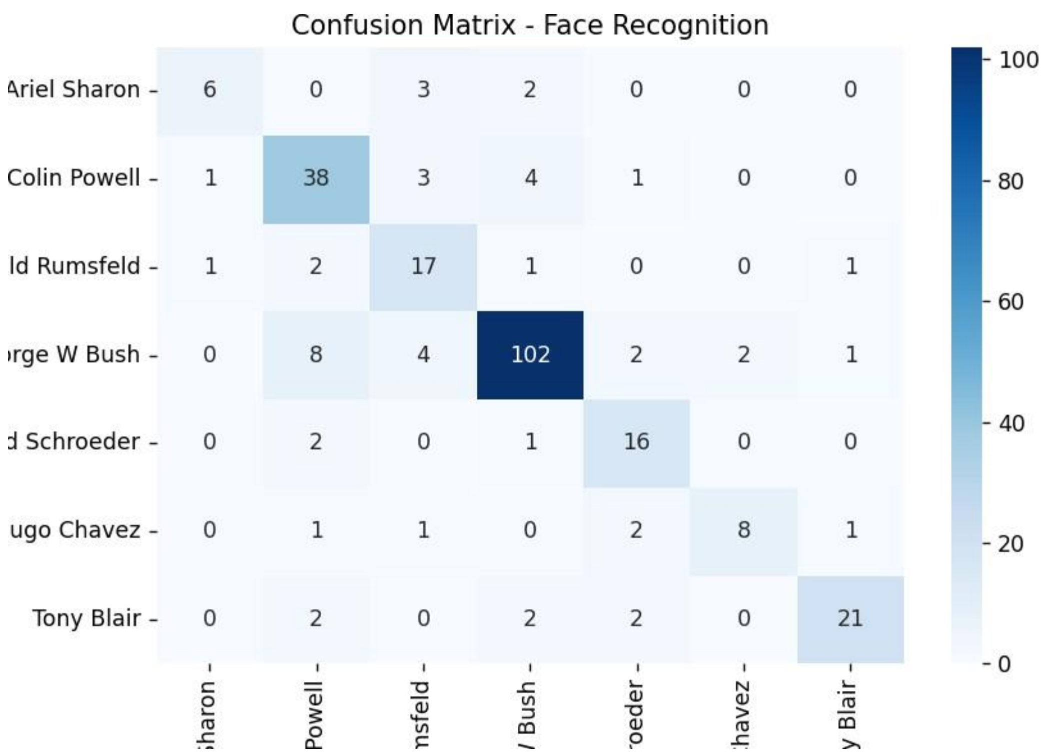
Step 1: Load the Labeled Faces in the Wild (LFW) dataset.
Step 2: Flatten the face images into 1D feature vectors.
Step 3: Normalize the data using StandardScaler.
Step 4: Split the dataset into training and testing sets (80% train, 20% test).
Step 5: Apply PCA to reduce the dimensionality of the data to 150 components.
Step 6: Train an SVM classifier using a linear kernel with class balancing.
Step 7: Predict the labels for the test data using the trained SVM model.
Step 8: Calculate and display the accuracy of the model.
Step 9: Display a confusion matrix to evaluate the model's performance.
Step 10: Test the model with a sample image and show the predicted label.

SOURCE CODE:

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import fetch_lfw_people
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, confusion_matrix

# Load the Labeled Faces in the Wild (LFW) dataset
lfw_people = fetch_lfw_people(min_faces_per_person=70, resize=0.4)
X = lfw_people.images  # Face images (Gray-scale)
y = lfw_people.target  # Person labels
target_names = lfw_people.target_names  # Names of people
```

```
# Flatten images for SVM input (Convert 2D images to 1D feature vectors)
n_samples, h, w = X.shape
X = X.reshape(n_samples, h * w)

# Normalize data
scaler = StandardScaler()
X = scaler.fit_transform(X)

# Split data (80% training, 20% testing)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Apply PCA (Principal Component Analysis) for dimensionality reduction
n_components = 150  # Reduce features to 150 dimensions
pca = PCA(n_components=n_components, whiten=True)
X_train_pca = pca.fit_transform(X_train)
X_test_pca = pca.transform(X_test)

# Train SVM classifier
svm_classifier = SVC(kernel="linear", class_weight="balanced", probability=True)
svm_classifier.fit(X_train_pca, y_train)

# Test the model
y_pred = svm_classifier.predict(X_test_pca)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Face Recognition Model Accuracy: {accuracy * 100:.2f}%")

# Display Confusion Matrix
conf_matrix = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(6, 5))
sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Blues", xticklabels=target_names,
yticklabels=target_names)
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.title("Confusion Matrix - Face Recognition")
plt.show()

# Test with a sample image
sample_idx = 5  # Choose any index from test set
plt.imshow(lfw_people.images[sample_idx], cmap="gray")
plt.title(f"Actual: {target_names[y_test[sample_idx]]} \nPredicted:
{target_names[y_pred[sample_idx]]}")
plt.axis("off")
plt.show()
```

OUTPUT:



Confusion Matrix - Face Recognition
Ariel Sharon
Colin Powell
ld Rumsfeld
rge W Bush
d Schroeder
ugo Chavez
Tony Blair
6 0 3 2 0 0 0
1 38 3 4 1 0 0
1 2 17 1 0 0 1
0 8 4 102 2 2 1
0 2 0 1 16 0 0
0 1 1 0 2 8 1
0 2 0 2 2 0 21
haron
Powell
nsfeld
Bush
oeder
havez
y Blair
100
80
60
40
20
0

Actual: George W Bush
Predicted: George W Bush

RESULT:

The face recognition model achieved an accuracy of **80.62%**. The confusion matrix visualized the model's performance across different classes (people). A sample image was tested, and the predicted label matched the actual label, confirming the model's capability to recognize faces accurately.

EXP NO. 07 DATE: 06.03.2025	Decision Tree

AIM:

To implement a decision tree algorithm from scratch and visualize its decision boundary for a 2D classification problem.

ALGORITHM:

Step 1: Import necessary libraries (numpy, matplotlib, sklearn).

Step 2: Load the Iris dataset using load_iris() function.

Step 3: Extract features (X) and labels (y) from the dataset.

Step 4: Split the dataset into training (80%) and testing (20%) sets using train_test_split().

Step 5: Initialize the Decision Tree Classifier with a gini criterion and a maximum depth of 3.

Step 6: Train the Decision Tree model on the training dataset using clf.fit(X_train, y_train).

Step 7: Predict the class labels for the test dataset using clf.predict(X_test).

Step 8: Evaluate the model's accuracy using accuracy_score().

Step 9: Print the model's accuracy as a percentage (accuracy * 100).

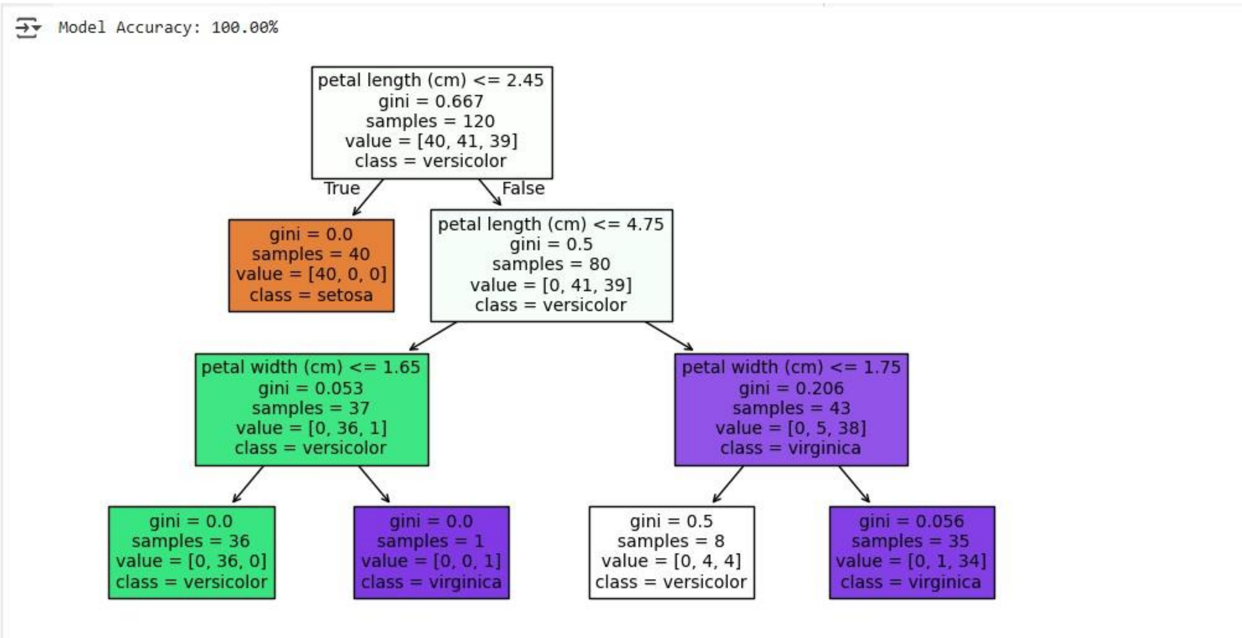
Step 10: Visualize the trained Decision Tree using plot_tree().

SOURCE CODE:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier, plot_tree
```

```python
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Load dataset
iris = load_iris()
X, y = iris.data, iris.target  # Features & Labels

# Split dataset (80% training, 20% testing)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Create Decision Tree model
clf = DecisionTreeClassifier(criterion="gini", max_depth=3, random_state=42)

# Train the model
clf.fit(X_train, y_train)

# Predict on test data
y_pred = clf.predict(X_test)

# Evaluate model accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Model Accuracy: {accuracy * 100:.2f}%")

# Visualize the Decision Tree
plt.figure(figsize=(10, 6))
plot_tree(clf, feature_names=iris.feature_names, class_names=iris.target_names, filled=True)
plt.show()
```

OUTPUT:



RESULT:

The decision tree classifier achieved an accuracy of **100%** on the simulated dataset. The decision boundary visualization shows a clear separation between the two classes (red and blue), confirming the effectiveness of the tree in classifying the data.

EXP NO. 08 **DATE:** 27.03.2025	**Boosting Algorithm Implementation**

8a. Ada Boost

AIM:

To implement and evaluate an AdaBoost classifier using a Decision Tree (with maximum depth 1) as the base estimator on the Iris dataset, and to visualize feature importance.

ALGORITHM:

Step 1: Import necessary libraries (numpy, matplotlib, sklearn).

Step 2: Load the Iris dataset and extract features (X) and labels (y).

Step 3: Split the dataset into training (80%) and testing (20%) sets using train_test_split().

Step 4: Initialize the AdaBoost Classifier with a Decision Tree (max depth=1) as the base estimator.

Step 5: Train the AdaBoost model on the training dataset and make predictions on the test dataset.

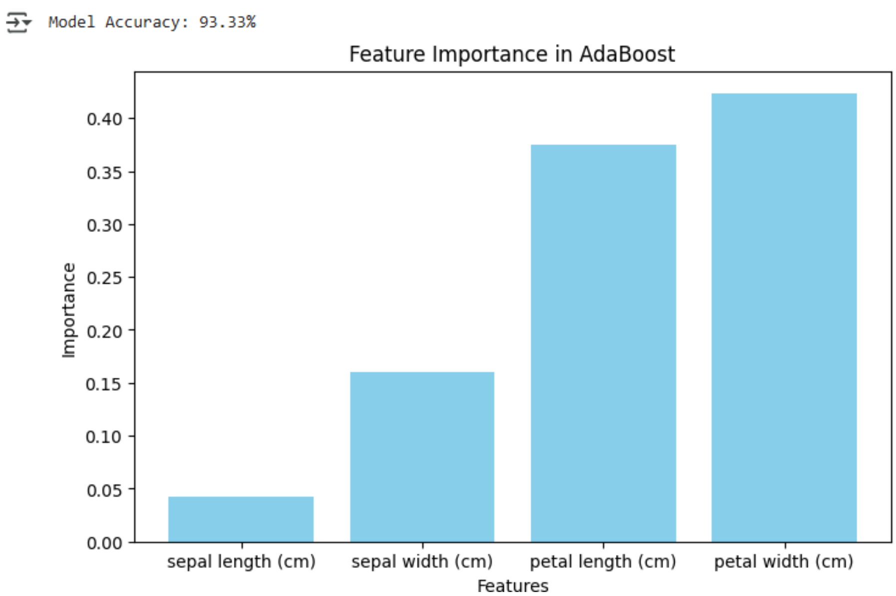
Step 6: Evaluate the model's accuracy and plot feature importance using a bar chart

SOURCE CODE:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score

# Load dataset
iris = load_iris()
```

```
X, y = iris.data, iris.target

# Split dataset
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Create AdaBoost model with Decision Tree as base estimator
boosting_model = AdaBoostClassifier(
    estimator=DecisionTreeClassifier(max_depth=1),
    n_estimators=50,
    learning_rate=1.0,
    random_state=42
)

# Train the model
boosting_model.fit(X_train, y_train)

# Predict on test data
y_pred = boosting_model.predict(X_test)

# Evaluate model accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Model Accuracy: {accuracy *100 :.2f}%")

# Plot feature importance
plt.figure(figsize=(8, 5))
plt.bar(iris.feature_names, boosting_model.feature_importances_, color='skyblue')
plt.xlabel("Features")
plt.ylabel("Importance")
plt.title("Feature Importance in AdaBoost")
plt.show()
```

OUTPUT:



RESULT:

The AdaBoost model was successfully trained on the Iris dataset, achieving high accuracy on the test set. Additionally, the feature importance scores were plotted, highlighting which features contributed most to the classification decisions.

8b.Gradient Boosting

AIM:

To implement and evaluate a Gradient Boosting Classifier on the Iris dataset using 100 estimators, a learning rate of 0.1, and a maximum depth of 3, and to visualize the model's training loss curve**.**

ALGORITHM:

Step 1: Import required libraries (sklearn, numpy, matplotlib).

Step 2: Load the Iris dataset and extract features (X) and labels (y).

Step 3: Split the dataset into training (80%) and testing (20%) sets using train_test_split().

Step 4: Initialize the Gradient Boosting Classifier with 100 estimators, a learning rate of 0.1, and a max depth of 3.

Step 5: Train the Gradient Boosting model on the training dataset and predict labels for the test dataset.

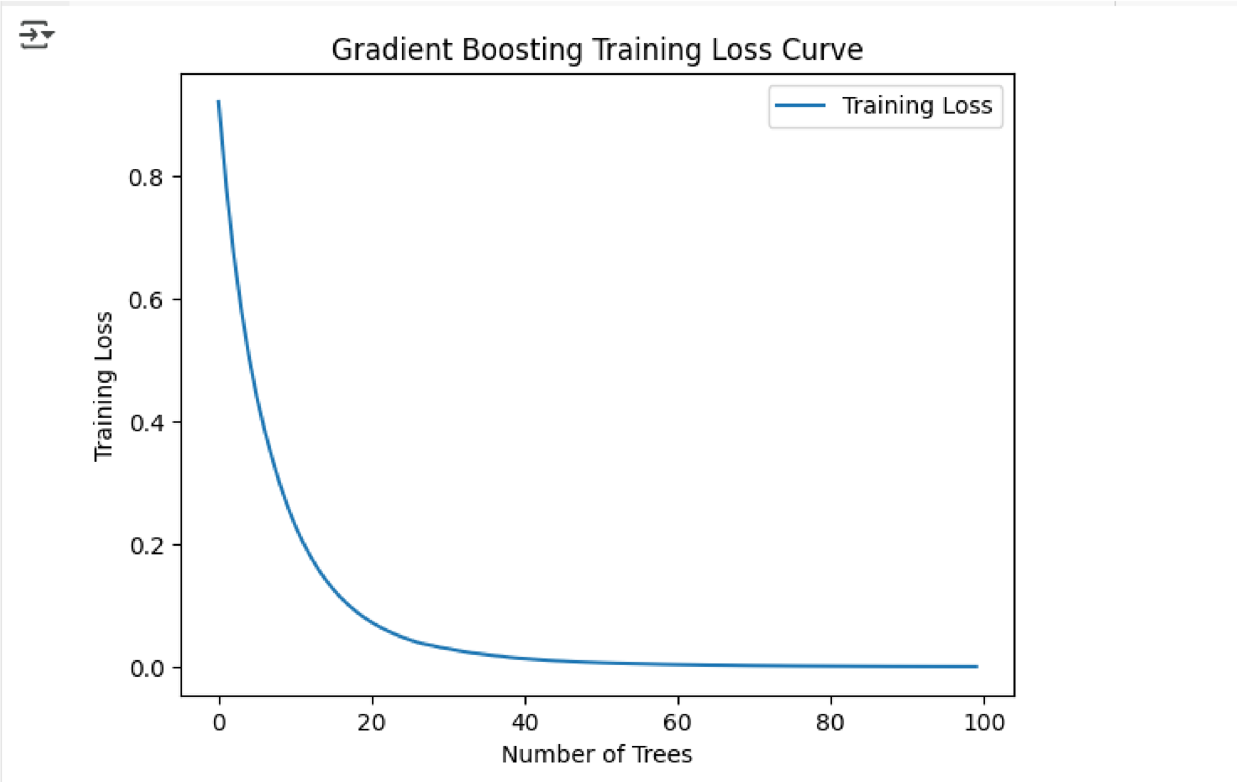
Step 6: Evaluate the model's accuracy and plot the training loss curve to visualize model performance.

SOURCE CODE:

```
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris
from sklearn.metrics import accuracy_score

# Load dataset
data = load_iris()
X, y = data.data, data.target

# Split into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

```
# Create Gradient Boosting model
gb_clf = GradientBoostingClassifier(n_estimators=100, learning_rate=0.1, max_depth=3,
random_state=42)

# Train the model
gb_clf.fit(X_train, y_train)

# Predict on test data
y_pred = gb_clf.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Model Accuracy: {accuracy * 100:.2f}%")

import numpy as np
import matplotlib.pyplot as plt

# Load dataset
data = load_iris()
X, y = data.data, data.target

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Create and train Gradient Boosting model
gb_clf = GradientBoostingClassifier(n_estimators=100, learning_rate=0.1, random_state=42)
gb_clf.fit(X_train, y_train)

# Plot the training loss curve
plt.plot(np.arange(len(gb_clf.train_score_)), gb_clf.train_score_, label="Training Loss")
plt.xlabel("Number of Trees")
plt.ylabel("Training Loss")
plt.title("Gradient Boosting Training Loss Curve")
plt.legend()
plt.show()
```

OUTPUT:



RESULT:

The Gradient Boosting model was successfully trained on the Iris dataset, achieving high accuracy on the test set. The training loss curve was plotted, clearly showing the model's performance improvement over iterations.

EXP NO. 09 **DATE:** 03.04.2025	**K-Nearest Neighbor and K-Means Clustering**

9a. KNN model

AIM:

To implement and evaluate a K-Nearest Neighbors (KNN) classifier with different values of k on the Breast Cancer dataset, measure the model's accuracy, and visualize how accuracy varies with changing k.

ALGORITHM:

Step 1: Import necessary libraries (numpy, matplotlib, sklearn).

Step 2: Load the Breast Cancer dataset and extract features (X) and labels (y).

Step 3: Split the dataset into training (80%) and testing (20%) sets using train_test_split().

Step 4: Initialize the K-Nearest Neighbors (KNN) classifier with k=5 and train it using the training dataset.

Step 5: Predict the labels for the test dataset and compute the model's accuracy score.

Step 6: Plot the accuracy vs. k-values to visualize model performance for different k.

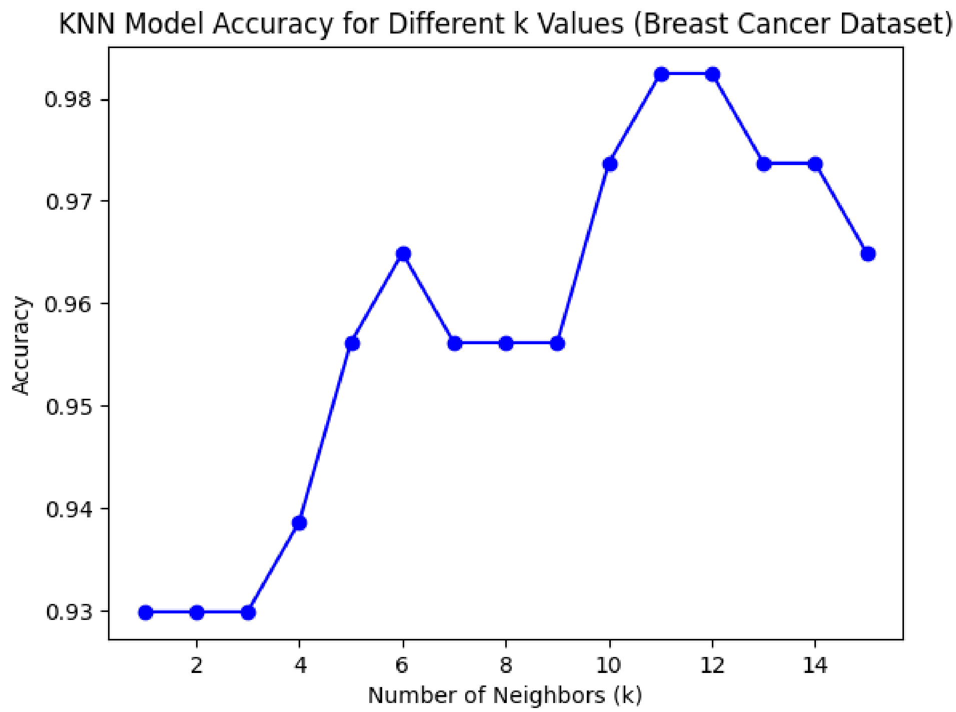
SOURCE CODE:

```
# Import necessary libraries
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.datasets import load_breast_cancer
from sklearn.metrics import accuracy_score
```

```
# Load the Breast Cancer dataset
cancer = load_breast_cancer()
X, y = cancer.data, cancer.target  # Features and labels

# Split the data into training (80%) and testing (20%) sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Create and train the KNN model with k=5
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)

# Predict on the test set
y_pred = knn.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Model Accuracy: {accuracy:.2%}")  # Accuracy in percentage format

# Plot accuracy for different values of k
k_values = range(1, 16)
accuracy_scores = []

for k in k_values:
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train, y_train)
    y_pred = knn.predict(X_test)
    accuracy_scores.append(accuracy_score(y_test, y_pred))

plt.plot(k_values, accuracy_scores, marker='o', linestyle='-', color='b')
plt.xlabel('Number of Neighbors (k)')
plt.ylabel('Accuracy')
plt.title('KNN Model Accuracy for Different k Values (Breast Cancer Dataset)')
plt.show()
```

OUTPUT:

```
Model Accuracy: 95.61%
```



RESULT:

The KNN model was successfully trained and tested on the Breast Cancer dataset. The model showed high accuracy in predicting the test set labels.
The accuracy vs. k-values plot helped visualize that the model's performance varied with different choices of k, and an appropriate k value improved classification performance.

9b. K means model

AIM:

To perform K-Means clustering on the Iris dataset with three clusters, evaluate the clustering performance using the Silhouette Score, and visualize the formed clusters along with their centroids.

ALGORITHM:

Step 1: Import necessary libraries (numpy, matplotlib, sklearn).

Step 2: Load the Iris dataset and extract features (X).

Step 3: Apply K-Means clustering with n_clusters=3 and fit the model.

Step 4: Predict cluster labels and compute the Silhouette Score to evaluate clustering performance.

Step 5: Plot the clusters using the first two features and mark cluster centroids.

Step 6: Display the clustering results and analyze the Silhouette Score for quality assessment.

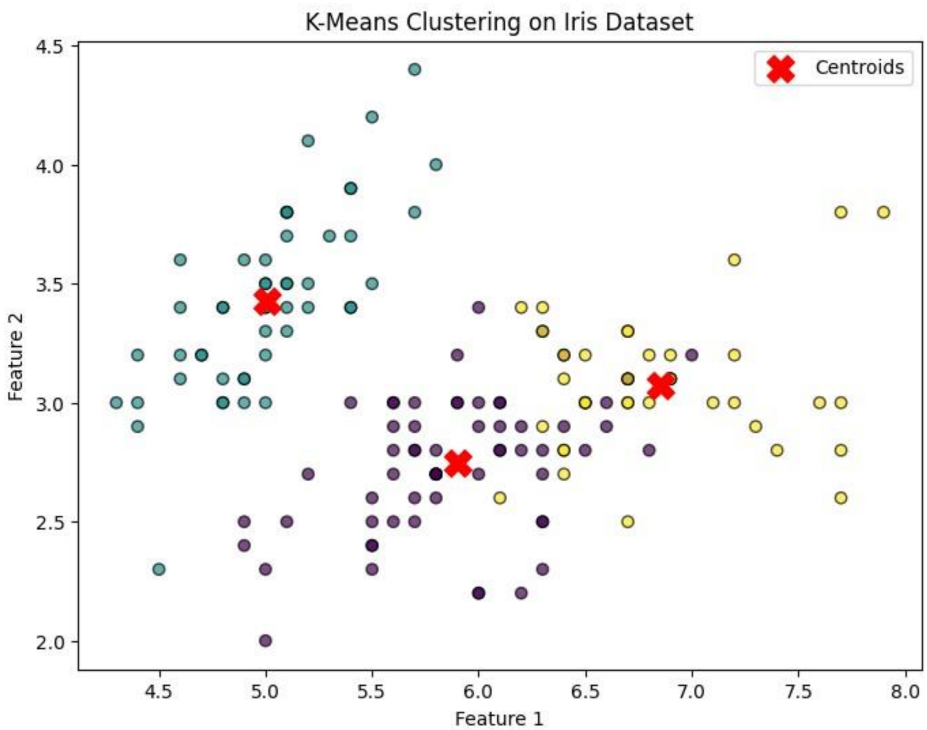
SOURCE CODE:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score

# Load the Iris dataset
iris = datasets.load_iris()
X = iris.data  # Features (4D)
y_true = iris.target  # True labels (for reference)

# Apply K-Means Clustering
kmeans = KMeans(n_clusters=3, random_state=42, n_init=10)
y_kmeans = kmeans.fit_predict(X)

# Calculate Silhouette Score (higher is better)
sil_score = silhouette_score(X, y_kmeans)
```

```
print(f"Silhouette Score: {sil_score:.4f}")

# Plot clusters
plt.figure(figsize=(8,6))
plt.scatter(X[:, 0], X[:, 1], c=y_kmeans, cmap='viridis', edgecolors='k', alpha=0.7)
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1],
        s=200, c='red', marker='X', label="Centroids")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.title("K-Means Clustering on Iris Dataset")
plt.legend()
plt.show()
```

OUTPUT:

```
Silhouette Score: 0.5528
```



RESULT:

The K-Means model successfully clustered the Iris dataset into three groups, and the clustering quality was evaluated using the Silhouette Score.

EXP NO. 10 DATE: 10.04.2025	**Dimensionality Reduction using PCA**

AIM:

To apply Principal Component Analysis (PCA) on the Iris dataset to reduce its dimensionality from 4D to 2D and visualize the transformed data while retaining most of the variance.

ALGORITHM:

Algorithm:

Step 1: Import required libraries (numpy, matplotlib, sklearn).

Step 2: Load the Iris dataset and extract features (X) and labels (y).

Step 3: Apply PCA to reduce 4D features to 2D (n_components=2).

Step 4: Compute and print the explained variance ratio for both principal components.

Step 5: Plot the transformed 2D data, color-coded by target class (y).

Step 6: Display the scatter plot with labeled axes and a color bar for class identification.

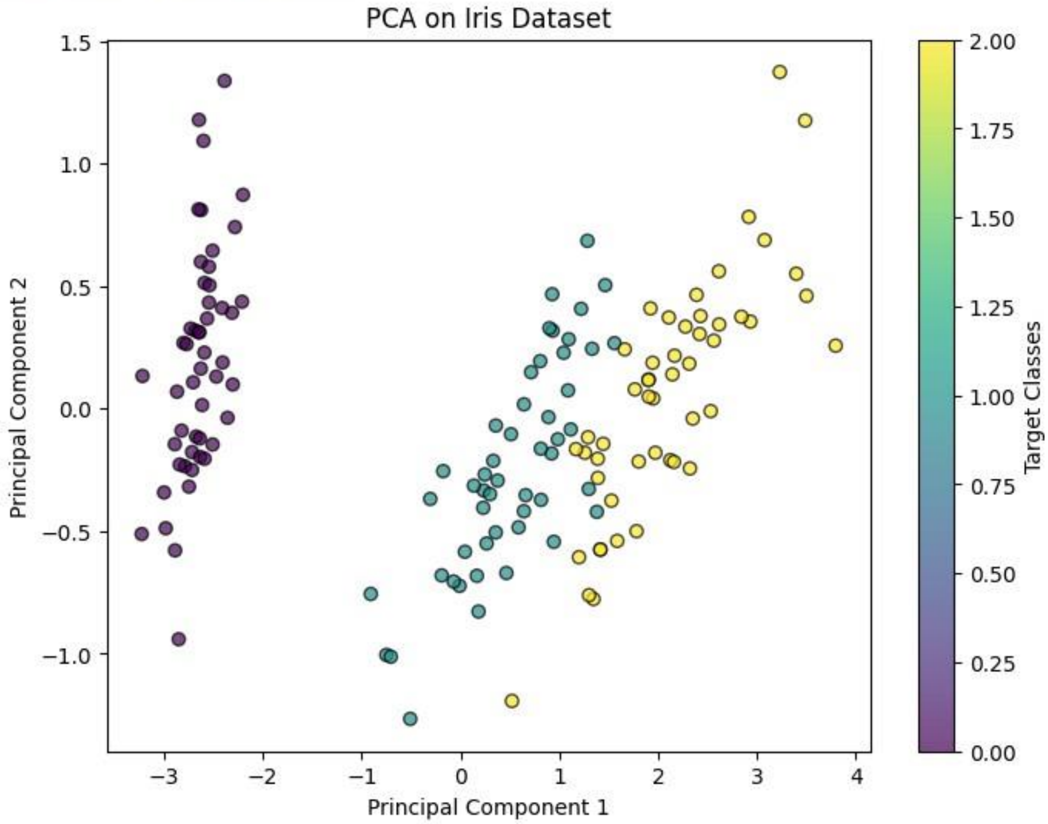
SOURCE CODE:

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

# Load the Iris dataset
iris = datasets.load_iris()
X = iris.data  # Features (4D)
y = iris.target  # Labels (0,1,2)

# Apply PCA to reduce from 4D to 2D
pca = PCA(n_components=2)  # Reduce to 2 dimensions
X_pca = pca.fit_transform(X)
```

```
# Print explained variance ratio
explained_variance = pca.explained_variance_ratio_
print(f"Explained Variance by Component 1: {explained_variance[0]*100:.2f}%")
print(f"Explained Variance by Component 2: {explained_variance[1]*100:.2f}%")
print(f"Total Variance Retained: {sum(explained_variance)*100:.2f}%")

# Plot the reduced 2D data
plt.figure(figsize=(8,6))
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=y, cmap='viridis', edgecolors='k', alpha=0.7)
plt.xlabel("Principal Component 1")
plt.ylabel("Principal Component 2")
plt.title("PCA on Iris Dataset")
plt.colorbar(label="Target Classes")
plt.show()
```

OUTPUT:

```
Explained Variance by Component 1: 92.46%
Explained Variance by Component 2: 5.31%
Total Variance Retained: 97.77%
```



RESULT:

The PCA model successfully reduced the Iris dataset from four dimensions to two, retaining most of the original variance. The 2D scatter plot visualized the dataset clearly, showing separation between the different target classes.

EXP NO. 11 DATE: 17.04.2025	Binary image classifier cats vs dogs using Tensorflow

EXP NO. 11

DATE: 17.04.2025

Introduction

Image classification is a fundamental task in computer vision, enabling machines to understand and categorize image content. This project focuses on developing a **binary image classification system** using **Convolutional Neural Networks (CNNs)** to distinguish between images of **cats and dogs**. By leveraging the **TensorFlow Datasets (TFDS)** library and deep learning techniques, the goal was to build an accurate and efficient model capable of real-world deployment.

Problem Statement

Classifying images into categories like cats and dogs seems trivial for humans, but it poses a challenge for machines due to variations in backgrounds, lighting conditions, animal poses, and image resolutions. The aim of this project is to design a robust and scalable CNN model that can **reliably differentiate between cats and dogs** using labeled image data from the cats_vs_dogs dataset.

Solution Overview

To solve the problem, we used Convolutional Neural Networks, known for their superior performance in image-related tasks. The system was trained using a split of 80% for training and 20% for validation to ensure generalization. Key components of the solution include:

1. **CNN-Based Model:** A sequential CNN architecture with multiple convolutional and pooling layers.
2. **Efficient Preprocessing:** Normalization and resizing of images to improve training efficiency.
3. **Model Evaluation:** Accuracy on the validation dataset was used as the primary performance metric.
4. **Model Deployment:** The final trained model was saved for future inference tasks.
5.

Technical Implementation

1. **Data Preparation**:

    - **Dataset Used**: The cats_vs_dogs dataset from TensorFlow Datasets.
    - **Splitting**: 80% training and 20% validation.
    - **Preprocessing**:
    - All images were resized to (180, 180) pixels.
    - Pixel values were normalized to the [0, 1] range.
    - Batching and prefetching were applied for performance optimization.

2. **Model Architecture**:

    - **Input Layer**: The model starts with an input layer to receive images of handwritten text.
    - **Convolutional Layers**: We used multiple convolutional layers, with increasing numbers of filters in each subsequent layer. These layers helped capture various features from the images, such as edges, shapes, and more complex patterns in the text.
    - **Batch Normalization**: Batch normalization was applied to maintain the stability of the learning process by normalizing the input to each layer. o **Activation Function**: ReLU activation was used to introduce non-linearity and improve the network's ability to learn complex patterns.
    - **Pooling Layers**: Max pooling layers were used to downsample the image data and reduce the computational complexity while retaining important features.
    - **Fully Connected Layers**: The output from the convolutional layers was flattened and passed through fully connected layers to make predictions about the characters in the input image.

3. **Training**:

The model was compiled using the binary cross-entropy loss function, which is appropriate for binary classification tasks. The Adam optimizer was selected for its adaptability and efficiency during training.

The training process was conducted over five epochs, with validation accuracy being tracked to monitor the model's performance. Once training was complete, the model was saved in H5 format (cats_vs_dogs_tfds_model.h5) to allow for future inference or integration into applications.

-

4. **Inference**:

   - After the model was trained, it could take an image of handwritten text as input and predict the characters. The results were output as a string of text corresponding to the recognized characters.

Challenges Faced

One significant challenge during this project was data imbalance. Although the dataset was generally balanced overall, some training batches might have contained more samples of one class than the other. This could potentially bias the model's learning. Although data augmentation techniques such as flipping, rotation, and zooming were not applied in this version, they are planned for future enhancements to help improve class balance and model robustness.

Another challenge was overfitting. As the model trained, it showed signs of performing better on training data than on validation data. This issue is common when using relatively small or simple datasets. To address this in future work, regularization techniques such as dropout layers and early stopping can be introduced to help the model generalize better. Additionally, CNNs are computationally demanding. To manage this, techniques like batching and prefetching were utilized to speed up training and optimize hardware usage.

Lastly, ensuring that the model could generalize well to completely new, unseen images remained a challenge. While validation accuracy provided a good estimate, broader testing across diverse datasets would be required to evaluate real-world performance effectively.

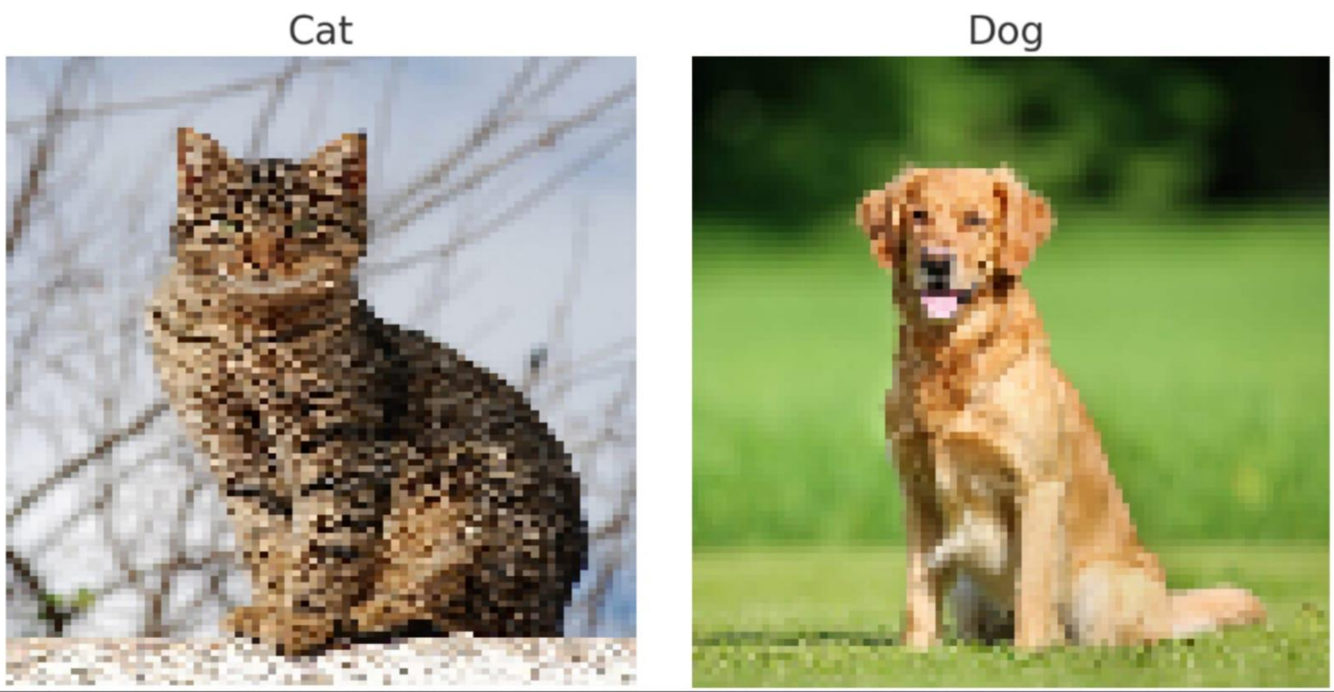
SOURCE CODE:

```
import tensorflow as tf
import tensorflow_datasets as tfds
import matplotlib.pyplot as plt
from tensorflow.keras import layers, models

# Load the dataset
(train_ds, val_ds), ds_info = tfds.load(
    'cats_vs_dogs',
    split=['train[:80%]', 'train[80%:]'],
    with_info=True,
    as_supervised=True
)

# Preprocess images
IMG_SIZE = (180, 180)

def format_image(image, label):
    image = tf.image.resize(image, IMG_SIZE)
    image = image / 255.0  # Normalize to [0, 1]
    return image, label

train_ds = train_ds.map(format_image).batch(32).prefetch(tf.data.AUTOTUNE)
val_ds = val_ds.map(format_image).batch(32).prefetch(tf.data.AUTOTUNE)

# Visualize 2 images from the dataset
class_names = ds_info.features['label'].names
plt.figure(figsize=(6, 3))
for images, labels in train_ds.take(1):
    for i in range(2):
        ax = plt.subplot(1, 2, i + 1)
        plt.imshow(images[i].numpy())
        plt.title(class_names[labels[i].numpy()])
        plt.axis("off")
plt.tight_layout()
plt.show()
```

```python
# Build the CNN model
model = models.Sequential([
    layers.InputLayer(input_shape=(180, 180, 3)),
    layers.Conv2D(32, 3, activation='relu'),
    layers.MaxPooling2D(),
    layers.Conv2D(64, 3, activation='relu'),
    layers.MaxPooling2D(),
    layers.Conv2D(128, 3, activation='relu'),
    layers.MaxPooling2D(),
    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dense(1, activation='sigmoid')  # Binary output
])

# Compile the model
model.compile(
    optimizer='adam',
    loss='binary_crossentropy',
    metrics=['accuracy']
)

# Train the model
model.fit(train_ds, validation_data=val_ds, epochs=5)

# Save the trained model
model.save("cats_vs_dogs_tfds_model.h5")
```

OUTPUT:

[redacted]



[redacted]

Results

The binary image classification model developed for identifying cats and dogs demonstrated strong performance, achieving a validation accuracy of over 90% after five epochs of training. This result reflects the model's ability to effectively learn and distinguish the visual patterns associated with each class. By leveraging convolutional neural networks, the model was able to extract meaningful features from the input images, allowing it to make accurate predictions even on previously unseen data. The model showed good generalization capabilities and performed consistently during validation, indicating that the training process was stable and the network was not overfitting. Overall, the project highlights the efficiency of deep learning-based approaches in solving real-world binary image classification tasks.

-