Decision trees

December 18th, 2024

AIM: PERFORM DECISION TREES ON A DATASET AND ANALYSE THE PERFORMANCE

DESCRIPTION: The CART (Classification and Regression Trees) algorithm constructs decision trees by selecting the best features to split the data based on criteria like Gini impurity (for clas- sification) or mean squared error (for regression). It recursively partitions the data into branches, creating decision paths that lead to final predictions or classifications at the leaf nodes. The process continues until a stopping condition is met, and the resulting tree can be pruned for simplicity and generalization.

PROPERTIES OF THE MODEL:

a. Non-Parametric model

b. White Box model (Easy to understand)

c. Ability to work with non-linear data

d. It is a giant if-else based model

ALGORITHM:

Step 1: Load necessary libraries and dataset

Step 2: Check for missing data

Step 3: Use label encoder on categorical columns

Step 4: Seperate the target column

Step 5: Divide dataset into training and testing sets

Step 6: Fit the decision tree model

Step 7: Plot the confusion matrix and visualize the classified data

Step 8: Plot the tree

Step 9: Iterate over various parameters and observe the training and testing accuracy

[90]:
```
import numpy as np
import pandas as pd
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report,
    confusion_matrix
import matplotlib.pyplot as plt
```

```
import seaborn as sns
from sklearn.preprocessing import LabelEncoder
```

[91]:
```
data = pd.read_csv ('https://github.com/YBIFoundation/Dataset/raw/main/Stars.csv')
```

[93]:
```
data.head()
```

[93] :

	Temperature (K)	Luminosity (L/Lo)	Radius (R/Ro)	Absolute magnitude (Mv)	\
0	3068	0.002400	0.1700	16.12	
1	3042	0.000500	0.1542	16.60	
2	2600	0.000300	0.1020	18.70	
3	2800	0.000200	0.1600	16.65	
4	1939	0.000138	0.1030	20.06	

	Star type	Star category	Star color	Spectral Class
0	0	Brown Dwarf	Red	M
1	0	Brown Dwarf	Red	M
2	0	Brown Dwarf	Red	M
3	0	Brown Dwarf	Red	M
4	0	Brown Dwarf	Red	M

[94] :
```
data.isnull().sum() # Checking for missing values
```

[94] :
```
Temperature (K)           0
Luminosity (L/Lo)         0
Radius (R/Ro)             0
Absolute magnitude (Mv)   0
Star type                 0
Star category             0
Star color                0
Spectral Class            0
dtype: int64
```

[95] :
```
original_class_names = data['Spectral Class'].unique()
```

[96] :
```
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
# Loop through columns and apply LabelEncoder to categorical columns
for column in data.columns:
    if data[column].dtype == 'object':    # Identify categorical columns
        data[column] = le.fit_transform(data[column])
```

[97] :
```
X = data.drop("Spectral Class", axis=1)  # Features
y = data["Spectral Class"]    # Target
```

[98] :
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    random_state=42)
```

[99] :
```
dt = DecisionTreeClassifier(criterion="gini", max_depth=5, random_state=42)
dt.fit(X_train,    y_train)
```

[99] : DecisionTreeClassifier(max_depth=5, random_state=42)

[100] :
```
# Evaluate Model Performance
y_pred = dt.predict(X_test)
print("Accuracy:", accuracy_score(y_test, y_pred))
print("Classification Report:\n", classification_report(y_test, y_pred))
```

```
Accuracy: 0.8958333333333334
Classification Report:
                precision    recall  f1-score   support

           0       0.33      0.50      0.40         2
           1       0.91      1.00      0.95        10
           2       0.50      1.00      0.67         2
           4       0.00      0.00      0.00         3
           5       1.00      1.00      1.00        21
           6       1.00      0.90      0.95        10

    accuracy                           0.90        48
   macro avg       0.62      0.73      0.66        48
weighted avg       0.87      0.90      0.88        48
```

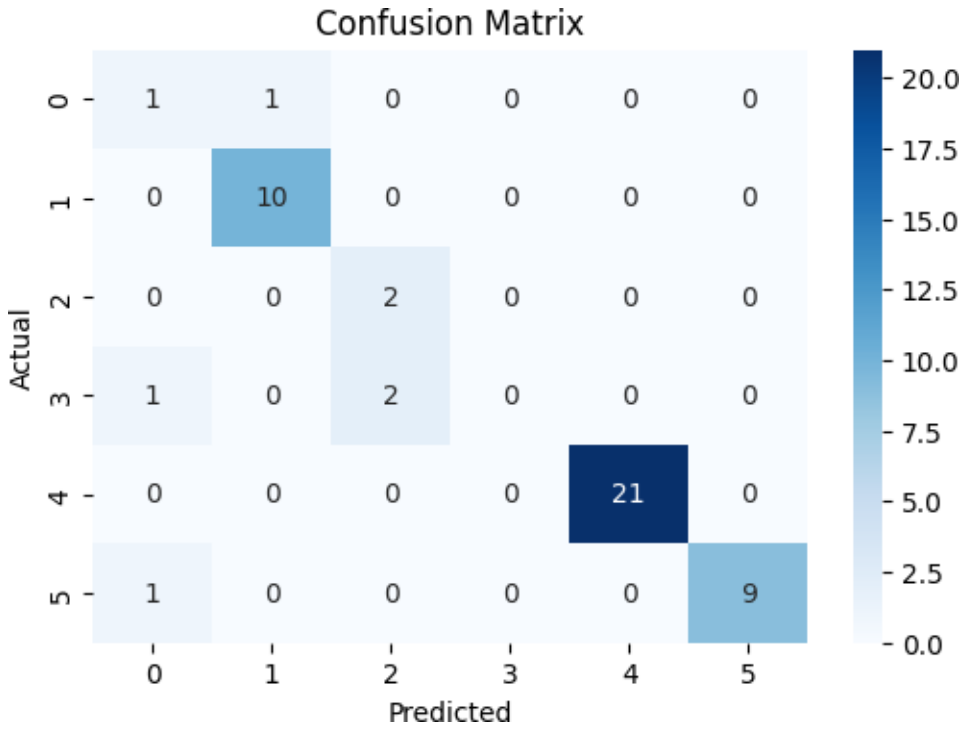
[101] :
```
# Assume X is your feature DataFrame, y_true is the true labels, and y_pred are
    your predictions.
misclassified_idx = np.where(y_pred != y_test)[0]
```

```
misclassified_data = X.iloc[misclassified_idx]
print("Number of misclassified instances:", len(misclassified_idx))
```

```
Number of misclassified instances: 5
```

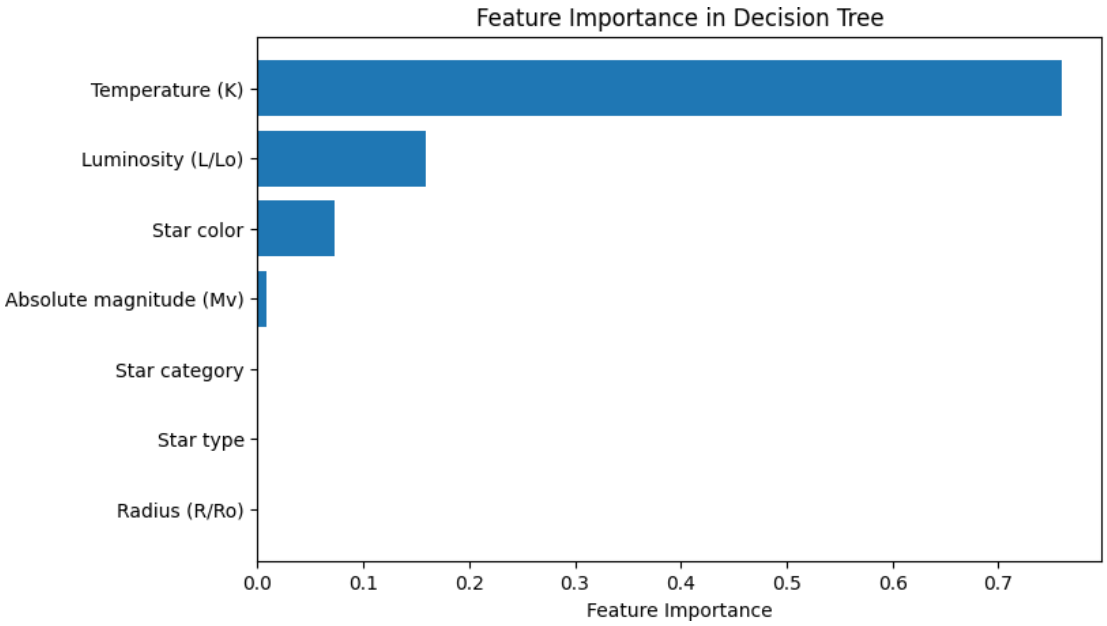
[102] :

```
# Confusion Matrix
plt.figure(figsize=(6, 4))
sns.heatmap(confusion_matrix(y_test, y_pred), annot=True, fmt='d', cmap="Blues")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.title("Confusion Matrix")
plt.show()
```



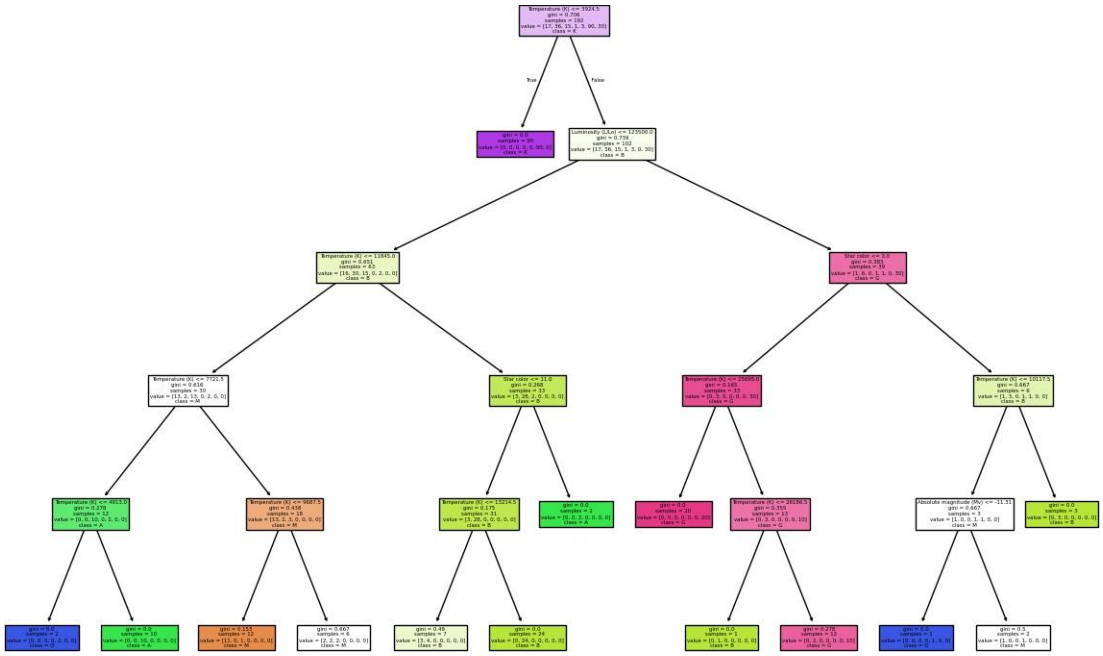
[103] :

```
# Feature Importance Analysis
feature_importance = dt.feature_importances_
sorted_idx = np.argsort(feature_importance)
plt.figure(figsize=(8,5))
plt.barh(range(len(sorted_idx)), feature_importance[sorted_idx], align='center')
plt.yticks(range(len(sorted_idx)), np.array(X.columns)[sorted_idx])
plt.xlabel("Feature Importance")
plt.title("Feature Importance in Decision Tree")
plt.show()
```



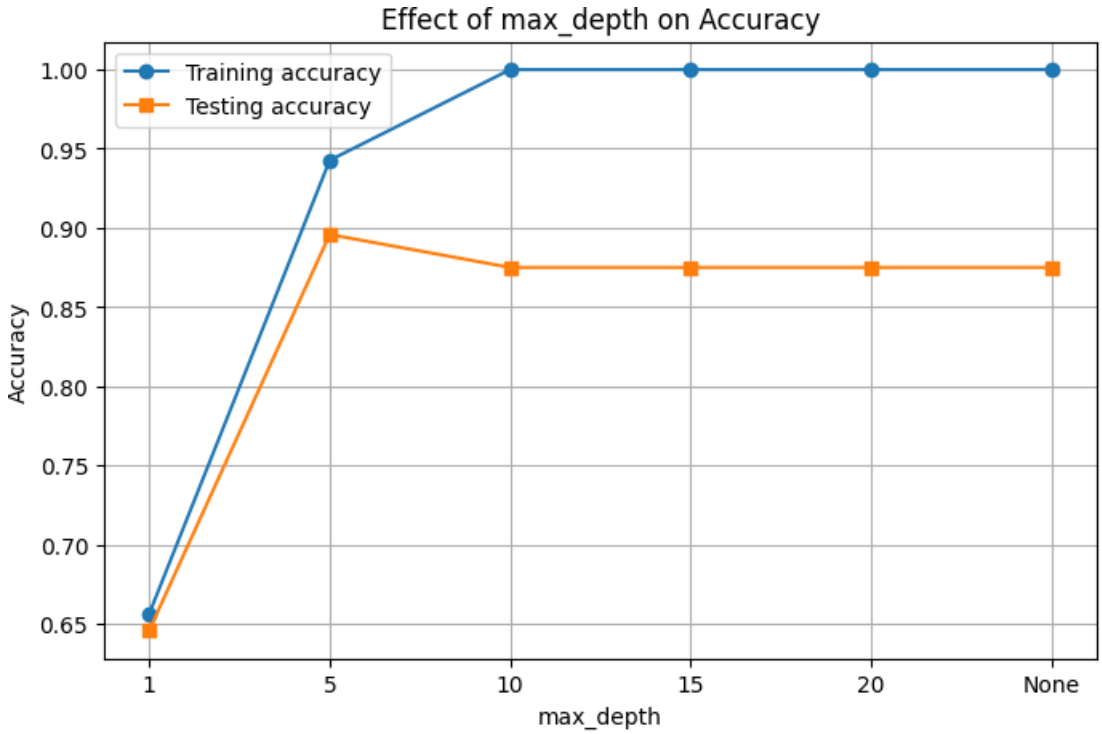
Temperature, luminosity and star colour are the most important features

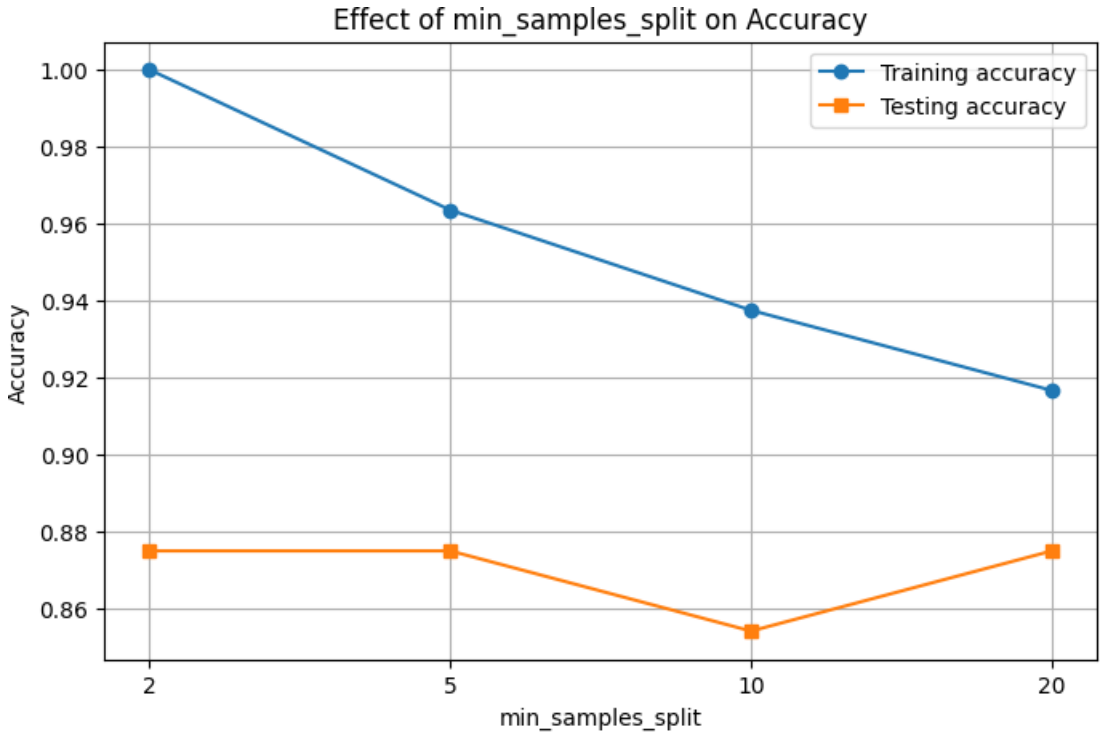
[106]:
```
# Visualize the Decision Tree
plt.figure(figsize=(15,10))
plot_tree(dt, feature_names=X.columns, class_names=original_class_names,␣
  ↪filled=True)
plt.show()
```

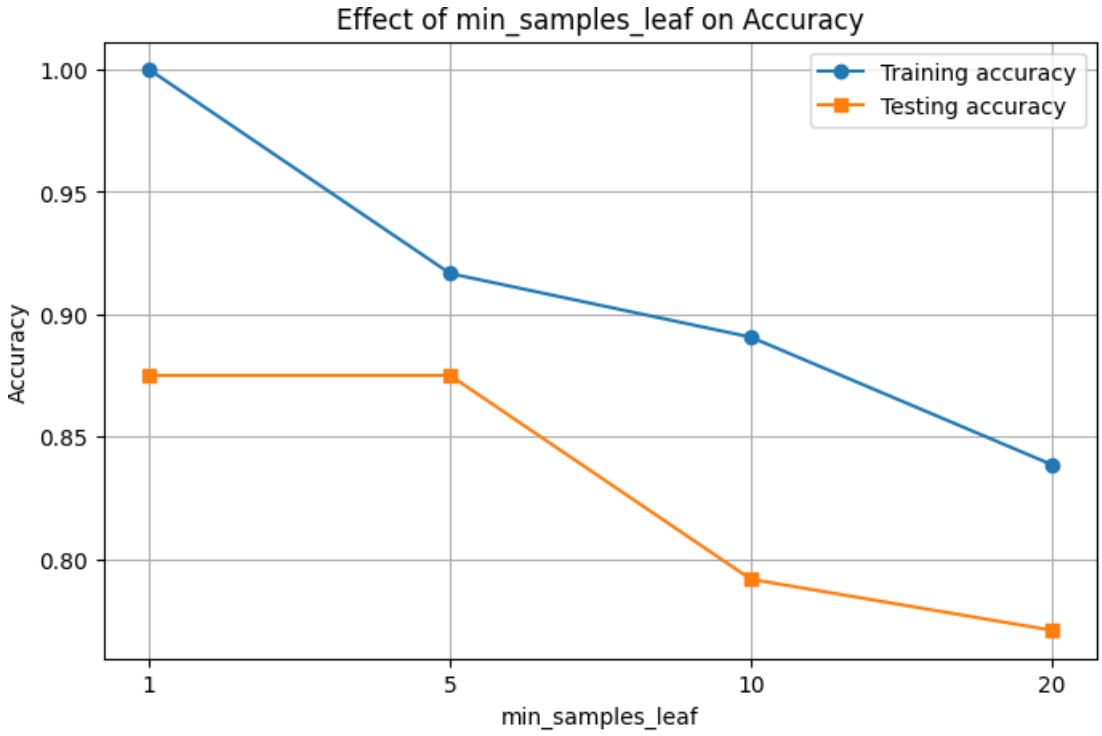


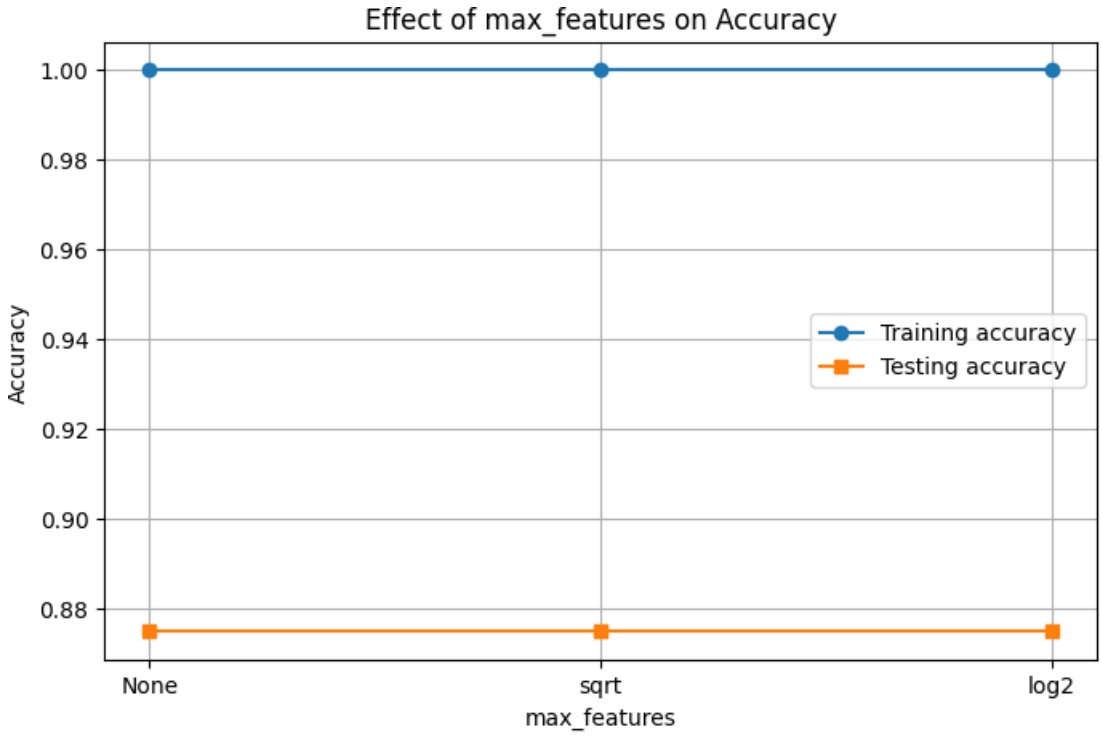
[107]:
```
# Parameters to iterate over
param_values = {
    "max_depth": [1, 5, 10, 15, 20, None],
    "min_samples_split": [2, 5, 10, 20],
    "min_samples_leaf": [1, 5, 10, 20],
    "max_features": [None, "sqrt", "log2"],
    "criterion": ["gini", "entropy"]
}

# Function to test parameters
def plot_parameter_effect(param_name, values):
    train_acc = []
    test_acc = []
    for value in values:
        model = DecisionTreeClassifier(random_state=42, **{param_name: value})
        model.fit(X_train, y_train)
        train_acc.append(model.score(X_train, y_train))
        test_acc.append(model.score(X_test, y_test))

    # Plot results
    plt.figure(figsize=(8, 5))
    # Convert None to string 'None' for plotting
    values_for_plot = [str(v) for v in values]
```

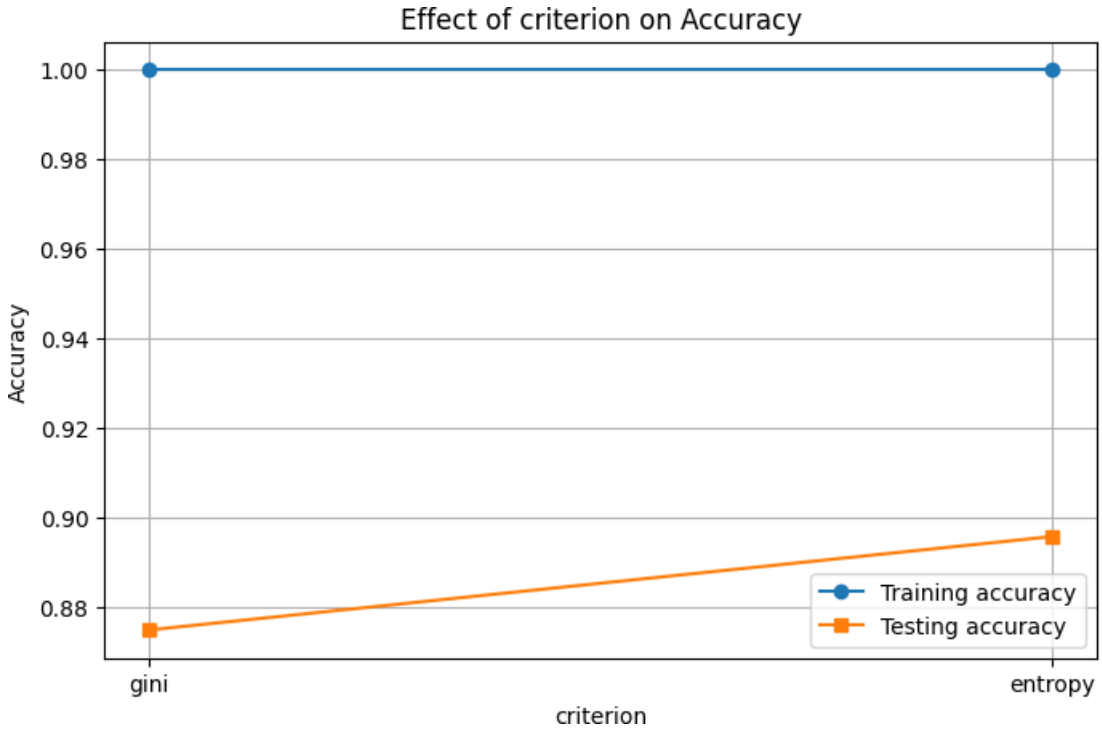
```
    plt.plot(values_for_plot, train_acc, marker='o',label="Training accuracy",
 linestyle='-')
    plt.plot(values_for_plot, test_acc, marker='s',label="Testing accuracy",
 linestyle='-')
    plt.xlabel(param_name)
    plt.ylabel("Accuracy")
    plt.legend()
    plt.title(f"Effect of {param_name} on Accuracy")
    plt.grid(True)
    plt.show()

# Iterate and plot for each parameter
for param, values in param_values.items():
    plot_parameter_effect(param, values)
```




Effect of min_samples_split on Accuracy
Training accuracy
Testing accuracy
Accuracy
1.00
0.98
0.96
0.94
0.92
0.90
0.88
0.86
2
5
10
20
min_samples_split


Effect of min_samples_leaf on Accuracy
Training accuracy
Testing accuracy
Accuracy
1.00
0.95
0.90
0.85
0.80
1
5
10
20
min_samples_leaf


Effect of max_features on Accuracy
1.00
0.98
0.96
0.94
0.92
0.90
0.88
Accuracy
Training accuracy
Testing accuracy
None
sqrt
log2
max_features


Effect of criterion on Accuracy
1.00
0.98
0.96
0.94
0.92
0.90
0.88
Accuracy
Training accuracy
Testing accuracy
gini
entropy
criterion

RESULT:

The following conclusions have been made:

Higher max_depth generally increases overfitting we can observe that too small of a value for this can cause underfitting.

Higher min_samples_split (If fewer than this number of samples are in a node, it won't split further.) reduces overfitting, but too high can lead to underfitting.

Lower min_samples_leaf (Prevents very small leaves, which might capture noise) leads to simpler tree, prevents overfitting.

Changing max_features(Controls how many features the model considers at each split.) Adds randomness and can improve generalization by limiting feature selection

Gini Impurity is preferred when speed is crucial (default in sklearn). Entropy is slightly more precise but takes more computation.

Gini= 1 - pi^2

Entropy= - pi log2 (pi)

Random forest

January 8th, 2025

AIM: PERFORM RANDOM FOREST ON A DATASET AND COMPARE ITS PERFORMANCE WITH DECISION TREES

DESCRIPTION: Random forest is an ensemble learning technique that combines the predictions of multiple decision trees to improve classification accuracy. It averages the predictions of many decision trees to reduce overfitting and variance leading to a more robust classifier.

PROPERTIES:

Ensemble Learning

Unlike a single decision tree, which is prone to overfitting, Random Forest generalizes better because: It averages multiple decision trees. Each tree is trained on a random subset of data. The randomness helps prevent the model from memorizing training data.

Uses Bootstrapping Each tree is trained on a bootstrapped sample of the dataset. Some data points may be selected multiple times, while others may be left out (Out-of-Bag samples).

Algorithm:

Step 1: Load necessary libraries and dataset

Step 2: Check for missing data

Step 3: Use label encoder on categorical columns

Step 4: Seperate the target column

Step 5: Divide dataset into training and testing sets

Step 6: Fit the random forst model

Step 7: Plot the confusion matrix and visualize the classified data

Step 8: Plot one of the decision trees

Step 9: Iterate over various parameters and observe the training and testing accuracy

[1]:
```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, confusion_matrix,
    classification_report
```

[4]:
```
data = pd.read_csv("https://github.com/YBIFoundation/Dataset/raw/main/Stars.csv")
```

[5]:
```
data.head()
```

[5]:

	Temperature (K)	Luminosity (L/Lo)	Radius (R/Ro)	Absolute magnitude (Mv)	Star type	Star category	Star color	Spectral Class
0	3068	0.002400	0.1700	16.12	0	Brown Dwarf	Red	M
1	3042	0.000500	0.1542	16.60	0	Brown Dwarf	Red	M
2	2600	0.000300	0.1020	18.70	0	Brown Dwarf	Red	M
3	2800	0.000200	0.1600	16.65	0	Brown Dwarf	Red	M
4	1939	0.000138	0.1030	20.06	0	Brown Dwarf	Red	M

[6]:
```
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
# Loop through columns and apply LabelEncoder to categorical columns
for column in data.columns:
    if data[column].dtype == 'object':    # Identify categorical columns
        data[column] = le.fit_transform(data[column])
```

[8]:
```
X = data.drop("Spectral Class", axis=1)  # Features
y = data["Spectral Class"]  # Target
```

[9]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

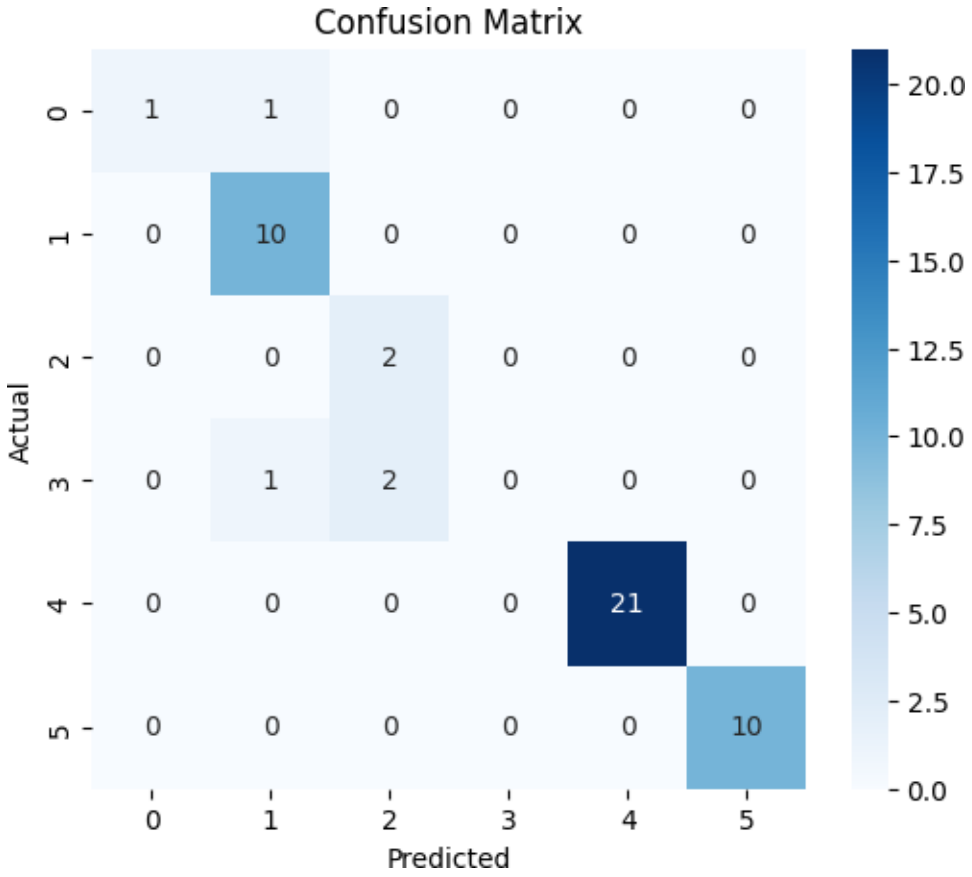
[18]:
```
rf_model = RandomForestClassifier(n_estimators=100, random_state=42)

# Fit the model to the training data
rf_model.fit(X_train, y_train)

# Make predictions
y_pred = rf_model.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)
class_report = classification_report(y_test, y_pred)

# Output the results
print(f'Accuracy: {accuracy}')
```

```
print(f'Classification Report:\n{class_report}')

# Import necessary libraries
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix

# Generate confusion matrix
conf_matrix = confusion_matrix(y_test, y_pred)

# Plot the confusion matrix using seaborn
plt.figure(figsize=(6, 5))
sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Blues")
plt.title("Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()


# Feature Importance (optional)
feature_importance = rf_model.feature_importances_
print("Feature Importance:", feature_importance)
```

Accuracy: 0.9166666666666666
Classification Report:

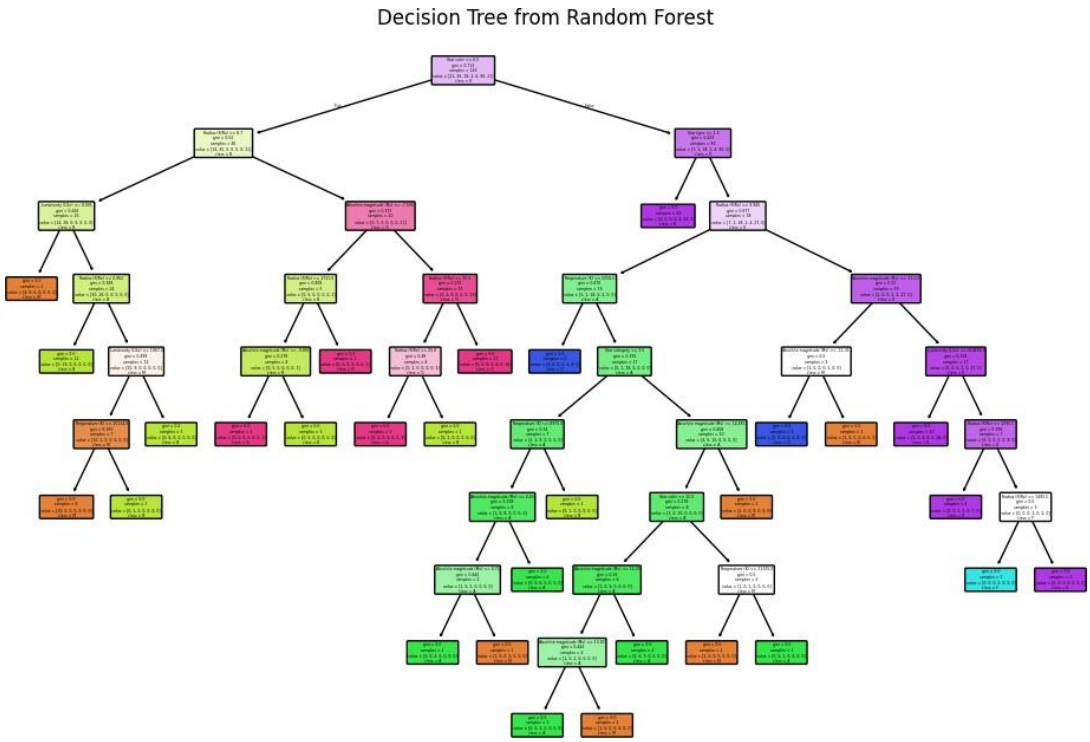
	precision	recall	f1-score	support
0	1.00	0.50	0.67	2
1	0.83	1.00	0.91	10
2	0.50	1.00	0.67	2
4	0.00	0.00	0.00	3
5	1.00	1.00	1.00	21

6	1.00	1.00	1.00	10
accuracy			0.92	48
macro avg	0.72	0.75	0.71	48
weighted avg	0.88	0.92	0.89	48



Feature Importance: [0.32255171 0.12167076 0.09407667 0.11571012 0.08800245 0.03993386
0.21805443]

[17]:
```
from sklearn.tree import plot_tree

# Plot one of the decision trees in the Random Forest
plt.figure(figsize=(12, 8))
plot_tree(rf_model.estimators_[0], filled=True, feature_names=X.columns,
          class_names=['M', 'B', 'A', 'F', 'O', 'K', 'G'], rounded=True)
plt.title("Decision Tree from Random Forest")
plt.show()
```



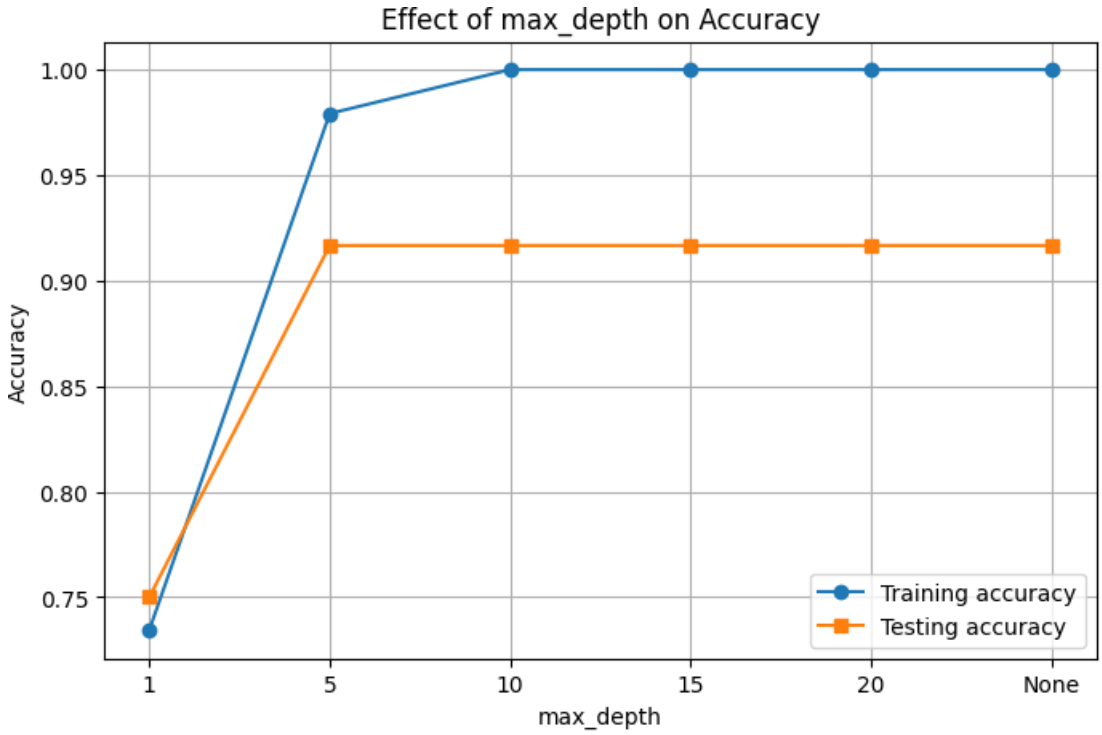
We get the same features as most important features as decision tree

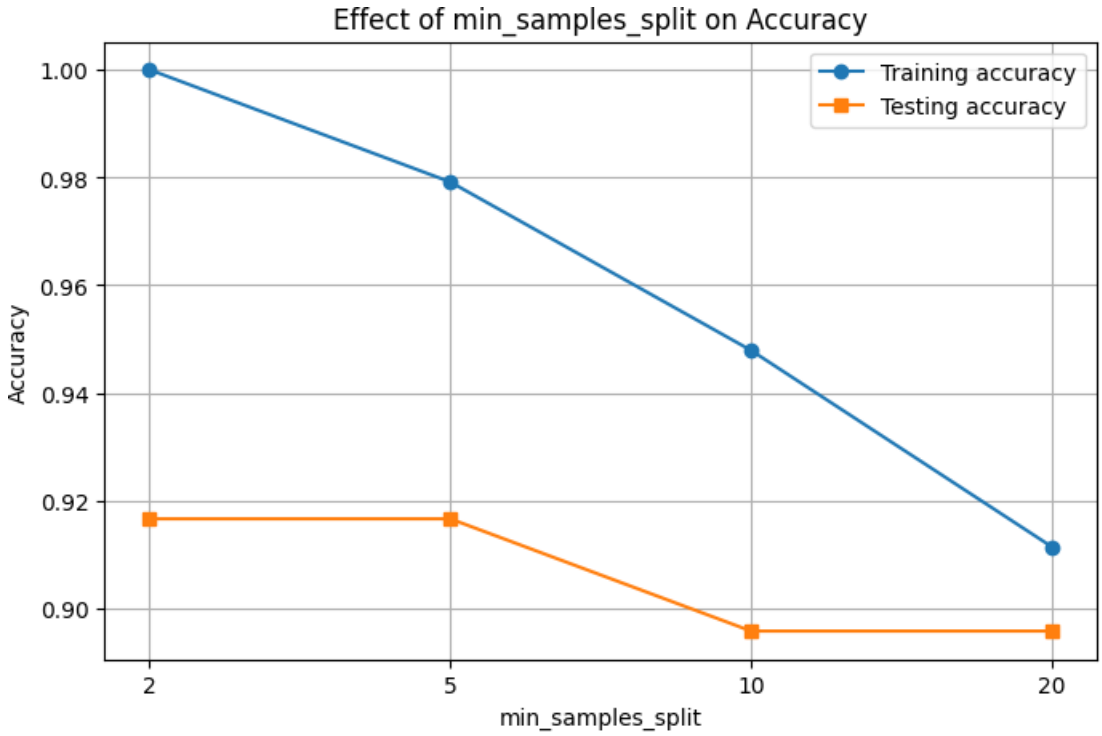
We get better accuracy, precision, f1 score, recall than decision tree

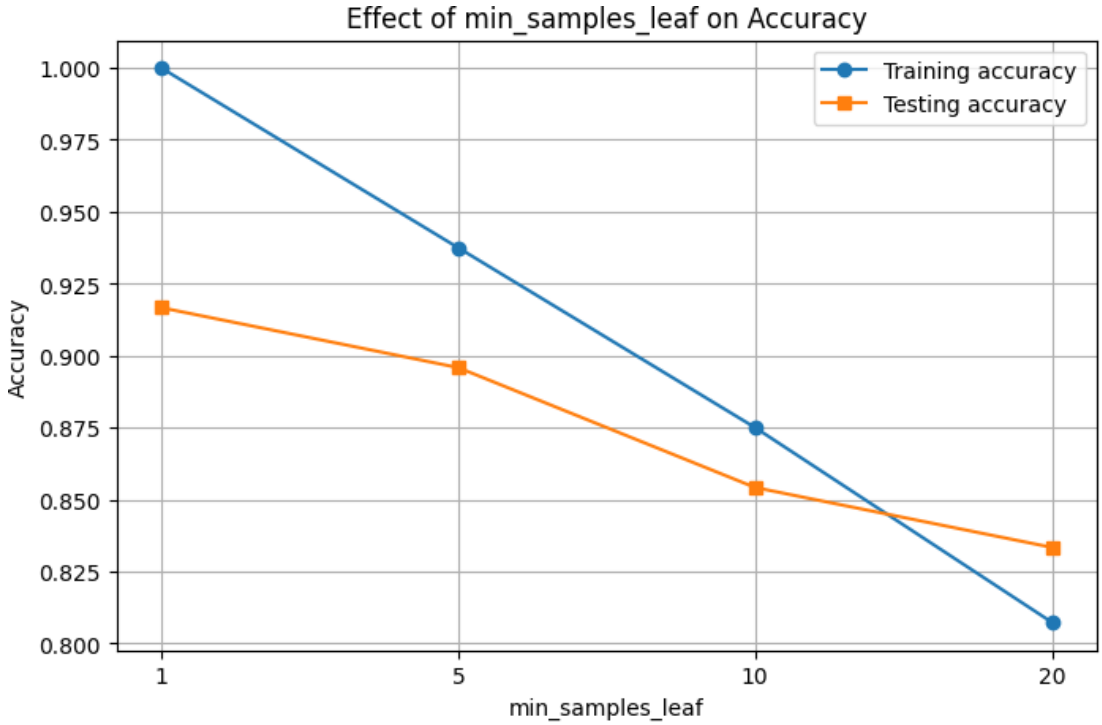
While other hyperparameters are same as decision tree, the hyperparameter n_estimators (number of trees) increases model stability

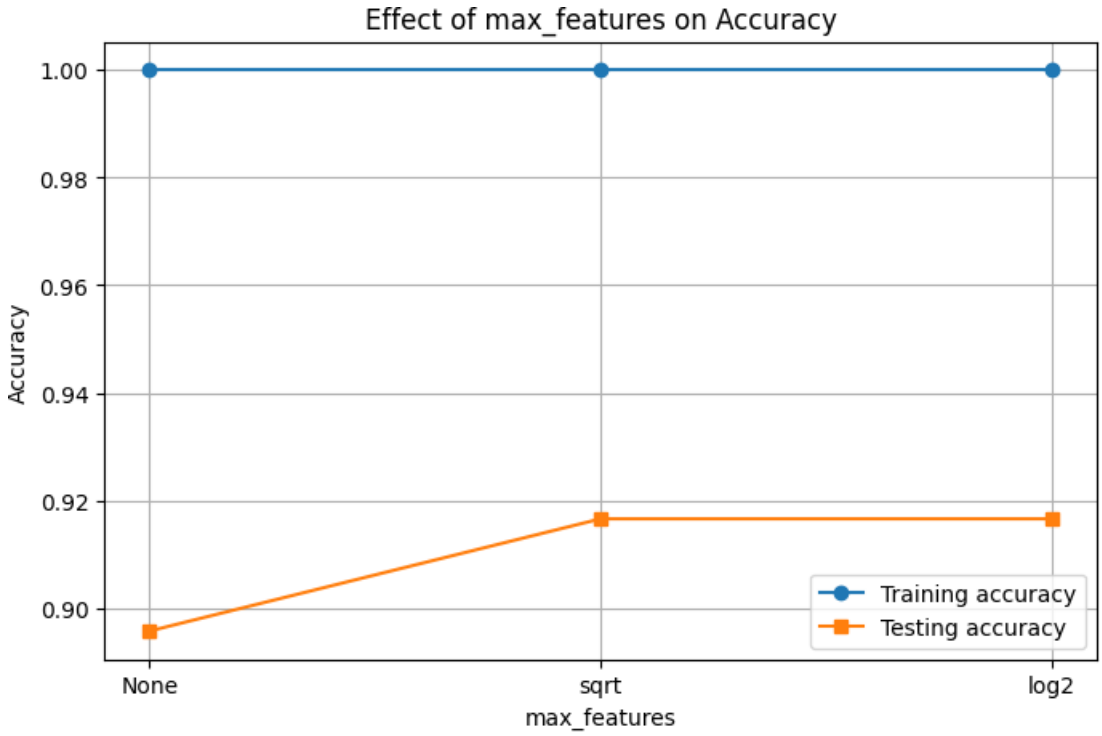
[21]:
```
# Parameters to iterate over
param_values = {
    "max_depth": [1, 5, 10, 15, 20, None],
    "min_samples_split": [2, 5, 10, 20],
    "min_samples_leaf": [1, 5, 10, 20],
    "max_features": [None, "sqrt", "log2"],
    "criterion": ["gini", "entropy"],
    "n_estimators": [25,50,75,100,125,150]
}

# Function to test parameters
def plot_parameter_effect(param_name, values):
    train_acc = []
    test_acc = []
    for value in values:
        model = RandomForestClassifier(random_state=42, **{param_name: value})
        model.fit(X_train, y_train)
```

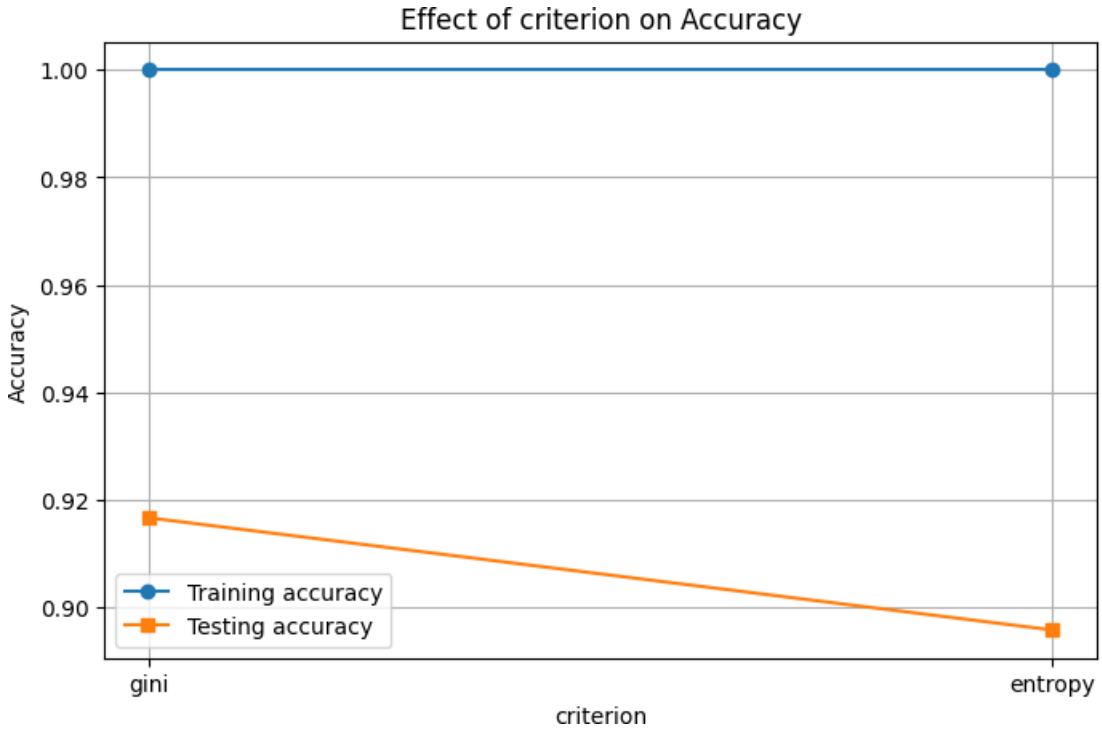
```
            train_acc.append(model.score(X_train, y_train))
            test_acc.append(model.score(X_test, y_test))

        # Plot results
        plt.figure(figsize=(8, 5))

        values_for_plot = [str(v) for v in values]
        plt.plot(values_for_plot, train_acc, marker='o',label="Training accuracy",
    linestyle='-')
        plt.plot(values_for_plot, test_acc, marker='s',label="Testing accuracy",
    linestyle='-')
        plt.xlabel(param_name)
        plt.ylabel("Accuracy")
        plt.legend()
        plt.title(f"Effect of {param_name} on Accuracy")
        plt.grid(True)
        plt.show()

# Iterate and plot for each parameter
for param, values in param_values.items():
    plot_parameter_effect(param, values)
```

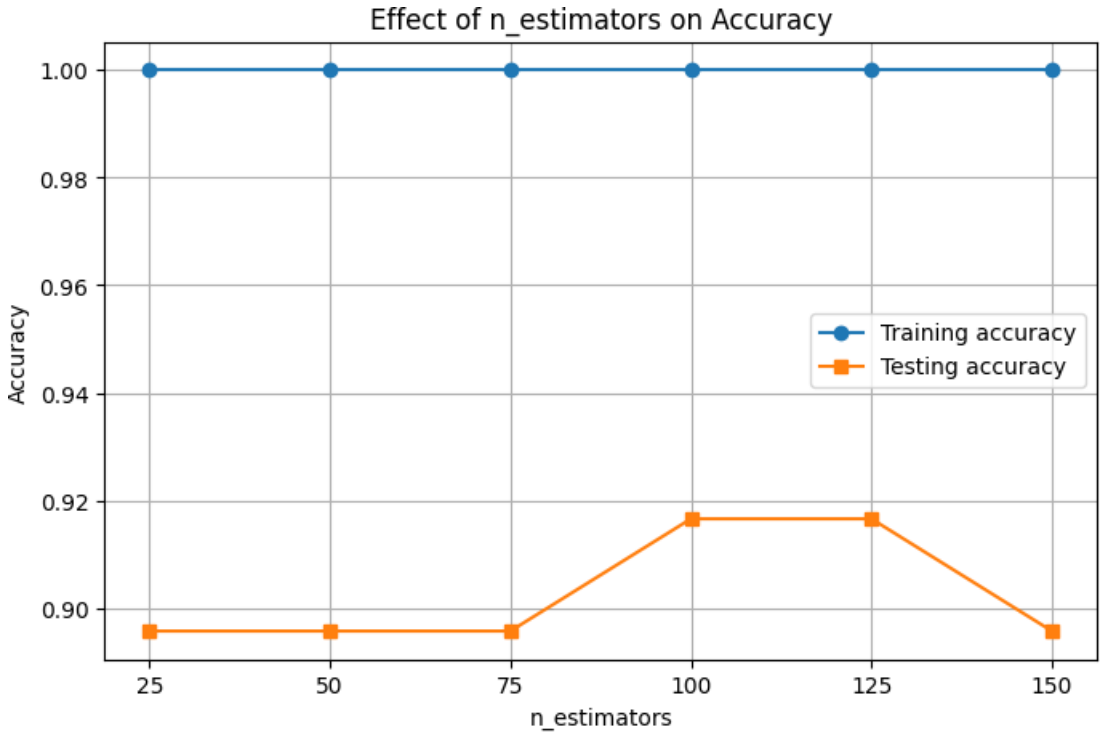



Effect of min_samples_split on Accuracy
Training accuracy
Testing accuracy
Accuracy
1.00
0.98
0.96
0.94
0.92
0.90
2
5
10
20
min_samples_split


Effect of min_samples_leaf on Accuracy
Training accuracy
Testing accuracy
Accuracy
1.000
0.975
0.950
0.925
0.900
0.875
0.850
0.825
0.800
1
5
10
20
min_samples_leaf


Effect of max_features on Accuracy
1.00
0.98
0.96
0.94
0.92
0.90
Accuracy
Training accuracy
Testing accuracy
None
sqrt
log2
max_features


Effect of criterion on Accuracy
1.00
0.98
0.96
0.94
0.92
0.90
Accuracy
Training accuracy
Testing accuracy
gini
entropy
criterion



RESULT:

We get the same features as most important features as decision tree We get better accuracy, precision, f1 score, recall than decision tree
While other hyperparameters are same as decision tree, the hyperparameter n_estimators (number of trees) increases model stability

Naïve Bayes

January 1st, 2025

AIM: PERFORM NAÏVE BAYES ON A DATASET AND ANALYSE ITS PERFORMANCE

Description:

Naive Bayes is a classification technique that is based on Bayes' Theorem with an assumption that all the features that predicts the target value are independent of each other. It calculates the probability of each class and then pick the one with the highest probability.

Naive Bayes classifier assumes that the features we use to predict the target are independent and do not affect each other. While in real-life data, features depend on each other in determining the target, but this is ignored by the Naive Bayes classifier.

Though the independence assumption is never correct in real-world data, but often works well in practice. so that it is called "Naive"

Requires a small amount of training data. So the training takes less time.

Handles continuous and discrete data, and it is not sensitive to irrelevant features.

Very simple, fast, and easy to implement.

Can be used for both binary and multi-class classification problems.

Highly scalable as it scales linearly with the number of predictor features and data points.

When the Naive Bayes conditional independence assumption holds true, it will converge quicker than discriminative models like logistic regression.

Algorithm:

Step 1: Load data, libraries

Step 2: Split data to train and test

Step 3: Train model

Step 4: Test model

Step 5: Evaluate accuracy

Step 6: Visualize confusion matrix

Step 7: Derive insights

[2]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

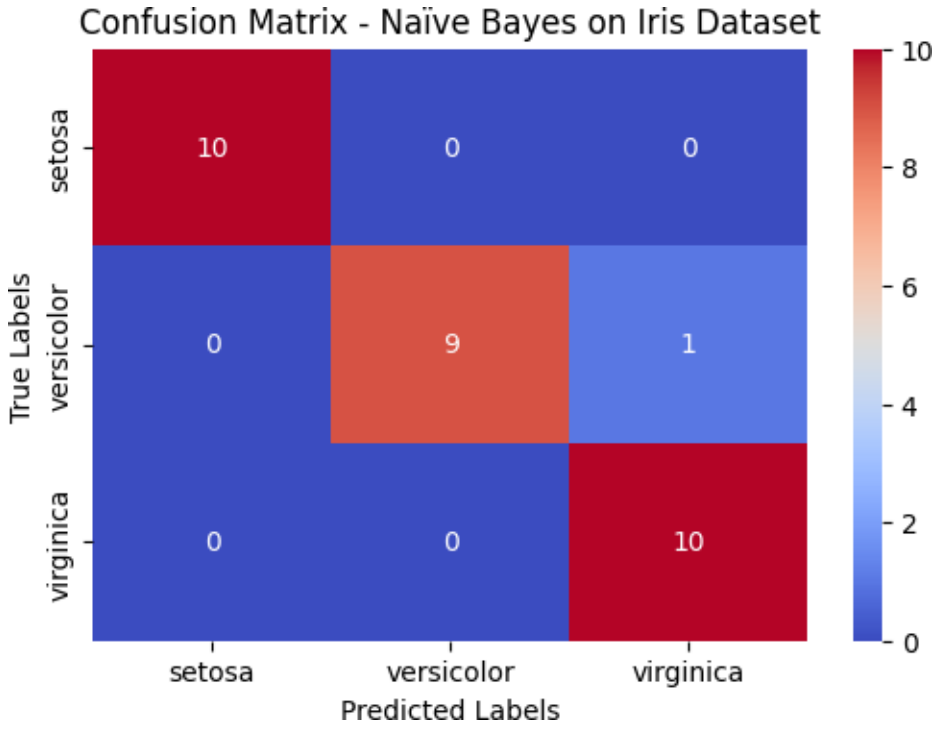
```
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.datasets import load_iris
from sklearn.metrics import accuracy_score, confusion_matrix,
    classification_report

# Load the Iris dataset
data_iris = load_iris()
X_iris, y_iris = data_iris.data, data_iris.target
feature_names = data_iris.feature_names
class_names = data_iris.target_names

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X_iris, y_iris, test_size=0.
    2, random_state=42, stratify=y_iris)

# Train Naïve Bayes model
nb_model = GaussianNB()
nb_model.fit(X_train, y_train)

# Predictions
y_pred = nb_model.predict(X_test)

# Evaluation
accuracy = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)
class_report = classification_report(y_test, y_pred, target_names=class_names)

print(" Naïve Bayes on Iris Dataset:")
print(f"Accuracy: {accuracy:.4f}")
print("\nConfusion Matrix:\n", conf_matrix)
print("\nClassification Report:\n", class_report)

# Visualizing Confusion Matrix
plt.figure(figsize=(6,4))
sns.heatmap(conf_matrix, annot=True, cmap='coolwarm', fmt='d',
    xticklabels=class_names, yticklabels=class_names)
plt.xlabel("Predicted Labels")
plt.ylabel("True Labels")
plt.title("Confusion Matrix - Naïve Bayes on Iris Dataset")
plt.show()
```

```
 Naïve Bayes on Iris Dataset:
Accuracy: 0.9667

Confusion Matrix:
```

```
[[10  0  0]
 [ 0  9  1]
 [ 0  0 10]]
```

Classification Report:

	precision	recall	f1-score	support
setosa	1.00	1.00	1.00	10
versicolor	1.00	0.90	0.95	10
virginica	0.91	1.00	0.95	10
accuracy			0.97	30
macro avg	0.97	0.97	0.97	30
weighted avg	0.97	0.97	0.97	30



Result:

The accuracy is high due to the simplicity and size of dataset

Insights and Potential Issues:

Excellent performance on setosa: The model perfectly classifies setosa flowers. Some confusion between versicolor and virginica: The model struggles slightly to distinguish between versicolor and virginica, as evidenced by the one misclassification. This is a common observation with the Iris dataset, as these two species have some overlapping features. Overall good performance: Despite the minor confusion, the model demonstrates high accuracy and performs well overall for this classification task.

Linear regression

January 29th, 2025

AIM: PERFORM LINEAR REGRESSION ON TWO DATASETS AND ANALYSE ITS PERFORMANCE

DESCRIPTION: Linear Regression is a supervised learning algorithm used for predicting a continuous target variable based on one or more input features.

Objective: Find the best-fit line that minimizes the difference between actual and predicted values using Mean Squared Error (MSE).

Algorithm:

Step 1: Import necessary libraries and data

Step 2: Encode Categorical data

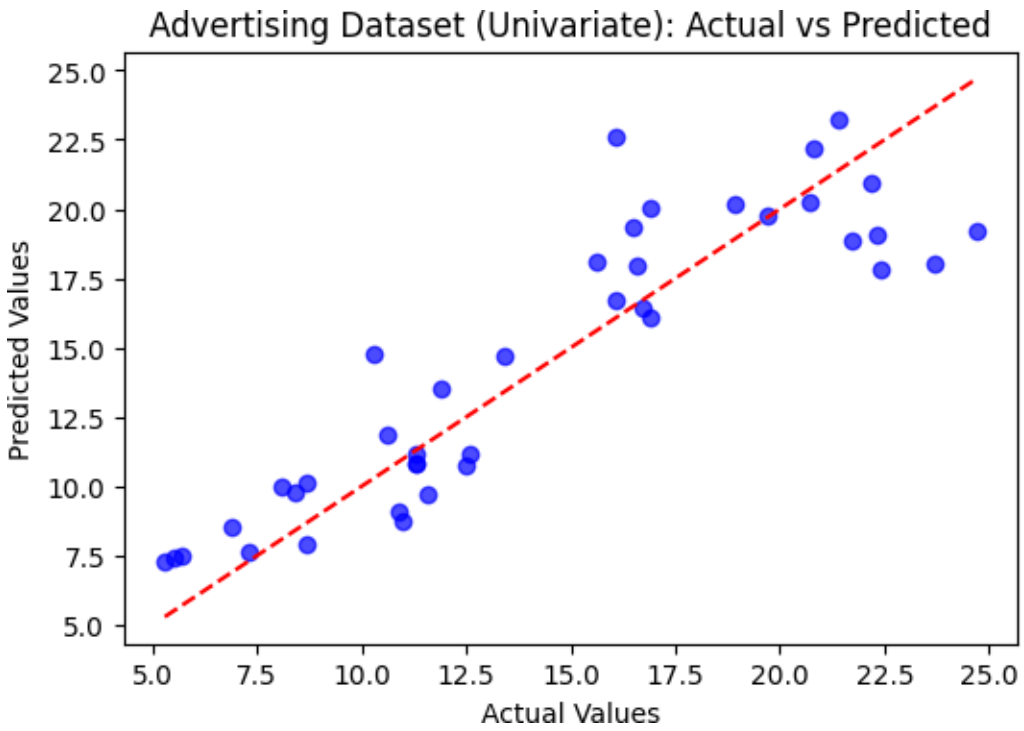
Step 3: Split dataset into train and test sets

Step 4: Train and evalauate the model

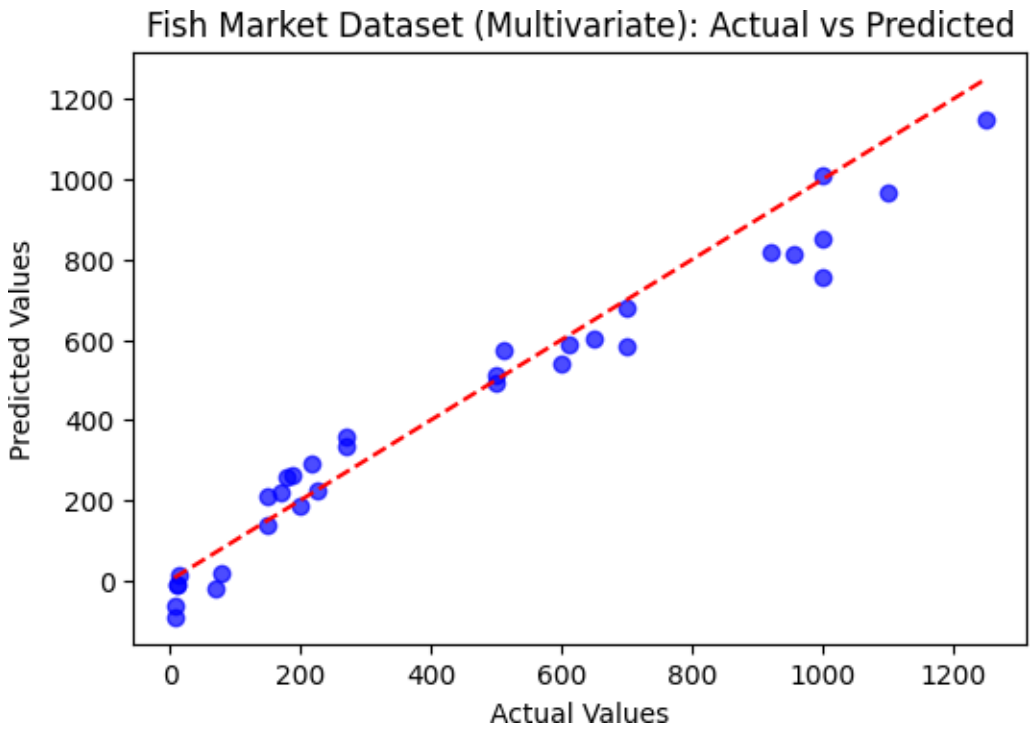
[3]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score

# Load Advertising Dataset (Univariate)
df_ad = pd.read_csv("/content/advertising.csv")      # Contains 'TV', 'Radio',␣
  ↪'Newspaper', 'Sales'
df_ad = df_ad[['TV', 'Sales']]   # Use only TV advertising spend
df_ad.columns = ['TV_Spend', 'Sales']

# Load Fish Market Dataset (Multivariate)
url_fish = "/content/Fish[1].csv"
df_fish = pd.read_csv(url_fish)  # Contains 'Species', 'Weight', 'Length1',␣
  ↪'Length2', 'Length3', 'Height', 'Width'

# Encode categorical feature (Species)
df_fish = pd.get_dummies(df_fish, columns=['Species'], drop_first=True)
```

```python
# Function to train and evaluate a model
def train_and_evaluate(df, features, target, dataset_name):
    # Split dataset
    X = df[features]
    y = df[target]
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ▫random_state=42)

    # Train Model
    model = LinearRegression()
    model.fit(X_train, y_train)

    # Predictions
    y_pred = model.predict(X_test)

    # Metrics
    mse = mean_squared_error(y_test, y_pred)
    mae = mean_absolute_error(y_test, y_pred)
    r2 = r2_score(y_test, y_pred)

    # Print results
    print(f"\nResults for {dataset_name}:")
    print(f"MSE: {mse:.4f}, MAE: {mae:.4f}, R² Score: {r2:.4f}")

    # Visualization (Actual vs Predicted)
    plt.figure(figsize=(6, 4))
    plt.scatter(y_test, y_pred, alpha=0.7, color="blue")
    plt.plot([min(y_test), max(y_test)], [min(y_test), max(y_test)],␣
 ▫linestyle='dashed', color="red")
    plt.xlabel("Actual Values")
    plt.ylabel("Predicted Values")
    plt.title(f"{dataset_name}: Actual vs Predicted")
    plt.show()

# Train on Univariate Dataset (Advertising)
train_and_evaluate(df_ad, ['TV_Spend'], 'Sales', "Advertising Dataset␣
 ▫(Univariate)")

# Train on Multivariate Dataset (Fish Market)
features_fish = df_fish.columns.drop('Weight')  # Exclude target
train_and_evaluate(df_fish, features_fish, 'Weight', "Fish Market Dataset␣
 ▫(Multivariate)")
```

Results for Advertising Dataset (Univariate):
MSE: 6.1011, MAE: 1.9503, R² Score: 0.8026



Results for Fish Market Dataset (Multivariate):
MSE: 7007.3832, MAE: 65.3001, R^2 Score: 0.9507



:

Result:

The model can perform well for both univariate and multivariate data as long as there is a linear relationship.

K-means clustering with outlier detection(using LOF)

January 29th, 2025

AIM: PERFORM K MEANS CLUSTERING ON A DATASET AND ANALYSE ITS PERFORMANCE. ALSO PERFORM OUTLIER DETECTION

DESCRIPTION:

K-Means is an unsupervised learning algorithm used for clustering data into k groups. It minimizes the variance within each cluster by iteratively updating cluster centroids.

We first choose k and assign nearest point to centroid using Euclidian distance. Then we recalculate centroids of the clusters based on current assignments. We keep repeating it untill the centroids no longer change or thet change is below a threshold.

The objective of k-means is to minimize the WCSS: within-cluster sum of square distances

LOF is a density-based anomaly detection algorithm that compares the local density of a point to that of its neighbors. Anomalies have significantly lower densities than surrounding points

For each point, it finds k nearest neighbors and calculates the reachability distance from point A to pint B

reach-distk(A,B)=max{k-distance(B),distance(A,B)}

then the local reachability distance and LOF score is calculated

LOF close to 1 means point is similar to neighbors

ALGORITHM:

Step 1: Load necessary libraries and dataset

Step 2: Scale data

Step 3: Perform outlier detection using LOF

Step 4: Perform clustering using k-means

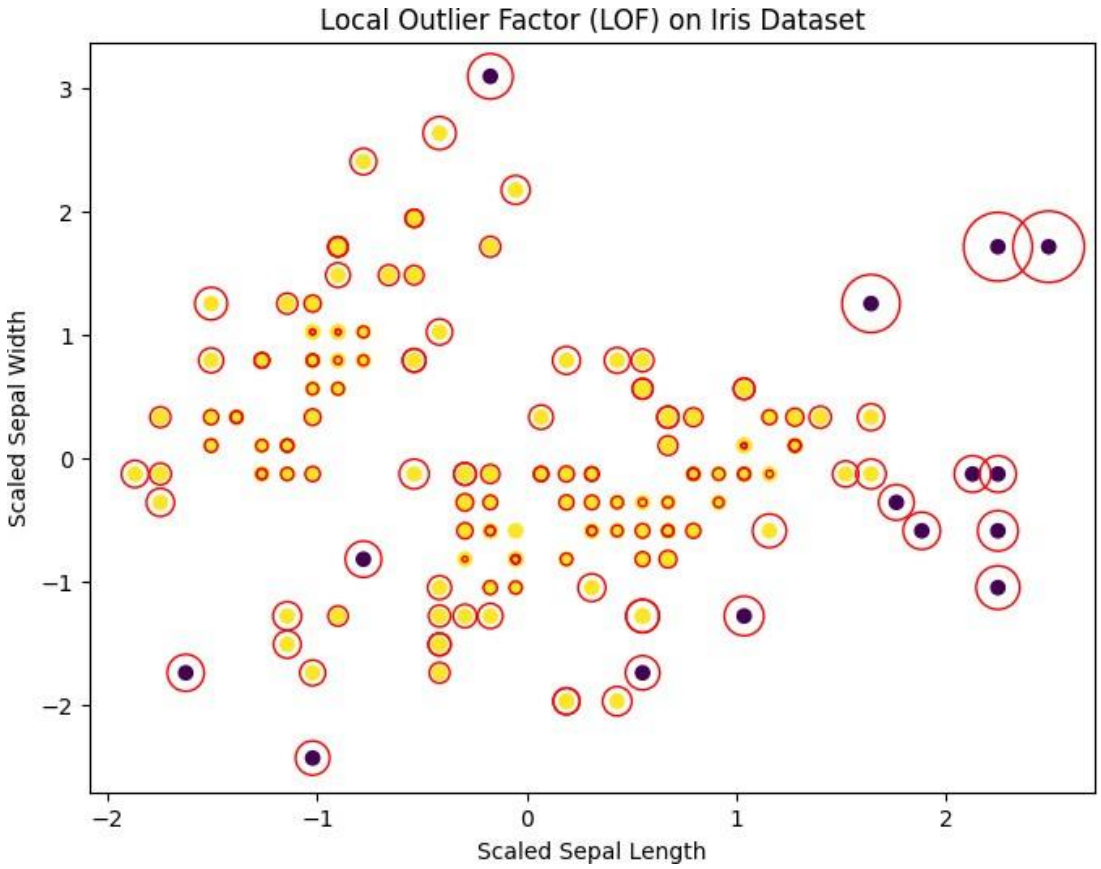
Step 5: Use silhouette score to evaluate clustering

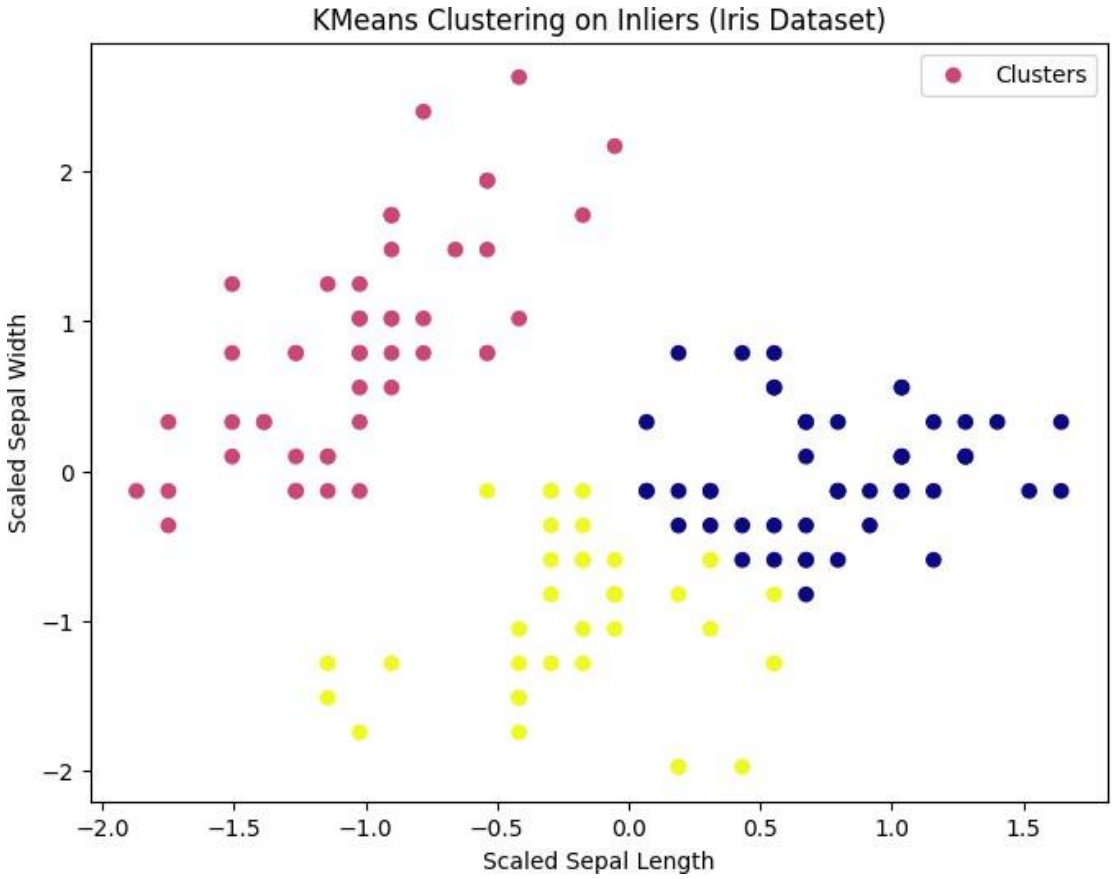
Step 6: Plot the clusters for better visualization

```python
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans
from sklearn.neighbors import LocalOutlierFactor
from sklearn.metrics import silhouette_score
import matplotlib.pyplot as plt
from matplotlib.legend_handler import HandlerPathCollection
from sklearn.datasets import load_iris  # Or any other sklearn dataset

# Load an sklearn dataset (e.g., iris)
iris = load_iris()
df = pd.DataFrame(data=iris.data, columns=iris.feature_names)
X = df[['sepal length (cm)', 'sepal width (cm)']].copy()  # Select two features
  for 2D visualization

# Scale the data (important for both LOF and KMeans)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# 1. Outlier Detection using LOF
lof = LocalOutlierFactor(n_neighbors=10, contamination=0.1)  # Adjust as needed
outlier_labels = lof.fit_predict(X_scaled)
X_scores = lof.negative_outlier_factor_

X_inliers = X_scaled[outlier_labels == 1]
X_outliers = X_scaled[outlier_labels == -1]
```

```python
# 2. Clustering with KMeans (on the inliers only)
kmeans = KMeans(n_clusters=3, random_state=42)  # Choose the number of clusters
kmeans.fit(X_inliers)
cluster_labels = kmeans.labels_

# Assign cluster labels back to the original DataFrame (inliers)
X_clustered = X[outlier_labels == 1].copy()
X_clustered['cluster'] = cluster_labels

# Evaluate clustering performance (e.g., silhouette score)
if len(set(cluster_labels)) > 1:    # Check if more than one cluster is present
    silhouette_avg = silhouette_score(X_inliers, cluster_labels)
    print(f"Silhouette Score: {silhouette_avg}")
else:
    print("Silhouette Score cannot be calculated with only one cluster.")

# 3. Visualization

# Outlier Visualization
plt.figure(figsize=(8, 6))
plt.scatter(X_scaled[:, 0], X_scaled[:, 1], c=outlier_labels, cmap='viridis',
    label='Data points')
radius = (X_scores.max() - X_scores) / (X_scores.max() - X_scores.min())
scatter = plt.scatter(X_scaled[:, 0], X_scaled[:, 1], s=1000 * radius,
    edgecolors='r', facecolors='none', label='Outlier scores')
plt.title('Local Outlier Factor (LOF) on Iris Dataset')
plt.xlabel('Scaled Sepal Length')
plt.ylabel('Scaled Sepal Width')
plt.show()

# Cluster Visualization (on inliers)
plt.figure(figsize=(8, 6))
plt.scatter(X_inliers[:, 0], X_inliers[:, 1], c=cluster_labels, cmap='plasma',
    label='Clusters')
plt.title('KMeans Clustering on Inliers (Iris Dataset)')
plt.xlabel('Scaled Sepal Length')
plt.ylabel('Scaled Sepal Width')
plt.legend()
plt.show()

# (Optional) Analyze the clusters
print("\nCluster Analysis:")
print(X_clustered.groupby('cluster').mean())
```

```
Silhouette Score: 0.4526407325876225
```


Local Outlier Factor (LOF) on Iris Dataset
Scaled Sepal Width
Scaled Sepal Length
3
2
1
0
−1
−2
−2
−1
0
1
2



Cluster Analysis:

cluster	sepal length (cm)	sepal width (cm)
0	6.489796	3.053061
1	5.002083	3.431250
2	5.705263	2.626316

Result: The LOF analysis helped remove potential outliers that could have negatively impacted the clus- tering results. KMeans was able to identify three distinct clusters that likely correspond to the three species of Iris flowers. The cluster analysis table provides insights into the characteristics of each cluster, showing how the sepal length and width vary across the groups.

Gaussian Mixture Models

February 5th, 2025

AIM: Perform Gaussian Mixture Model on Two Different Datasets and Analyse its Performance

Description:

Gaussian Mixture Models (GMMs) are powerful probabilistic models that represent the underlying distribution of data as a mixture of multiple Gaussian (normal) distributions. Each component in the mixture corresponds to a cluster, and the overall model can capture complex, multi-modal data distributions.

Unlike traditional clustering methods like K-Means, which assume spherical clusters and use Euclidean distance to assign points to the nearest cluster center, GMMs allow for elliptically shaped clusters and provide soft clustering—assigning probabilities that a point belongs to each cluster.

Algorithm:

Step 1: Import necessary libraries

Step 2: Fit and predict first dataset to kmeans and gmm

Step 3: Use PCA for dimensionality reduction

Step 4: Visualize clusters using a plot

Step 5: Repeat for a different dataset

Step 6: Evaluate performance based on various metrrics like silouhette score, homogenity etc

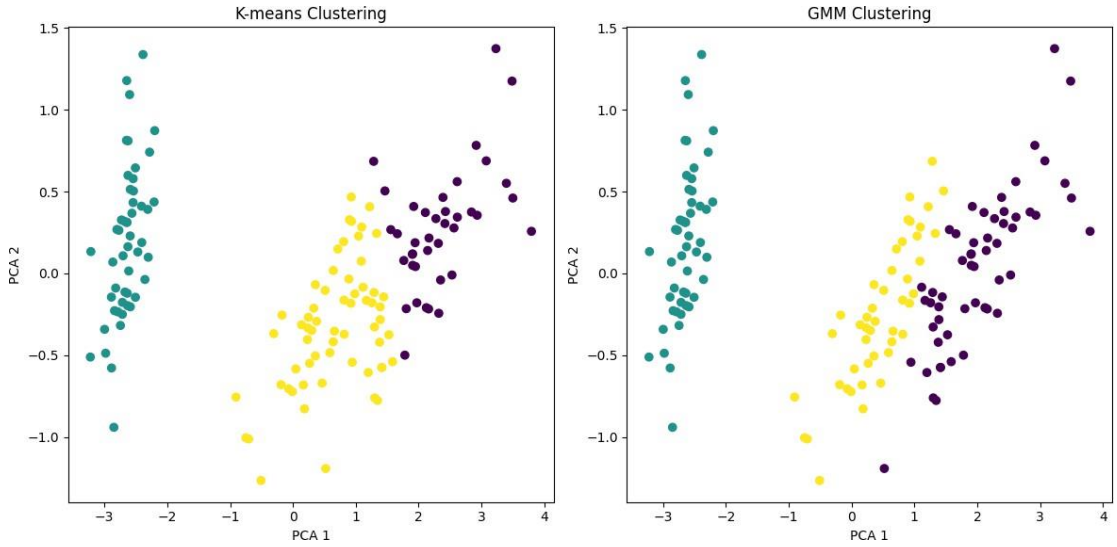
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.cluster import KMeans
from sklearn.mixture import GaussianMixture
from sklearn.decomposition import PCA

# Load the iris dataset
iris = load_iris()
X = iris.data
```

```
y = iris.target
```

[ ]:
```
# Apply K-means clustering with 3 clusters (since there are 3 species)
kmeans = KMeans(n_clusters=3, random_state=42)
kmeans_labels = kmeans.fit_predict(X)
```

[ ]:
```
# Apply GMM clustering with 3 components (again, since there are 3 species)
gmm = GaussianMixture(n_components=3, random_state=42)
gmm_labels = gmm.fit_predict(X)
```

[ ]:
```
# Reduce the data to 2D using PCA for visualization purposes
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)

# Visualize K-means clustering result
plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=kmeans_labels, cmap='viridis')
plt.title("K-means Clustering")
plt.xlabel("PCA 1")
plt.ylabel("PCA 2")

# Visualize GMM clustering result
plt.subplot(1, 2, 2)
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=gmm_labels, cmap='viridis')
plt.title("GMM Clustering")
plt.xlabel("PCA 1")
plt.ylabel("PCA 2")

plt.tight_layout()
plt.show()
```




```
[ ]: from sklearn.metrics import adjusted_rand_score, homogeneity_score

# Compare the clustering results with the true labels
print("Adjusted Rand Index (K-means):", adjusted_rand_score(y, kmeans_labels))
print("Adjusted Rand Index (GMM):", adjusted_rand_score(y, gmm_labels))

print("Homogeneity (K-means):", homogeneity_score(y, kmeans_labels))
print("Homogeneity (GMM):", homogeneity_score(y, gmm_labels))
```

```
Adjusted Rand Index (K-means): 0.7163421126838476
Adjusted Rand Index (GMM): 0.9038742317748124
Homogeneity (K-means): 0.7364192881252849
Homogeneity (GMM): 0.8983263672602775
```

Since the Iris dataset contains clusters that are somewhat linearly separable but may also have some overlap, GMM may better capture the subtleties of the cluster structure due to its probabilistic nature. K-means may perform well here too, but it can struggle in cases where the clusters are not of equal size or are non-spherical in shape. Visualization: When you visualize the clusters in 2D (using PCA), you will see that the K-means clusters might appear to form distinct, non-overlapping groups, whereas GMM might show more fluid, overlapping groupings due to the probabilistic assignments.

Adjusted Rand Index (ARI): This measure compares the clustering result with the true labels, adjusting for chance. A higher value indicates better clustering. GMM might perform slightly better if the clusters are not perfectly spherical, but K-means might still give reasonable results.

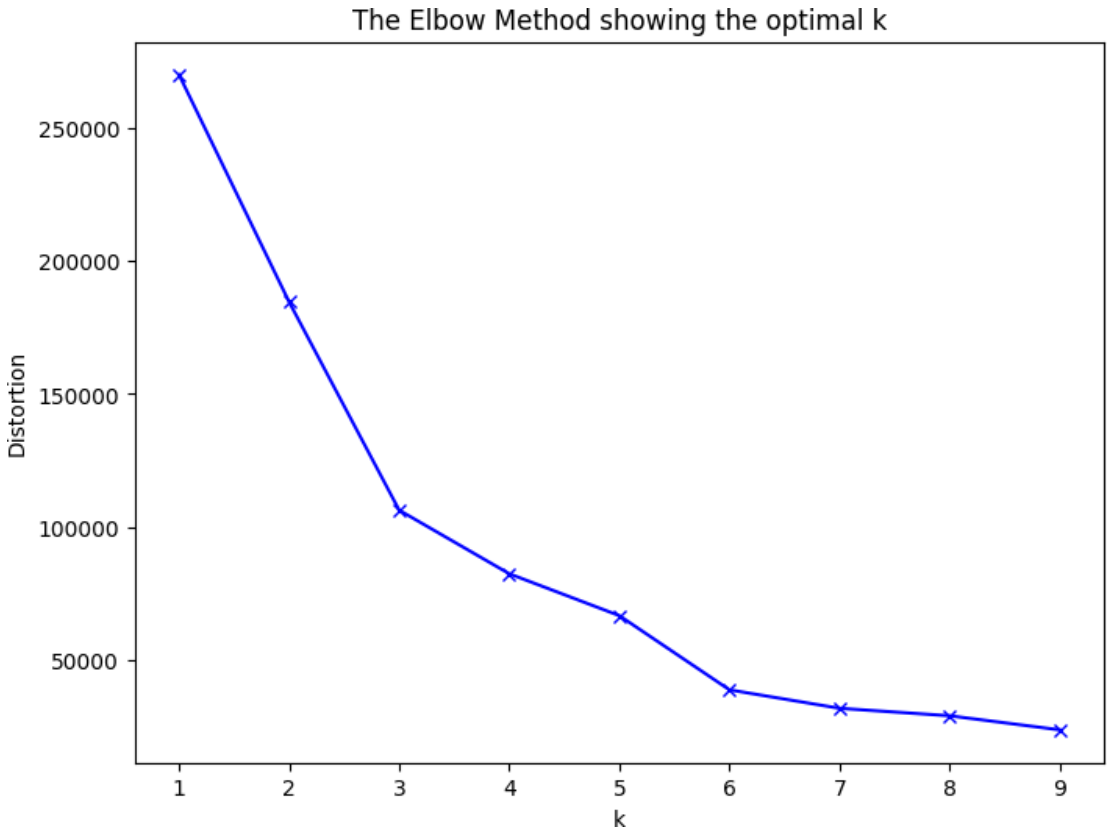
Homogeneity: This metric tells you whether each cluster only contains data points from one class. If K-means or GMM clusters are more homogeneous, it indicates that the algorithm is doing well in separating different classes of the Iris dataset.

[ ]:
```
#Second dataset
import pandas as pd
df = pd.read_csv('/content/Mall_Customers.csv')
df.head()
```

[ ]:

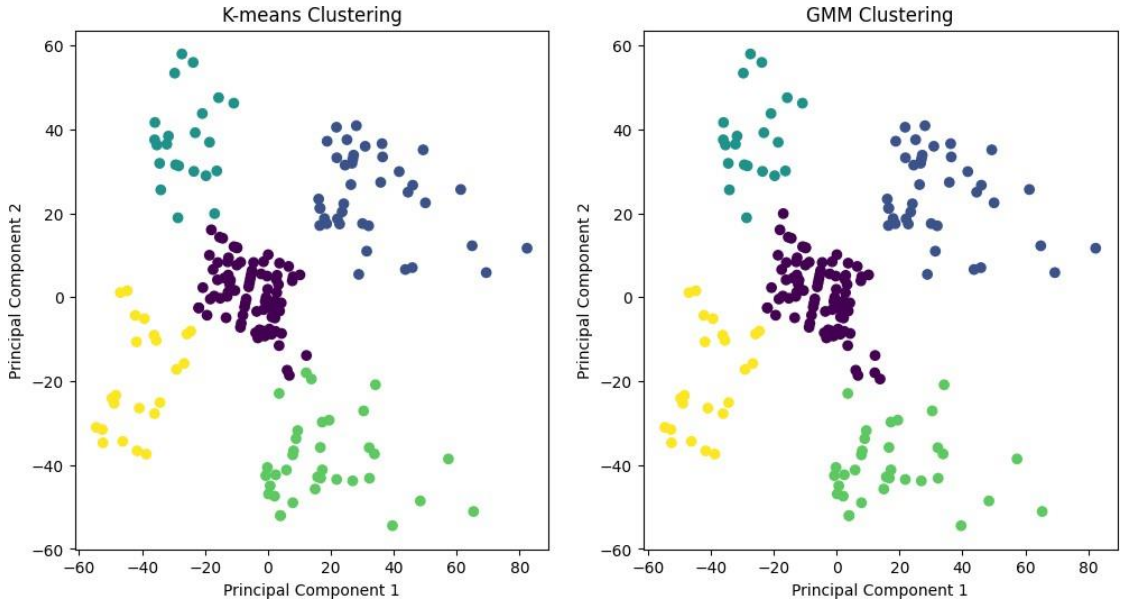
	CustomerID	Gender	Age	Annual Income (k$)	Spending Score (1-100)
0	1	Male	19	15	39
1	2	Male	21	15	81
2	3	Female	20	16	6
3	4	Female	23	16	77
4	5	Female	31	17	40

[ ]:
```
# Prepare the data (assuming 'df' is your dataframe and you want to use Annual Income and Spending Score)
X_mall = df[['Annual Income (k$)', 'Spending Score (1-100)']]

# Calculate distortions for different numbers of clusters (e.g., 1 to 10)
distortions = []
K = range(1, 10)
for k in K:
    kmeanModel = KMeans(n_clusters=k)
    kmeanModel.fit(X)
    distortions.append(kmeanModel.inertia_)

# Create the elbow plot
plt.figure(figsize=(8, 6))
plt.plot(K, distortions, 'bx-')
plt.xlabel('k')
plt.ylabel('Distortion')
plt.title('The Elbow Method showing the optimal k')
plt.show()
```



[ ]:
```
#Second dataset
from sklearn.metrics import silhouette_score # Import silhouette_score

kmeans = KMeans(n_clusters=5, random_state=42)
kmeans_labels = kmeans.fit_predict(X_mall)

# Gaussian Mixture Model (GMM) clustering
gmm = GaussianMixture(n_components=5, random_state=42)
gmm_labels = gmm.fit_predict(X_mall)

# PCA for dimensionality reduction
pca = PCA(n_components=2)
X_mall_pca = pca.fit_transform(X_mall)

# Visualize K-means clustering
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.scatter(X_mall_pca[:, 0], X_mall_pca[:, 1], c=kmeans_labels, cmap='viridis')
plt.title('K-means Clustering')
plt.xlabel('Principal Component 1')
```

```
plt.ylabel('Principal Component 2')

# Visualize GMM clustering
plt.subplot(1, 2, 2)
plt.scatter(X_mall_pca[:, 0], X_mall_pca[:, 1], c=gmm_labels, cmap='viridis')
plt.title('GMM Clustering')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.show()

# Evaluate clustering performance using Silhouette Score
kmeans_silhouette = silhouette_score(X_mall, kmeans_labels)
gmm_silhouette = silhouette_score(X_mall, gmm_labels)

print(f"Silhouette Score (K-means): {kmeans_silhouette}")
print(f"Silhouette Score (GMM): {gmm_silhouette}")
```



Silhouette Score (K-means): 0.553931997444648
Silhouette Score (GMM): 0.5528243704895652

Result:

Comparison between k-means and GMM

K-means and Gaussian Mixture Models (GMMs) are both clustering algorithms, but they differ in several ways regarding their output and approach:

Cluster Shape:

K-means: Assumes that clusters are spherical (circular) and evenly sized. It works well when the clusters are relatively isotropic (i.e., similar in size and shape). GMM: Makes fewer assumptions about the shape of the clusters. Clusters can have different shapes, sizes, and orientations because GMM models each cluster as a Gaussian distribution, which can be elongated or have different variances. Cluster Representation:

K-means: Each cluster is represented by a single point, called the centroid. Each data point is assigned to exactly one cluster, based on the closest centroid. GMM: Each cluster is represented by a Gaussian distribution with parameters (mean, covariance). A data point has a probability of belonging to each cluster, not a hard assignment. Cluster Membership:

K-means: The assignment of data points to clusters is "hard," meaning each point is assigned to exactly one cluster, with no overlap. GMM: The assignment is "soft." Each data point has a probability of belonging to each cluster, which is based on the likelihood of that point given the Gaussian distributions. Centroids vs. Probabilities:

K-means: The output consists of centroids for each cluster and the assignment of each data point to a cluster. GMM: The output consists of the parameters of the Gaussian distributions (means and covariances) for each cluster, along with the probability that each data point belongs to each cluster. Variance Handling:

K-means: Assumes all clusters have the same variance (since they are spherical and equidistant from the centroid). GMM: Allows for different variances (covariances) for each cluster. This is useful for modeling more complex cluster shapes. Sensitivity to Initialization:

K-means: Can be sensitive to initial cluster centroid placement, potentially converging to a subop- timal solution. GMM: Also sensitive to initialization, but it often uses techniques like Expectation- Maximization (EM) to refine the estimates of the parameters. Likelihood:

K-means: Does not explicitly model the likelihood of the data under the clustering assignment. It minimizes squared Euclidean distance. GMM: Maximizes the likelihood of the data under a mix- ture of Gaussian distributions, which is more probabilistic and allows for more nuanced clustering behavior. In summary, K-means tends to work better when clusters are compact and roughly equal in size, while GMM is more flexible and can handle varying cluster shapes and densities due to its probabilistic nature.

SVM with linear data

February 19th, 2025

AIM: To implement Support Vector Machine (SVM) for classification on linearly separable and non-linearly separable 2D datasets, visualize the decision boundary and margins, and compare the performance of SVM with different kernels**.**

ALGORITHM

Step 1: Start

Step 2: Import necessary libraries.

Step 3: Generate datasets using make_classification for linear data and make_moons for non-linearly separabale data

Step 4: Split the data into train and split

Step 5: Train the linearly separable data with linear kernel and explore all kernels for non linearly separable data.

Step 6: Get the predicted values and evalute the performance using accuracy_score and classification_report.

Step 7: Visualise the decision boundary, margin and support vectors.

Step 8: Compare the linear and non linear performance. Compare the different kernel functions' performance for non linear data.

Step 9: End

MODEL DESCRIPTION

Support Vector Machine (SVM) SVM is a supervised machine learning algorithm used for classification tasks. It finds the hyperplane that best separates data points of different classes with the maximum margin, using support vectors (data points closest to the decision boundary) to define the boundary.

Linear SVM: - Kernel: A linear kernel assumes the classes are linearly separable and tries to find a straight line (or hyperplane in higher dimensions) that separates the classes. - Use Case: Best for datasets where the decision boundary is a straight line.

Non-Linear SVM: - Kernel: A non-linear kernel maps the data into a higher-dimensional space, where a linear decision boundary can separate the data. Common kernels include:

Radial Basis Function (RBF): Popular for complex, non-linear data.
Polynomial: Maps data to a higher-dimensional space based on polynomial functions. Sigmoid: Inspired by neural networks, useful for data that can be separated by a sigmoid-l

- Use Case: Best for datasets where the decision boundary is non-linear and complex.

Mathematical Formulation: - Linear SVM: The decision boundary is defined by w x+b=0w x+b=0, where ww is the weight vector and bb is the bias. The goal is to maximize the margin between classes. - Non-Linear SVM: The decision boundary is found using kernel functions (e.g., RBF, polynomial, sigmoid) that transform data into a higher-dimensional space, where a linear decision boundary can separate the classes.

Plot the Decision Boundary, Margin, and Support Vectors

[1]:
```
def plot_svm_decision_boundary_with_margin(X, y, clf, title="SVM Decision
Boundary with Margin"):
    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))

    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    plt.contourf(xx, yy, Z, alpha=0.75, cmap=plt.cm.coolwarm)
    plt.scatter(X[:, 0], X[:, 1], c=y, edgecolors='k', cmap=plt.cm.coolwarm)

    # Extract the support vectors using the indices from clf.support_
    support_vectors = X[clf.support_]

    # Highlight support vectors
    plt.scatter(support_vectors[:, 0], support_vectors[:, 1],
facecolors='none', edgecolors='yellow', s=100, label="Support Vectors")

    # Plot the decision boundary
    ax = plt.gca()
    xlim = ax.get_xlim()
    ylim = ax.get_ylim()

    # Create a grid of points to plot decision boundary
```

```
    xx, yy = np.meshgrid(np.linspace(xlim[0], xlim[1], 30),
                         np.linspace(ylim[0], ylim[1], 30))
    Z = clf.decision_function(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    # Plot margins (decision boundary ± 1)
    plt.contour(xx, yy, Z, levels=[-1, 1], alpha=0.7, colors='k',
 linestyles='dashed')

    # Plot decision boundary (where the decision function is 0)
    plt.contour(xx, yy, Z, levels=[0], alpha=0.7, colors='b', linewidths=2)

    plt.title(title)
    plt.xlabel('Feature 1')
    plt.ylabel('Feature 2')
    plt.legend()
```

1.1.1 Linearly Separable Data

Import Libraries

[2]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report
```

Generate data

[3]:
```
X, y = datasets.make_classification(n_samples=100, n_features=2,
 n_informative=2, n_redundant=0,
                                    n_classes=2, random_state=42)
```

Split data

[4]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
 random_state=42)
```

Fit the classifier

[5]:
```
clf = SVC(kernel='linear')
clf.fit(X_train, y_train)
```

[5] :
```
SVC(kernel='linear')
```

Predict and Performance Evaluation

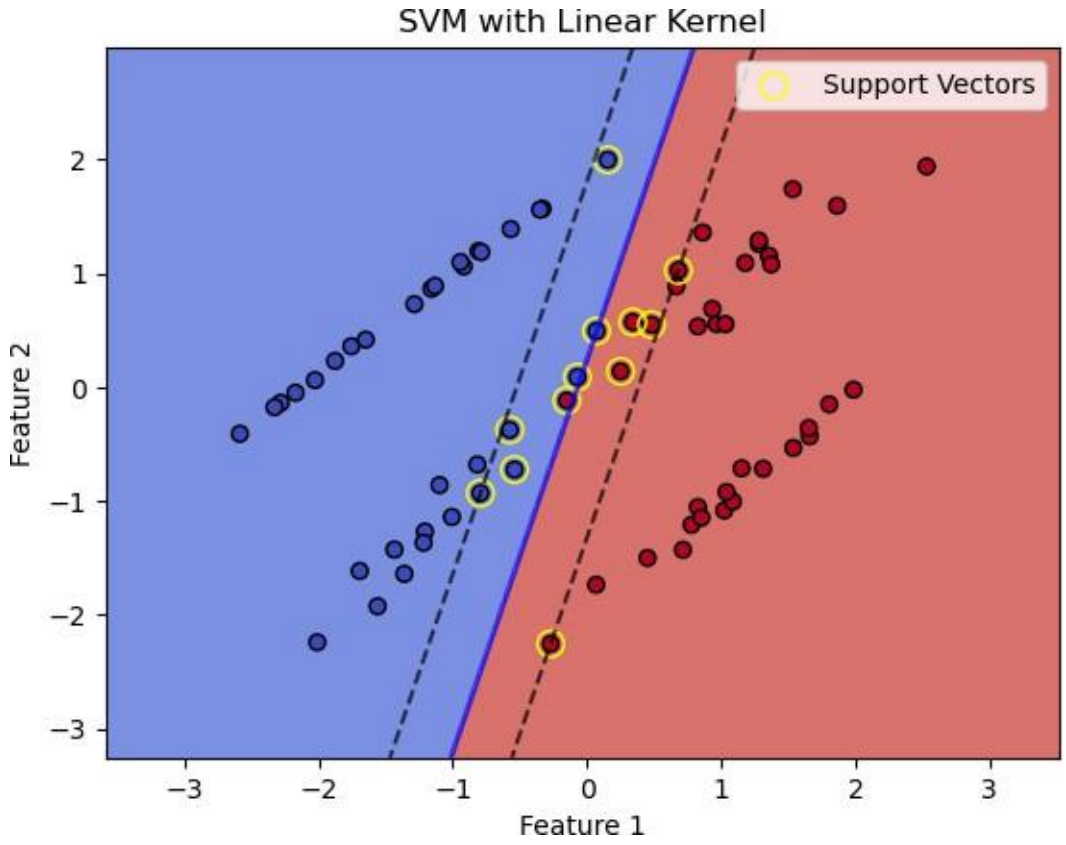
[6] :
```
y_pred = clf.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
class_rep = classification_report(y_test, y_pred)
print(class_rep)
print("Accuracy score: ", accuracy)
```

	precision	recall	f1-score	support
0	1.00	1.00	1.00	16
1	1.00	1.00	1.00	14
accuracy			1.00	30
macro avg	1.00	1.00	1.00	30
weighted avg	1.00	1.00	1.00	30

Accuracy score: 1.0

Visualisation

[7] :
```
plot_svm_decision_boundary_with_margin(X_train, y_train,clf, title=f"SVM with␣
␣Linear Kernel")
```



1.1.2 Non- Linearly Separable Data

Import Libraries

[8] :
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report
```

Generate data

[9] :
```
X_nl, y_nl = datasets.make_moons(n_samples=100, noise=0.1, random_state=42)
```

Split data

[10] :
```
X_train_nl, X_test_nl, y_train_nl, y_test_nl = train_test_split(X_nl, y_nl,
    test_size=0.3, random_state=42)
```

Fit Classifier

[11] :
```
#Linear
clf_linear = SVC(kernel='linear')
clf_linear.fit(X_train_nl, y_train_nl)
```

[11] :
```
SVC(kernel='linear')
```

[12] :
```
#Polynomial
clf_poly = SVC(kernel='poly')
clf_poly.fit(X_train_nl, y_train_nl)
```

[12] :
```
SVC(kernel='poly')
```

[13] :
```
#Gaussian
clf_rbf = SVC(kernel='rbf')
clf_rbf.fit(X_train_nl, y_train_nl)
```

[13] :
```
SVC()
```

Prediction and Evaluation

[14] :
```
print("LINEAR KERNEL")
y_pred_nl = clf_linear.predict(X_test_nl)
accuracy_linear = accuracy_score(y_test_nl, y_pred_nl)
class_rep_linear = classification_report(y_test_nl, y_pred_nl)
print(class_rep_linear)
print("Accuracy score: ", accuracy_linear)
```

```
LINEAR KERNEL
              precision    recall  f1-score   support

           0       1.00      0.80      0.89        20
           1       0.71      1.00      0.83        10

    accuracy                           0.87        30
   macro avg       0.86      0.90      0.86        30
weighted avg       0.90      0.87      0.87        30

Accuracy score:  0.8666666666666667
```

[15] :
```
print("POLYNOMIAL KERNEL")
y_pred_nl = clf_poly.predict(X_test_nl)
accuracy_poly = accuracy_score(y_test_nl, y_pred_nl)
class_rep_poly = classification_report(y_test_nl, y_pred_nl)
print(class_rep_poly)
print("Accuracy score: ", accuracy_poly)
```

```
POLYNOMIAL KERNEL
              precision    recall  f1-score   support

           0       1.00      0.80      0.89        20
           1       0.71      1.00      0.83        10

    accuracy                           0.87        30
   macro avg       0.86      0.90      0.86        30
weighted avg       0.90      0.87      0.87        30

Accuracy score:  0.8666666666666667
```

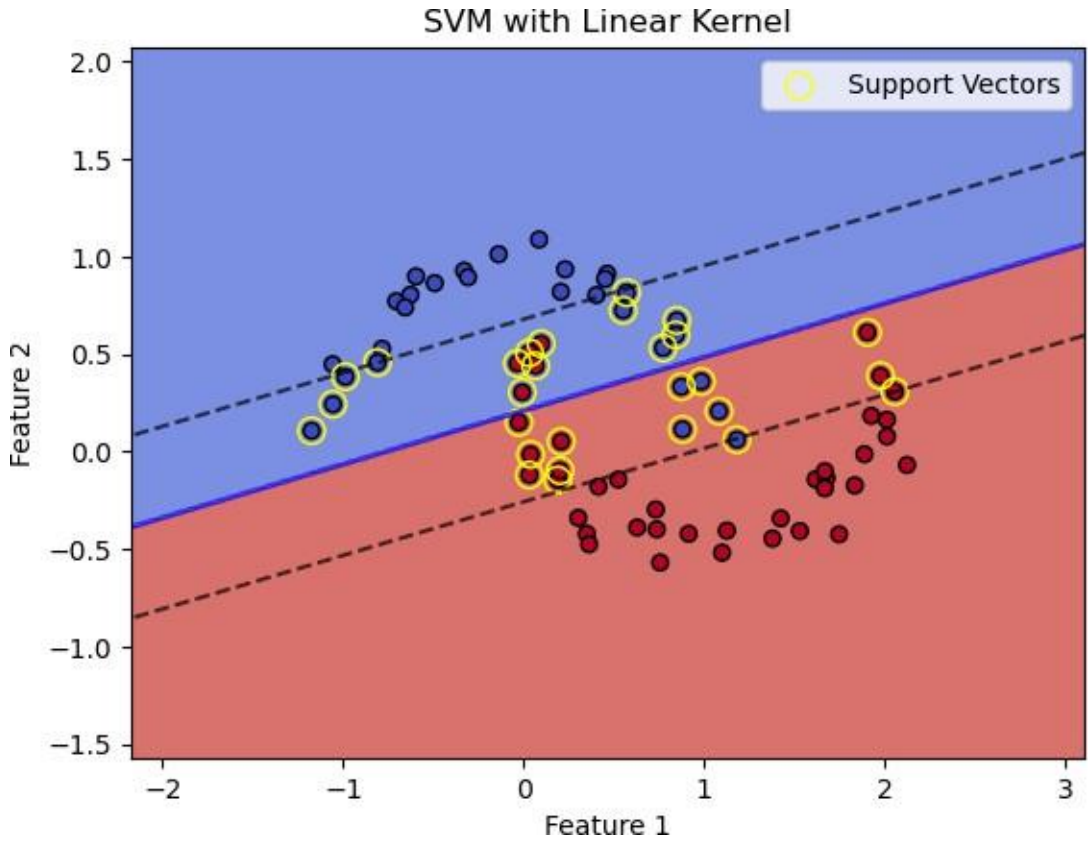
[16] :
```
print("RBF KERNEL")
y_pred_nl = clf_rbf.predict(X_test_nl)
accuracy_rbf = accuracy_score(y_test_nl, y_pred_nl)
class_rep_rbf = classification_report(y_test_nl, y_pred_nl)
print(class_rep_rbf)
print("Accuracy score: ", accuracy_rbf)
```

```
RBF KERNEL
              precision    recall  f1-score   support

           0       1.00      0.95      0.97        20
           1       0.91      1.00      0.95        10

    accuracy                           0.97        30
   macro avg       0.95      0.97      0.96        30
weighted avg       0.97      0.97      0.97        30
```

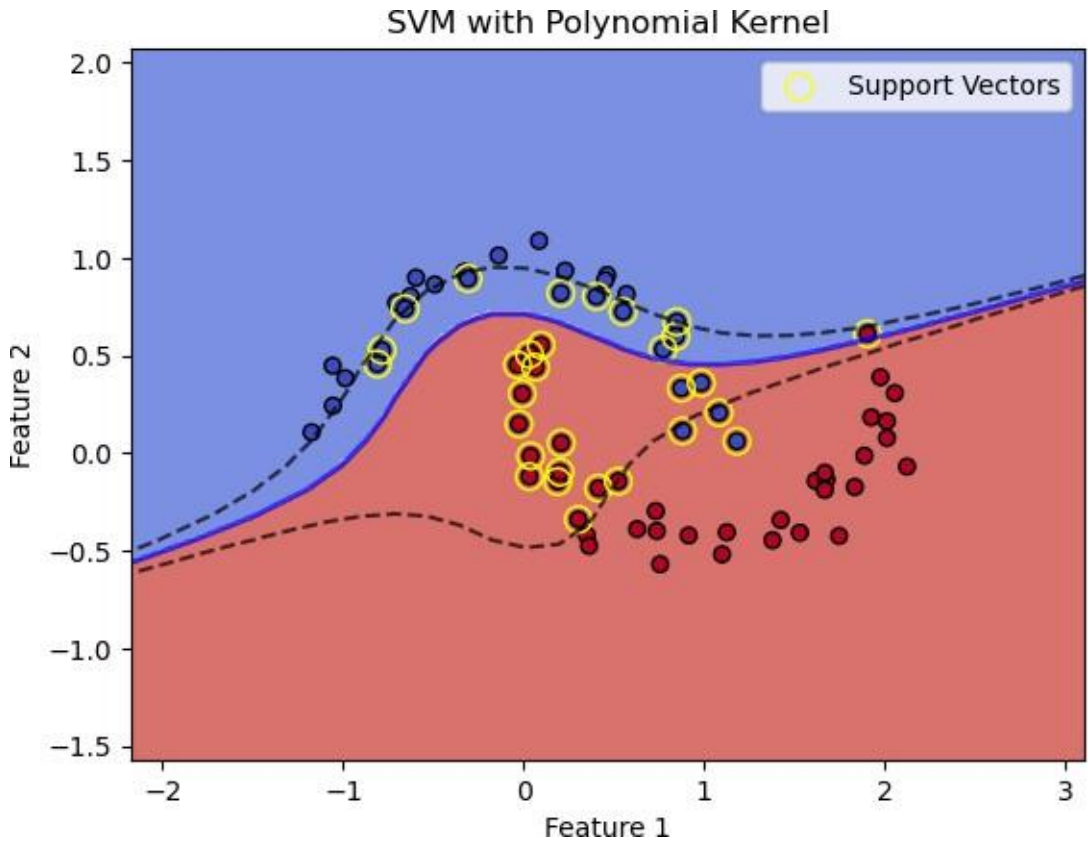
Accuracy score: 0.9666666666666667

Visualisation

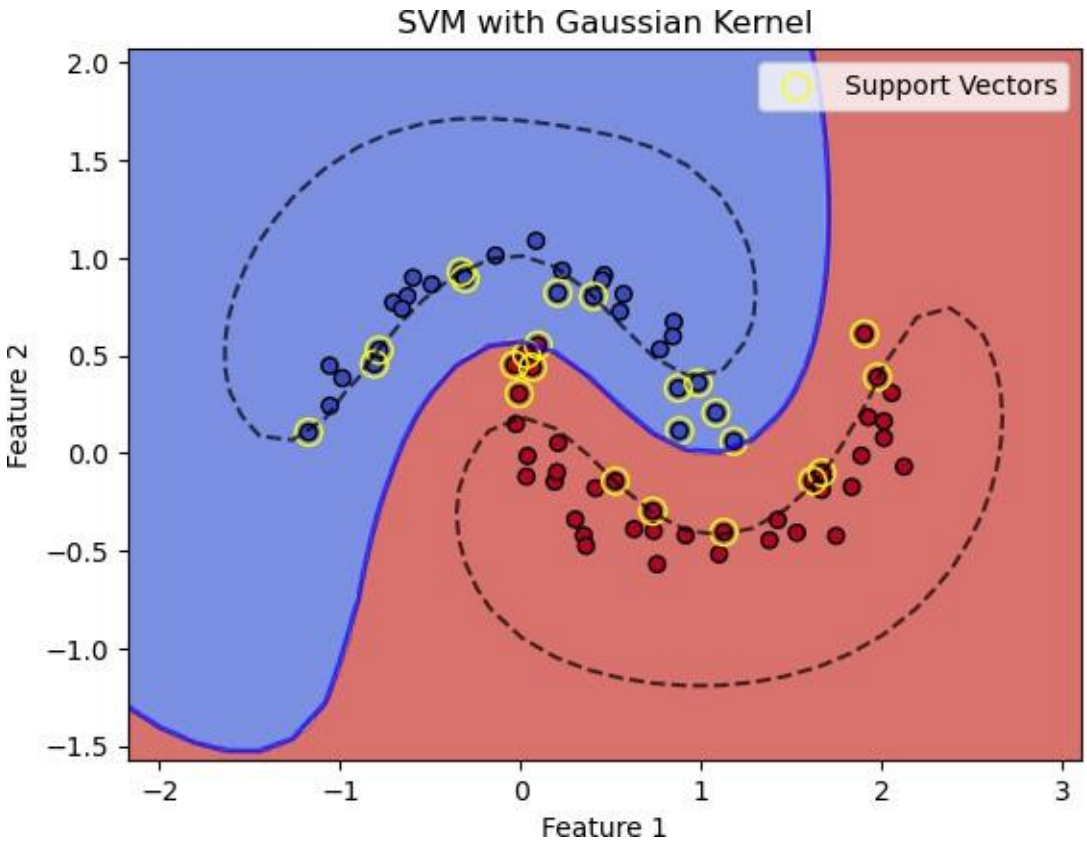
[17] :
```
plot_svm_decision_boundary_with_margin(X_train_nl, y_train_nl,clf_linear,␣
  ▫title=f"SVM with Linear Kernel")
```



[18] :
```
plot_svm_decision_boundary_with_margin(X_train_nl, y_train_nl,clf_poly,␣
  ▫title=f"SVM with Polynomial Kernel")
```



[19] :
```
plot_svm_decision_boundary_with_margin(X_train_nl,  y_train_nl,clf_rbf,␣
  title=f"SVM  with  Gaussian  Kernel")
```



RESULT

The linear SVM performed well on linearly separable data, achieving high accuracy with a clear decision boundary. For non-linear data, the RBF kernel outperformed the polynomial and linear kernels, providing the most effective decision boundary and highest accuracy. Therefore, the RBF kernel is preferred for non-linear datasets due to its flexibility and superior performance**.**

SVM OvO, OvA

February 26th 2025

AIM: Perform multiclass classification using One v One and One v All classifiers Algorithm

DESCRIPTION:

One-vs-One (OVO)

- **Strategy**: One classifier for **each pair** of classes.
- **Number of classifiers**: n(n−1)2\frac{n(n - 1)}{2}2n(n−1)
- **Training**: Uses only two classes per model.
- **Prediction**: **Voting** system — class with most votes wins.
- **Pros**: Better for small class numbers, handles class imbalance well.
- **Cons**: Slow with many classes.

One-vs-All (OVA)

- **Strategy**: One classifier for **each class vs the rest**.
- **Number of classifiers**: nnn
- **Training**: Uses all data; labels one class as positive, others negative.
- **Prediction**: Chooses class with highest score.
- **Pros**: Simple, faster with many classes.
- **Cons**: Class imbalance may affect performance.

Algorithm:

SVM on Digits Dataset

Step 1: Import Necessary Libraries Import numpy, mat- plotlib.pyplot for visualization.

Import sklearn.datasets.load_digits to load the dataset.

Import train_test_split, StandardScaler, SVC, confusion_matrix, classification_report, Grid-SearchCV from sklearn.

Step 2: Load and Visualize the Digits Dataset Load the dataset using load_digits().

Display some images using matplotlib.pyplot.imshow().

Step 3: Preprocessing the Data Flatten the images if needed.

Scale the data using StandardScaler() to normalize feature values.

Split data into training and testing sets.

Step 4: Train a Support Vector Machine (SVM) Classifier Use SVC() with a default linear kernel.

Train the model on the training data.

Step 5: Implement One-vs-One (OvO) Classification Train an OvO SVM classifier using SVC(decision_function_shape='ovo').

Make predictions and evaluate accuracy.

Step 6: Implement One-vs-All (OvA) Classification Train an OvA SVM classifier using SVC(decision_function_shape='ovr').

Make predictions and compare accuracy with OvO.

Step 7: Compare Performance Compute accuracy scores for OvO and OvA.

Generate classification reports and confusion matrices for both.

Step 8: Hyperparameter Tuning Using GridSearchCV Perform tuning on C, gamma, and kernel type (linear, poly, rbf, sigmoid).

Find the best hyperparameters and retrain the model.

Step 9: Evaluate Performance for Different Kernels Train separate SVM models for linear, poly, rbf, and sigmoid kernels.

Compute and compare their confusion matrices. Analyze which kernel performs best.

Step 10: Summarize Insights Which classification strategy (OvO vs OvA) performs better? Which kernel gives the best accuracy?

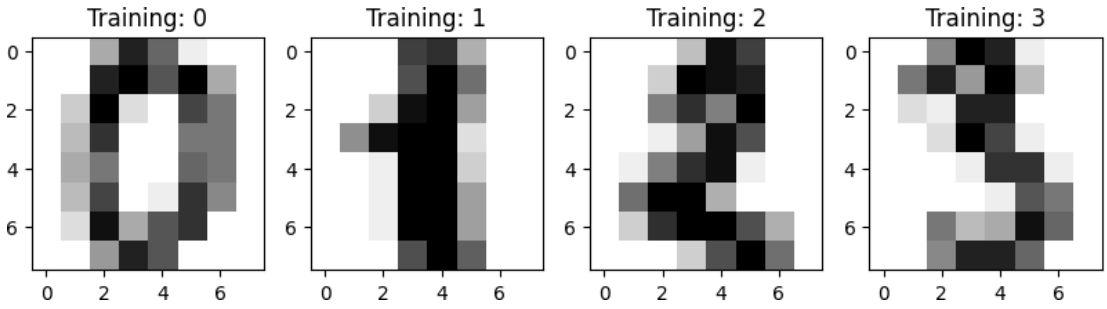
[ ]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.svm import SVC
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.model_selection import train_test_split
```

[ ]:
```
from sklearn import datasets
digits = datasets.load_digits()
from sklearn.multiclass import OneVsOneClassifier, OneVsRestClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score
```

[ ]:
```
print(digits.data[0])
```

```
[ 0.  0.  5. 13.  9.  1.  0.  0.  0.  0. 13. 15. 10. 15.  5.  0.  0.  3.
 15.  2.  0. 11.  8.  0.  0.  4. 12.  0.  0.  8.  8.  0.  0.  5.  8.  0.
  0.  9.  8.  0.  0.  4. 11.  0.  1. 12.  7.  0.  0.  2. 14.  5. 10. 12.
  0.  0.  0.  0.  6. 13. 10.  0.  0.  0.]
```

[ ]:
```
digits = datasets.load_digits()

# Create a figure with 1 row and 4 columns of subplots (axes)
_, axes = plt.subplots(nrows=1, ncols=4, figsize=(10, 3))

# Loop through the axes (subplot positions), images, and corresponding labels
# from the Digits dataset
for ax, image, label in zip(axes, digits.images, digits.target):

    # Display the image in the current subplot using the "gray_r" color map
    # (reversed grayscale)
    ax.imshow(image, cmap=plt.cm.gray_r, interpolation="nearest")

    # Set the title of the subplot to display the corresponding label (digit)
    # of the image
    ax.set_title("Training: %i" % label)
```



[ ]:

[ ]:
```
# flatten the images
n_samples = len(digits.images)
data = digits.images.reshape((n_samples, -1))


X_train, X_test, y_train, y_test = train_test_split(data, digits.target,␣
  ↪test_size=0.5)

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# One-vs-One (OvO)
ovo_svm = OneVsOneClassifier(SVC(kernel='linear'))
ovo_svm.fit(X_train, y_train)
y_pred_ovo = ovo_svm.predict(X_test)
print("OvO Accuracy:", accuracy_score(y_test, y_pred_ovo))

# One-vs-All (OvA)
ova_svm = OneVsRestClassifier(SVC(kernel='linear'))
ova_svm.fit(X_train, y_train)
y_pred_ova = ova_svm.predict(X_test)
print("OvA Accuracy:", accuracy_score(y_test, y_pred_ova))
```

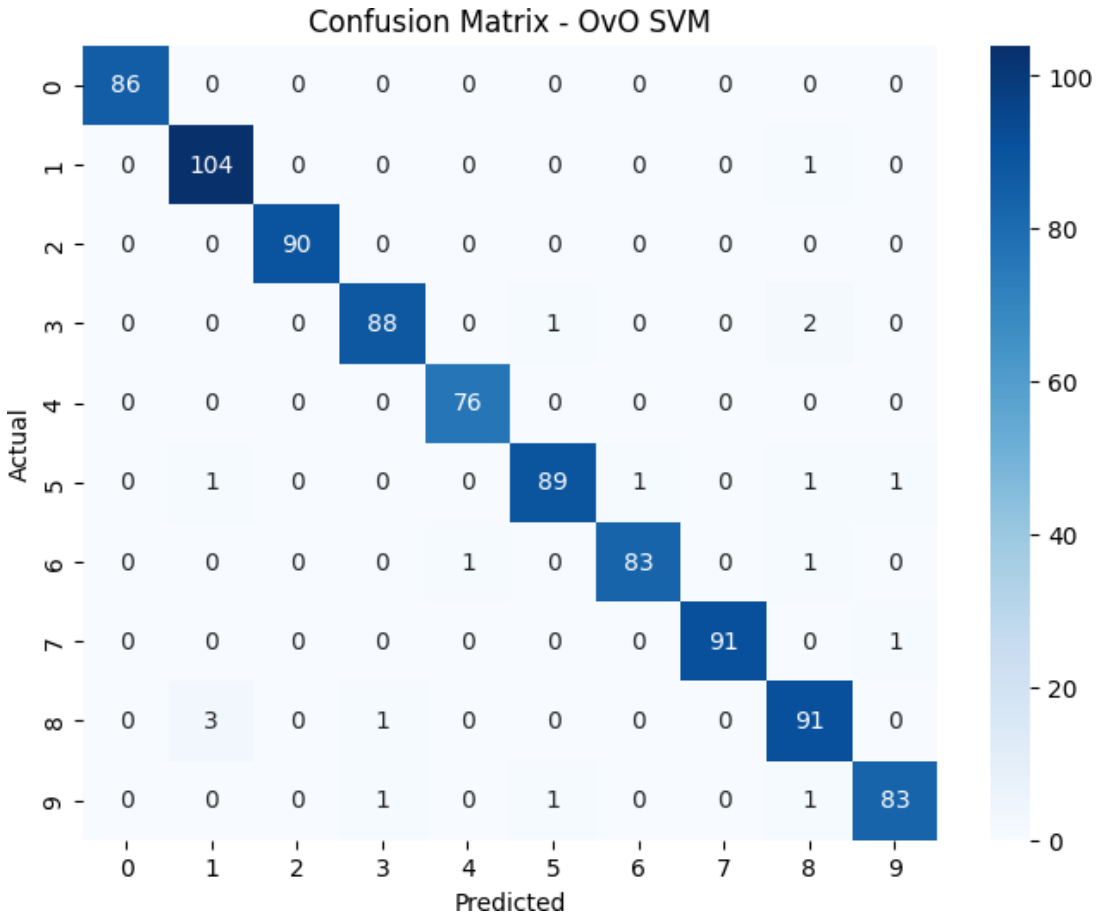
```
OvO Accuracy: 0.9799777530589544
OvA Accuracy: 0.92880978865406
```

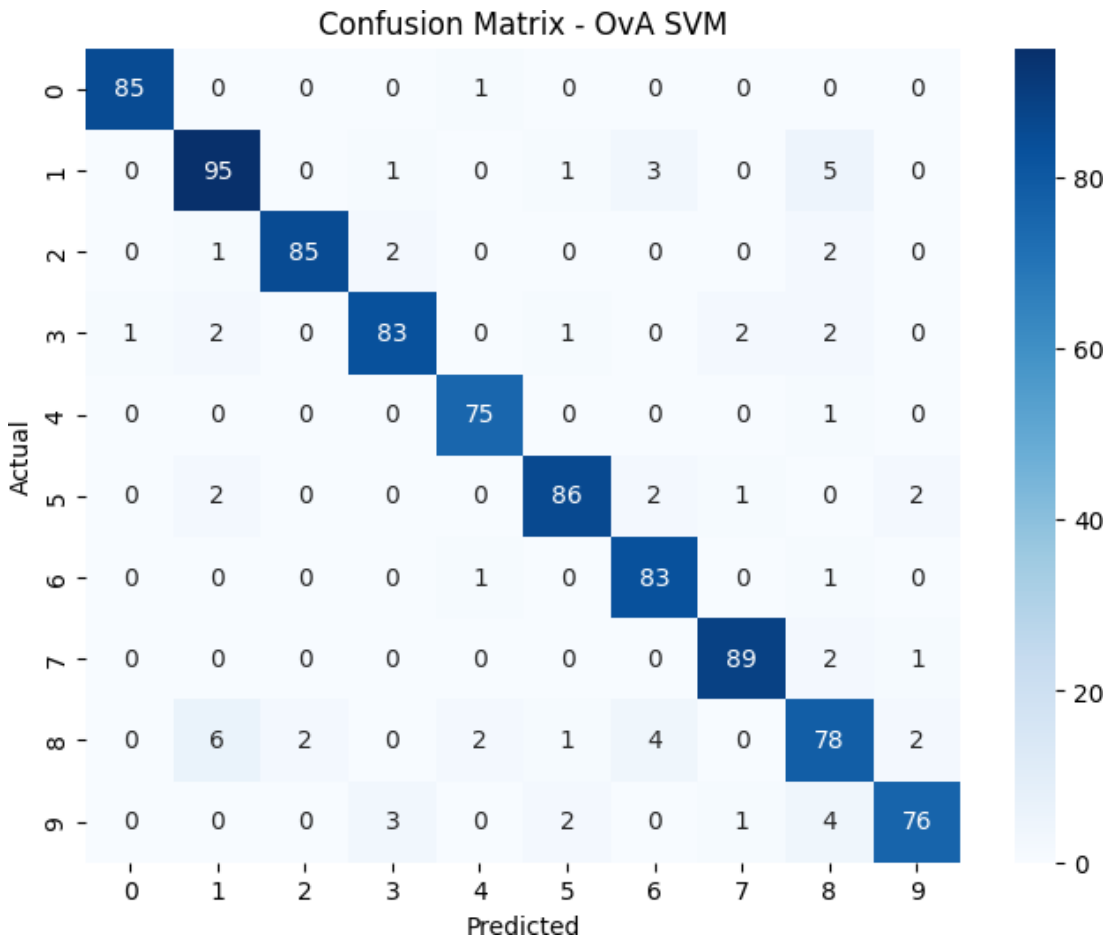
[ ]:
```
from sklearn import metrics
from sklearn.metrics import accuracy_score, classification_report,␣
  ↪confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt
```

```
[ ]: print("OvO Accuracy:", accuracy_score(y_test, y_pred_ovo))
     print("OvA Accuracy:", accuracy_score(y_test, y_pred_ova))

     # Print Classification Report
     print("\nOvO Classification Report:\n", classification_report(y_test,
       y_pred_ovo))
     print("\nOvA Classification Report:\n", classification_report(y_test,
       y_pred_ova))

     # Plot Confusion Matrices
     def plot_confusion_matrix(y_true, y_pred, title):
         cm = confusion_matrix(y_true, y_pred)
         plt.figure(figsize=(8,6))
         sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=range(10),
       yticklabels=range(10))
         plt.xlabel('Predicted')
         plt.ylabel('Actual')
         plt.title(title)
         plt.show()

     plot_confusion_matrix(y_test, y_pred_ovo, "Confusion Matrix - OvO SVM")
     plot_confusion_matrix(y_test, y_pred_ova, "Confusion Matrix - OvA SVM")
```

```
OvO Accuracy: 0.9799777530589544
OvA Accuracy: 0.92880978865406

OvO Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        86
           1       0.96      0.99      0.98       105
           2       1.00      1.00      1.00        90
           3       0.98      0.97      0.97        91
           4       0.99      1.00      0.99        76
           5       0.98      0.96      0.97        93
           6       0.99      0.98      0.98        85
           7       1.00      0.99      0.99        92
           8       0.94      0.96      0.95        95
           9       0.98      0.97      0.97        86

    accuracy                           0.98       899
   macro avg       0.98      0.98      0.98       899
weighted avg       0.98      0.98      0.98       899


OvA Classification Report:
              precision    recall  f1-score   support
```

0	0.99	0.99	0.99	86
1	0.90	0.90	0.90	105
2	0.98	0.94	0.96	90
3	0.93	0.91	0.92	91
4	0.95	0.99	0.97	76
5	0.95	0.92	0.93	93
6	0.90	0.98	0.94	85
7	0.96	0.97	0.96	92
8	0.82	0.82	0.82	95
9	0.94	0.88	0.91	86
accuracy			0.93	899
macro avg	0.93	0.93	0.93	899
weighted avg	0.93	0.93	0.93	899





HYPERPARAMETER TUNING

Gamma ( ) is a parameter used in kernels that controls the reach or extent of the influence of each support vector on the decision boundary.

When gamma is small: The decision boundary becomes smoother and less complex because each support vector has a larger influence on the region around it.

When gamma is large: The decision boundary becomes highly complex and curved, closely following the training data.

gamma='scale' sets gamma to 1 / (n_features * X.var()), where X.var() is the variance of the input data, and n_features is the number of features. This is often a good starting point.

'scale' (default) uses the data's variance and number of features to set gamma. It adapts the kernel more to the specific characteristics of the data. 'auto' uses only the number of features, which can work for high dimensiona data

The C parameter is a regularization parameter that determines the penalty for misclassified points in the training data. Essentially, it controls how much you want to avoid misclassification of the training data.

High values of C mean less regularization (more emphasis on minimizing classification errors). Low values of C mean more regularization (more flexibility to allow misclassifications to get a smoother decision boundary).

Kernel Function is a method used to take data as input and transform it into the required form of processing data.

1. Linear Kernel A linear kernel is the simplest form of kernel used in SVM. It is suitable when the data is linearly separable meaning that a straight line (or hyperplane in higher dimensions) can effectively separate the classes. It is represented as:

K(x,y)=x.y It is used for text classification problems such as spam detection

2. Polynomial Kernel The polynomial kernel allows SVM to model more complex relationships by introducing polynomial terms. It is useful when the data is not linearly separable but still follows a pattern. The formula of Polynomial kernel is:

K(x,y)=(x.y+c)^d

where is a constant and d is the polynomial degree. It is used in Complex problems like image recognition where relationships between features can be non-linear.

3. Radial Basis Function Kernel (RBF) Kernel The RBF kernel is the most widely used kernel in SVM. It maps the data into an infinite-dimensional space making it highly effective for complex classification problems. The formula of RBF kernel is:

K(x,y)=e – (x–y 2) where is a parameter that controls the influence of each training example. We use RBF kernel When the decision boundary is highly non-linear and we have no prior knowledge about the data's structure is available.

[ ]:
```
from sklearn.model_selection import GridSearchCV

# Set the parameters grid
param_grid = {
    'C': [0.1, 1, 10],                  # Regularization strength
    'kernel': ['linear', 'rbf', 'poly'],       # Types of kernels
    'gamma': ['scale', 'auto'],         # Kernel coefficient
}

# Initialize the SVC model
svc = SVC()

# Perform Grid Search with 5-fold cross-validation
grid_search = GridSearchCV(svc, param_grid, cv=5, n_jobs=-1, verbose=2)

# Fit the model to the training data
grid_search.fit(X_train, y_train)

# Display the best parameters from the grid search
print(f"Best parameters from Grid Search: {grid_search.best_params_}")
```

```
Fitting 5 folds for each of 18 candidates, totalling 90 fits
Best parameters from Grid Search: {'C': 10, 'gamma': 'auto', 'kernel': 'rbf'}
```

[ ]:
```python
# Get the best model from the grid search
best_model = grid_search.best_estimator_

# Evaluate the performance on the test set
accuracy = best_model.score(X_test, y_test)
print(f"Test set accuracy: {accuracy:.4f}")
```

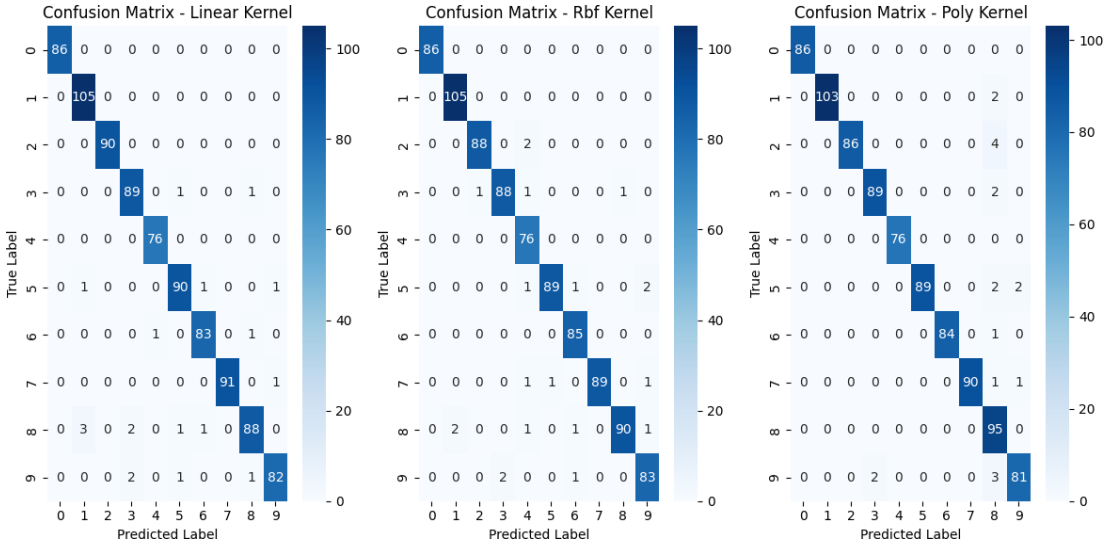
```
Test set accuracy: 0.9789
```

[ ]:
```python
kernels = ['linear', 'rbf', 'poly']
```

[ ]:
```python
plt.figure(figsize=(12, 6))

# Loop through each kernel type and evaluate the model
for i, kernel in enumerate(kernels, start=1):
    # Initialize the SVC with the current kernel
    svc = SVC(kernel=kernel, C=10, gamma='scale', random_state=42)

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

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

    # Scores
    y_pred = svc.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)
    print(f"Accuracy for {kernel.capitalize()} Kernel: {accuracy}")

    # Generate confusion matrix
    cm = confusion_matrix(y_test, y_pred)

    # Plot confusion matrix
    plt.subplot(1, 3, i)
    sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=digits.
  target_names, yticklabels=digits.target_names)
    plt.title(f"Confusion Matrix - {kernel.capitalize()} Kernel")
    plt.xlabel('Predicted Label')
    plt.ylabel('True Label')

# Show the plots
plt.tight_layout()
plt.show()
```

```
Accuracy for Linear Kernel: 0.978865406006674
```

Accuracy for Rbf Kernel: 0.9777530589543938
Accuracy for Poly Kernel: 0.9777530589543938



Lab viva:

The F1 score is a performance metric used in classification tasks, which combines both precision and recall into a single score. It is particularly useful when dealing with imbalanced datasets, where accuracy might not be the best measure of performance.

The F1 score is the harmonic mean of precision and recall, meaning it gives a balanced view of both metrics.

The formula for the F1 score is: 2(Precision * Recall)/(Precision + Recall)

Balanced Metric for Imbalanced Classes:

The F1 score is especially valuable when you have imbalanced datasets, where one class (such as "negative" in a binary classification problem) is much more frequent than the other (like "positive"). In such cases, accuracy can be misleading because a model can achieve high accuracy by simply predicting the majority class most of the time, but this would fail to properly predict the minority class. The F1 score helps give a more balanced view of the model's performance by accounting for both precision and recall.

Trade-off Between Precision and Recall:

The F1 score helps to balance precision and recall. Depending on the problem, you may care more about one over the other: High Precision means that when the model predicts a positive class, it is highly likely to be correct (fewer false positives). High Recall means that the model correctly identifies most of the positive instances (fewer false negatives). The F1 score tries to balance these two and is often a better measure of performance when both false positives and false negatives have a significant cost.

The F1 score provides a single score that encapsulates the model's performance. This makes it

easier to interpret and compare between models. Instead of looking at both precision and recall separately, the F1 score provides a concise way of evaluating the model's ability to classify both the positive and negative classes correctly.

RESULT:

OvO and Linear kernel give best accuracy in this case

SVM with a non-vectorial dataset

March 5th, 2025

Aim
To perform SVM on a non-vectorial dataset

Model Description
The Linear Kernel in Support Vector Machine (SVM) is used when the data is linearly separable.

It finds the optimal hyperplane that maximizes the margin between positive and negative sentiments.

Since the sentiment dataset consists of text-based data, converting it into numerical vectors allows the linear SVM to classify sentiments effectively.

Algorithm:

1. **Data Loading**
   The dataset is loaded into a Pandas DataFrame from an online source, consisting of labeled text reviews (positive and negative).
2. **Data Exploration**
   The dataset structure is analyzed to check for missing values and class distribution. Statistical analysis and visualizations like bar charts for class distribution are performed.
3. **Data Preprocessing**
   - **Text Vectorization:** The textual data is transformed into numerical representations.
   - **Feature Scaling:** TF-IDF values are normalized to ensure better learning.
4. **Model Training**
   A Support Vector Machine (SVM) classifier with a linear kernel is initialized and trained.
5. **Model Evaluation**
   The trained model is evaluated using accuracy, a confusion matrix, and a classification report.

Import necessary libraries

[1]:
```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, confusion_matrix
```

Load the dataset

[2]:
```
df = pd.read_csv("../Source/imdb_labelled.txt", delimiter='\t', header=None,
    names=['text', 'label'])
```

Preprocess the data

[3]:
```
vectorizer = TfidfVectorizer(stop_words='english', max_features=5000)
X = vectorizer.fit_transform(df['text'])
y = df['label']
```

Split the Data

[4]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    random_state=42)
```

Linear Kernel

[5]:
```
svm_model = SVC(kernel='linear')
svm_model.fit(X_train, y_train)
```

[5] :
```
SVC(kernel='linear')
```

Evaluating the model

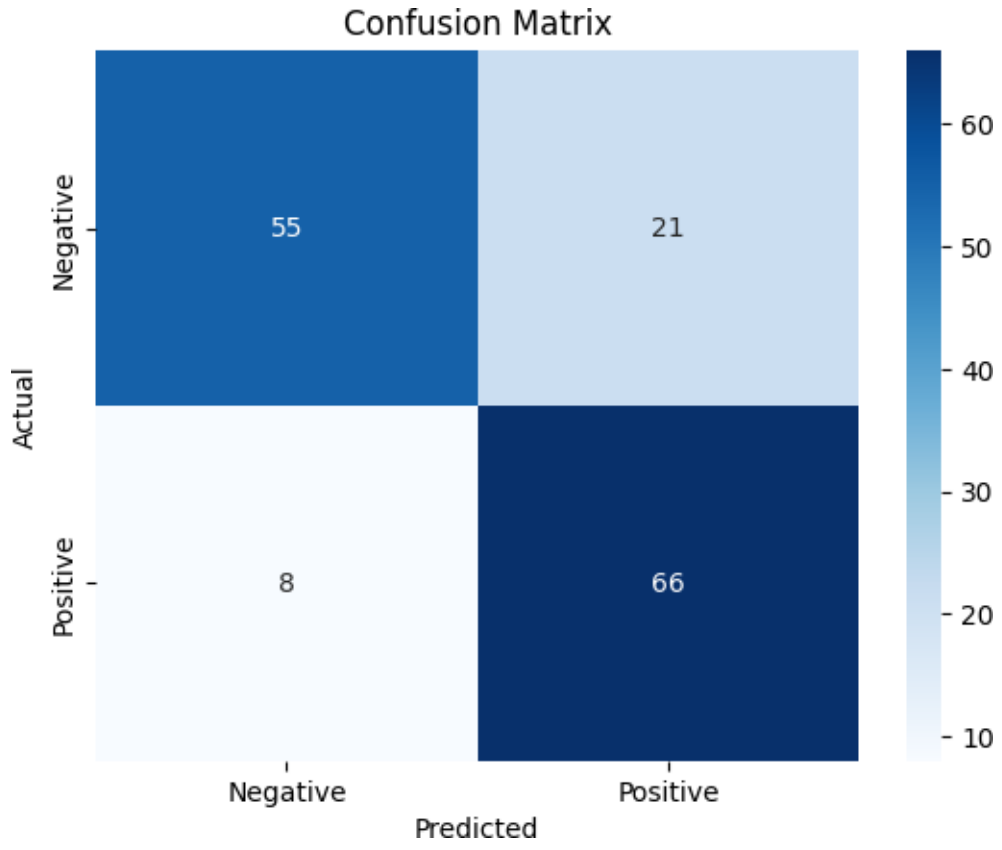
[6] :
```
y_pred = svm_model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy:    {accuracy:.4f}')
```

Accuracy: 0.8067

Visualisation

[7] :
```
cm = confusion_matrix(y_test, y_pred)
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=['Negative',
    'Positive'], yticklabels=['Negative', 'Positive'])
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')
```

```
plt.show()
```



Result:

The outputs were verified successfully. SVC can work on non-vectorial data.

SVR on non-linearly separable data

March 12th, 2025

AIM: Performing SVR on non linearly separable 2d data

Algorithm:

1. Import Required Libraries
2. Load numpy, matplotlib.pyplot, SVR from sklearn.svm, and other necessary modules. Generate Non-Linear Data
3. Create synthetic data using a non-linear function (e.g., sine wave with noise). Store the feature matrix X and target variable y.
4. Split the data into training and testing sets. Initialize the SVR Model
5. Choose the kernel function (e.g., rbf for non-linearity).
6. Set hyperparameters like C (regularization), gamma (kernel coefficient), and epsilon (margin of tolerance).
7. Train the SVR Model
8. Fit the model using training data train , train X train ,y train . Make Predictions.
9. Use the trained model to predict on the test data test X test Evaluate Model Performance
10. Compute the Mean Squared Error (MSE) to measure how well the model fits. Visualize the SVR Fit
11. Generate predictions over a smooth range of values
12. Plot original data points and SVR regression curve to observe the fit

MODEL DESCRIPTION:

Support Vector Regression is an extension of Support Vector Machine (SVM) designed for regression tasks

The goal is to predict continuous outcomes rather than discrete class labels. Like SVM, SVR seeks to find a function that approximates the relationship between input features and a continuous target variable by leveraging the concept of support vectors—data points that lie closest to the regression function and effectively "support" its position

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.svm import SVR
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
```

```
# Generate non-linear data (e.g., a sine wave with noise)
np.random.seed(42)
X = np.sort(5 * np.random.rand(100, 1), axis=0)  # Feature (independent variable)
y = np.sin(X).ravel() + np.random.normal(0, 0.1, X.shape[0])  # Target (dependent variable)

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

[5]:
```
# Initialize SVR with RBF kernel for non-linearity
svr_rbf = SVR(kernel='rbf', C=100, gamma=0.1, epsilon=0.1)

# Train the model
svr_rbf.fit(X_train, y_train)
```

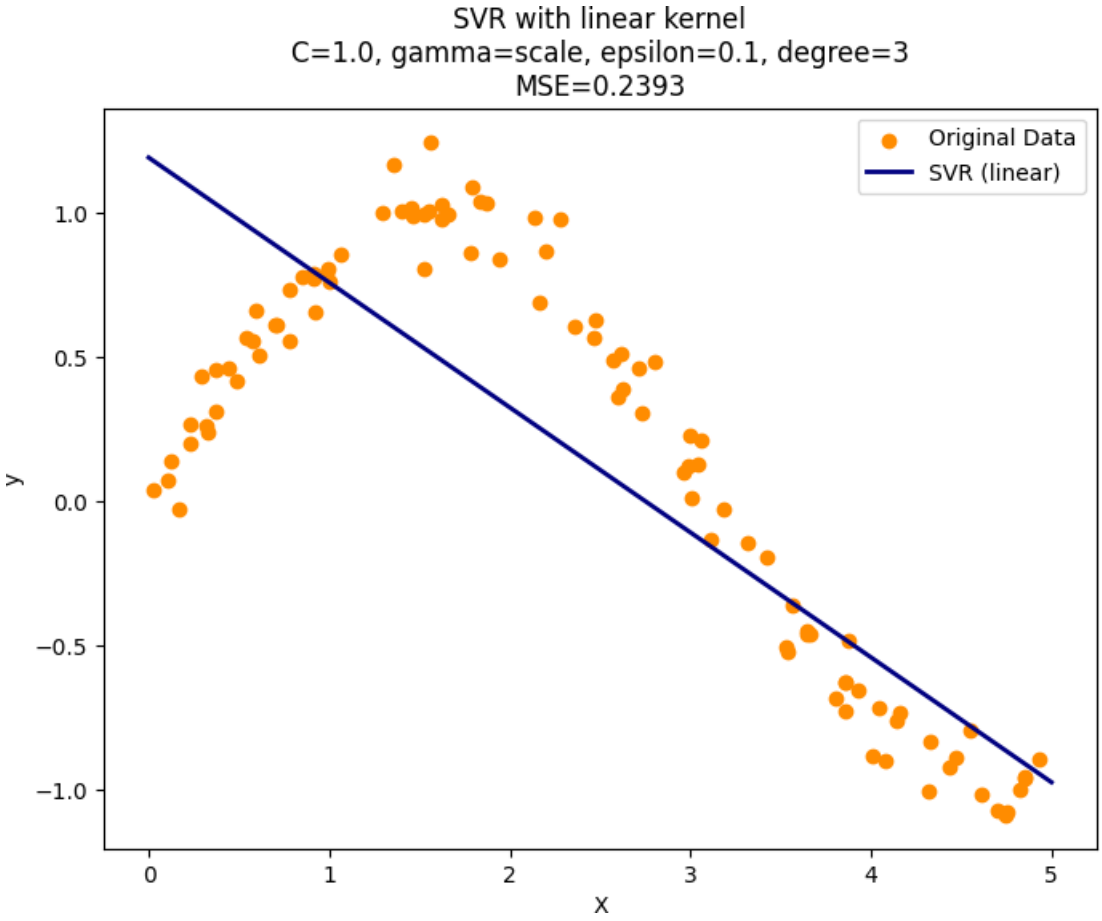
[5]: SVR(C=100, gamma=0.1)

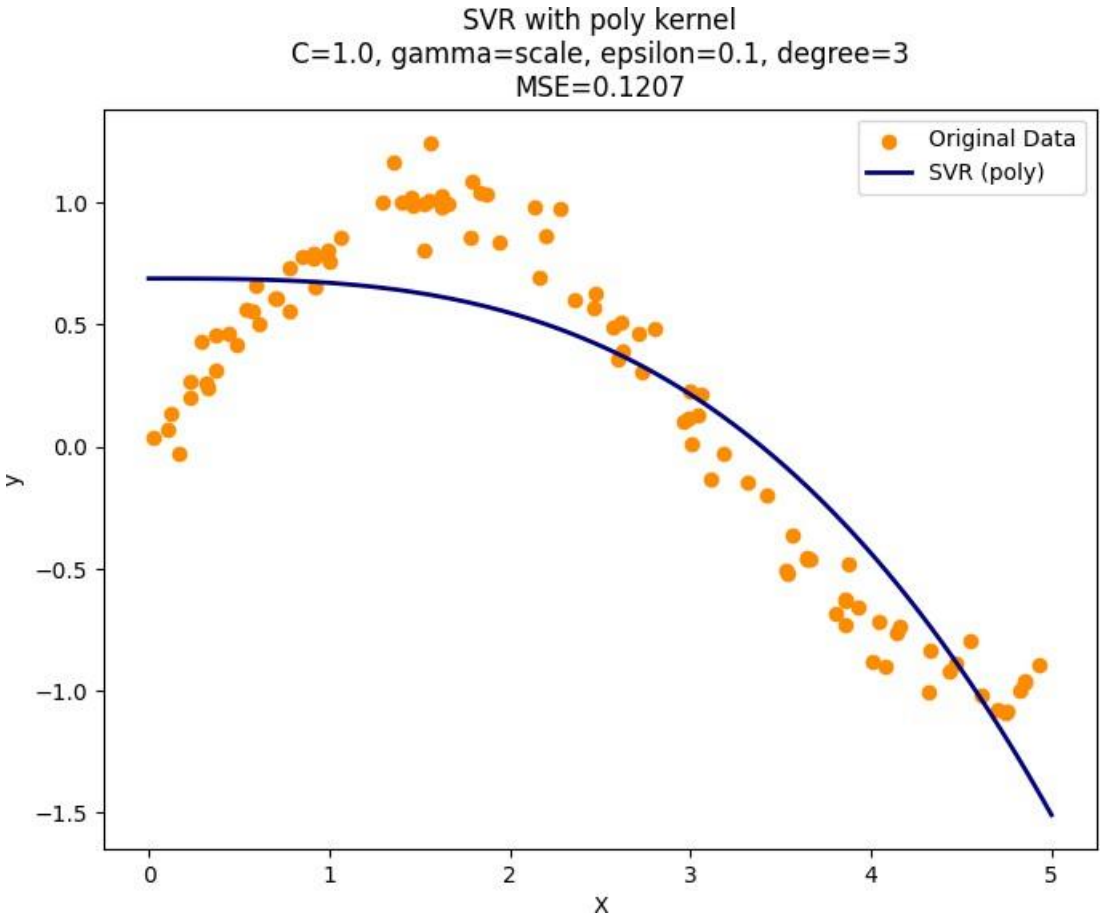
[8] :
```
def plot_svr(kernel, C=1.0, gamma='scale', epsilon=0.1, degree=3):
    """
    Trains an SVR model with specified hyperparameters and plots its predictions.

    Parameters:
        kernel: Kernel type ('linear', 'poly', 'rbf', 'sigmoid')
        C: Regularization parameter
        gamma: Kernel coefficient (used for RBF, poly, sigmoid)
        epsilon: Margin of tolerance for SVR
        degree: Degree of polynomial (used for poly kernel)
```

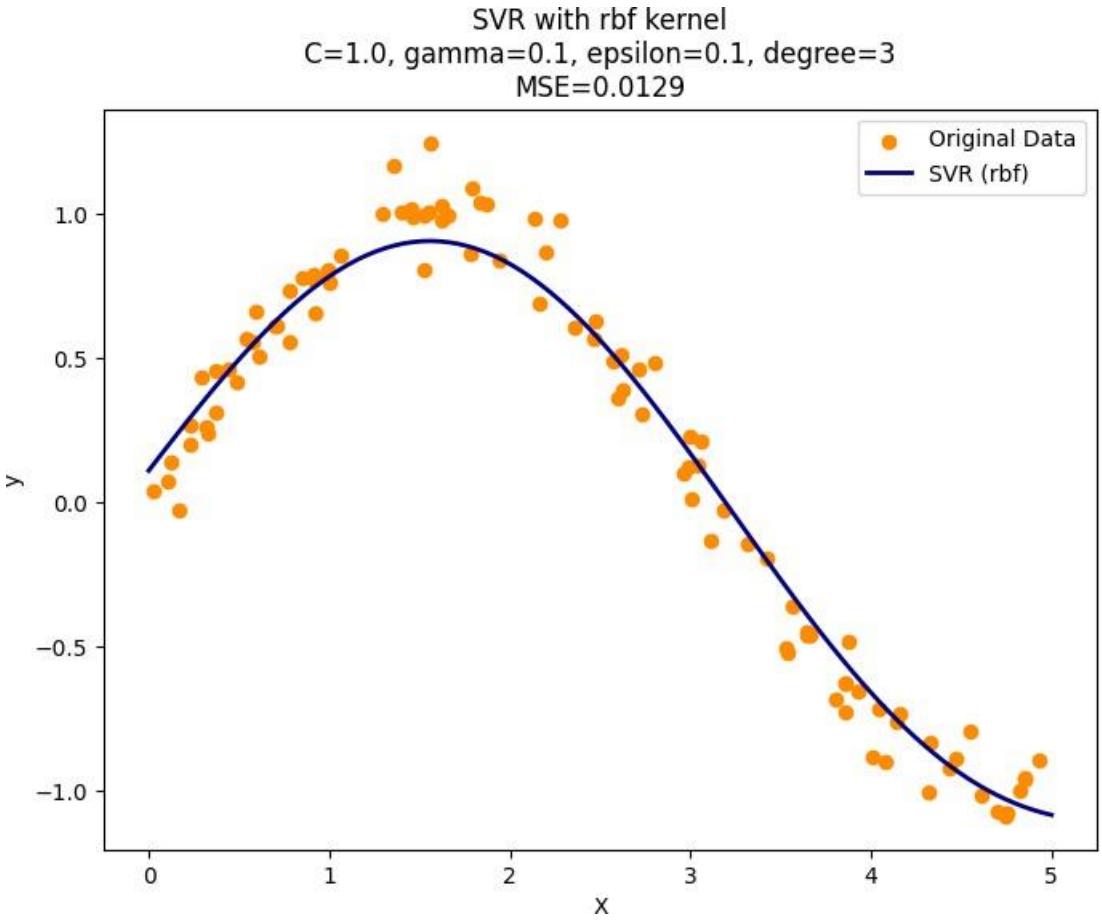
```
    """
    # Train SVR model
    svr = SVR(kernel=kernel, C=C, gamma=gamma, epsilon=epsilon, degree=degree)
    svr.fit(X_train, y_train)

    # Predict on test and smooth range for visualization
    X_smooth = np.linspace(0, 5, 100).reshape(-1, 1)  # Smooth curve
    y_smooth = svr.predict(X_smooth)
    y_pred = svr.predict(X_test)

    # Compute MSE
    mse = mean_squared_error(y_test, y_pred)

    # Plot original data and SVR predictions
    plt.figure(figsize=(8, 6))
    plt.scatter(X, y, color='darkorange', label="Original Data")  # Data points
    plt.plot(X_smooth, y_smooth, color='navy', lw=2, label=f"SVR ({kernel})")
    # SVR fit
    plt.title(f"SVR with {kernel} kernel\nC={C}, gamma={gamma},
    epsilon={epsilon}, degree={degree}\nMSE={mse:.4f}")
    plt.xlabel("X")
    plt.ylabel("y")
    plt.legend()
    plt.show()
```

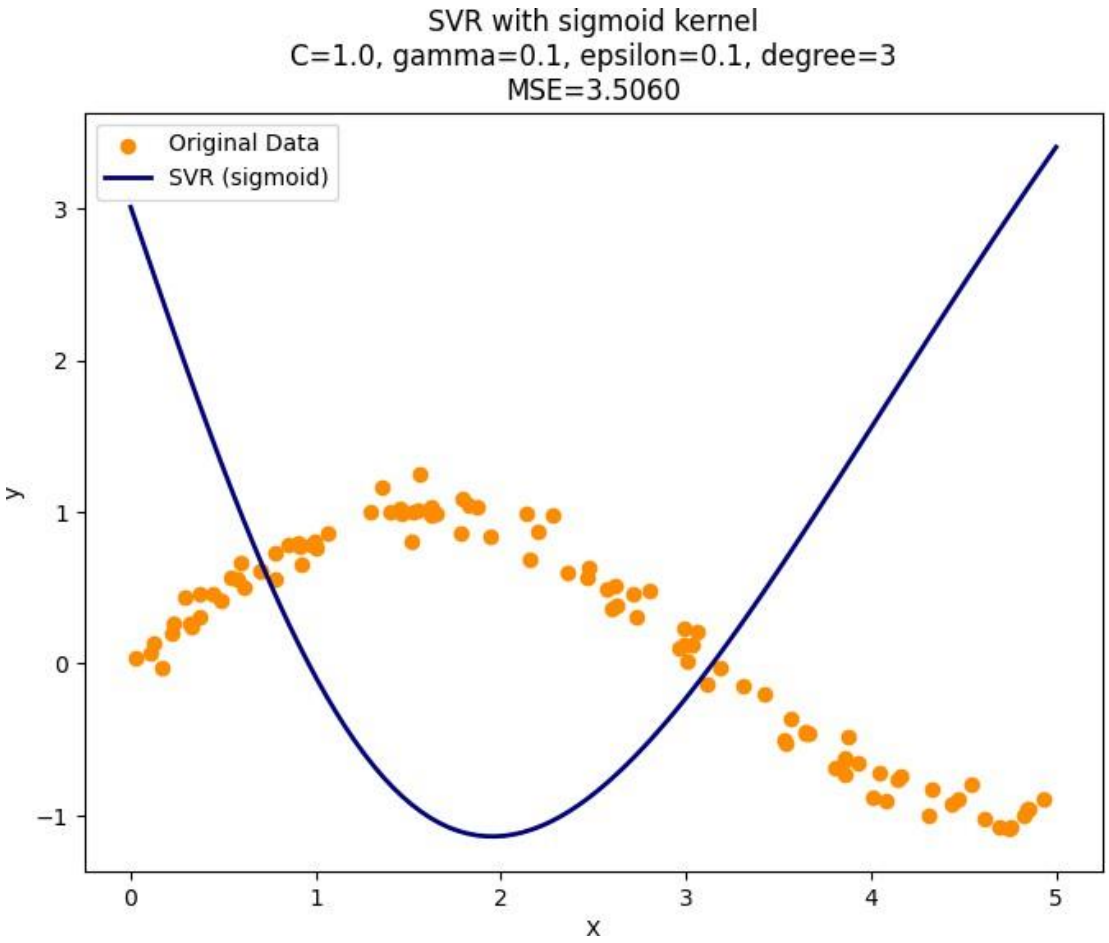
[9] :

```
# Linear Kernel (Good for linearly separable data)
plot_svr(kernel='linear', C=1.0)

# Polynomial Kernel (Can model complex relationships)
plot_svr(kernel='poly', C=1.0, degree=3)

# RBF Kernel (Best for general non-linear relationships)
plot_svr(kernel='rbf', C=1.0, gamma=0.1)

# Sigmoid Kernel (Less common but useful for some cases)
plot_svr(kernel='sigmoid', C=1.0, gamma=0.1)
```

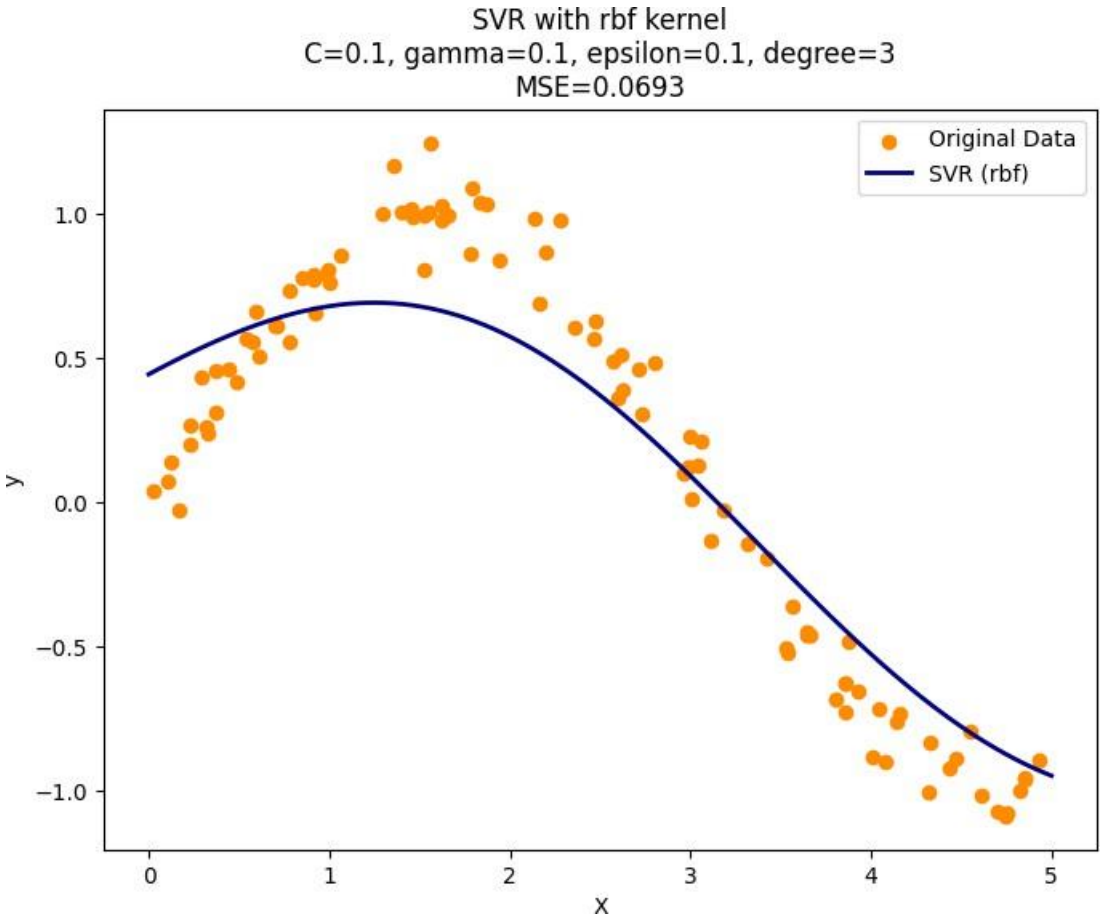

SVR with linear kernel
C=1.0, gamma=scale, epsilon=0.1, degree=3
MSE=0.2393
Original Data
SVR (linear)
1.0
0.5
0.0
−0.5
−1.0
y
0
1
2
3
4
5
X

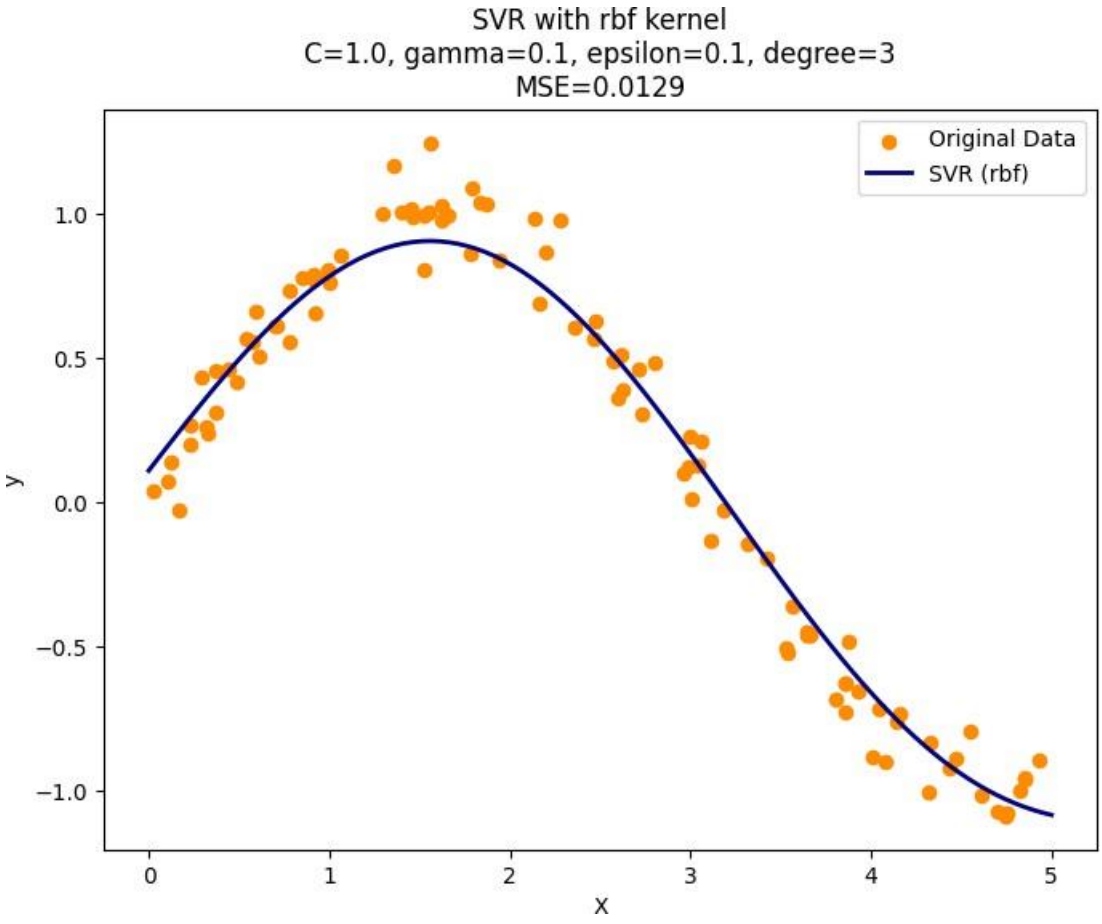

SVR with poly kernel
C=1.0, gamma=scale, epsilon=0.1, degree=3
MSE=0.1207
Original Data
SVR (poly)
1.0
0.5
0.0
−0.5
−1.0
−1.5
y
0
1
2
3
4
5
X

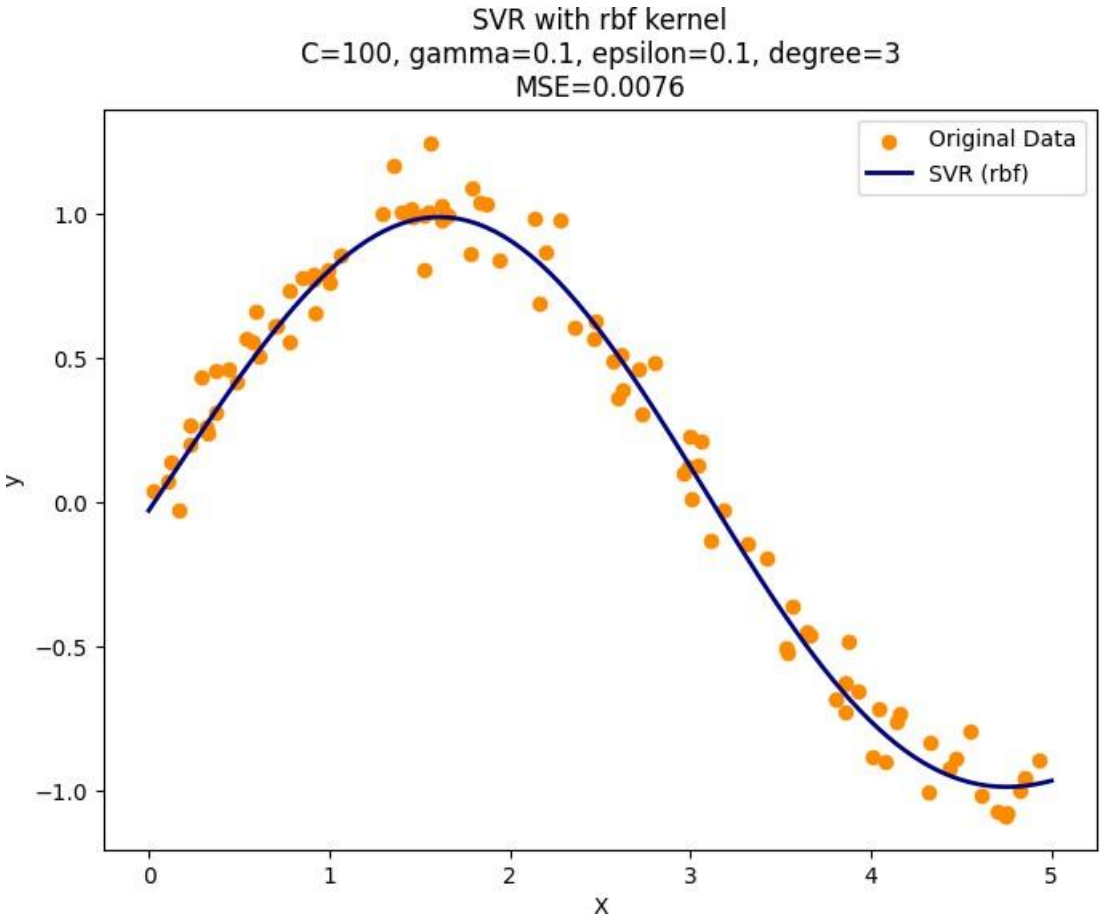

SVR with rbf kernel
C=1.0, gamma=0.1, epsilon=0.1, degree=3
MSE=0.0129
Original Data
SVR (rbf)
1.0
0.5
0.0
−0.5
−1.0
y
0
1
2
3
4
5
X



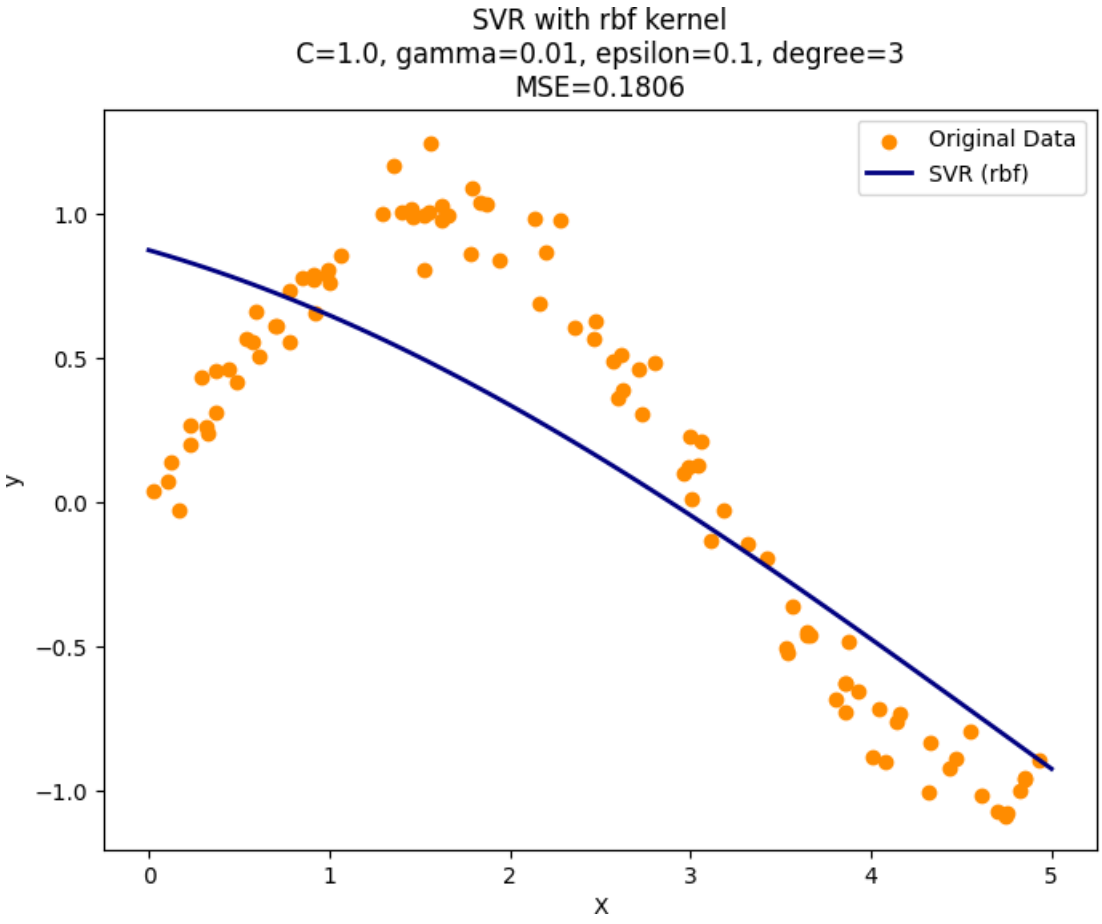
[10]:
```
# Low Regularization (High Bias)
plot_svr(kernel='rbf', C=0.1, gamma=0.1)

# Medium Regularization
plot_svr(kernel='rbf', C=1.0, gamma=0.1)

# High Regularization (Can lead to overfitting)
plot_svr(kernel='rbf', C=100, gamma=0.1)
```

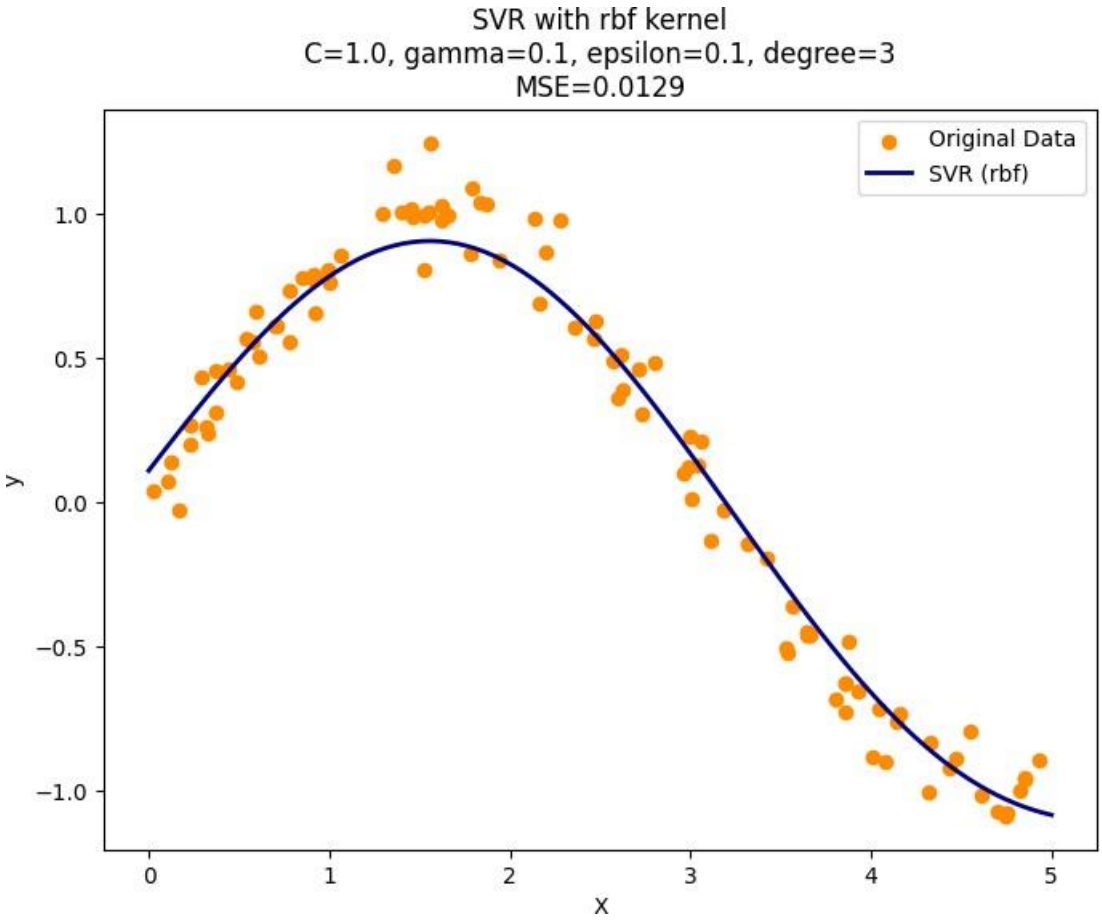

SVR with rbf kernel
C=0.1, gamma=0.1, epsilon=0.1, degree=3
MSE=0.0693
Original Data
SVR (rbf)
1.0
0.5
0.0
−0.5
−1.0
y
0
1
2
3
4
5
X

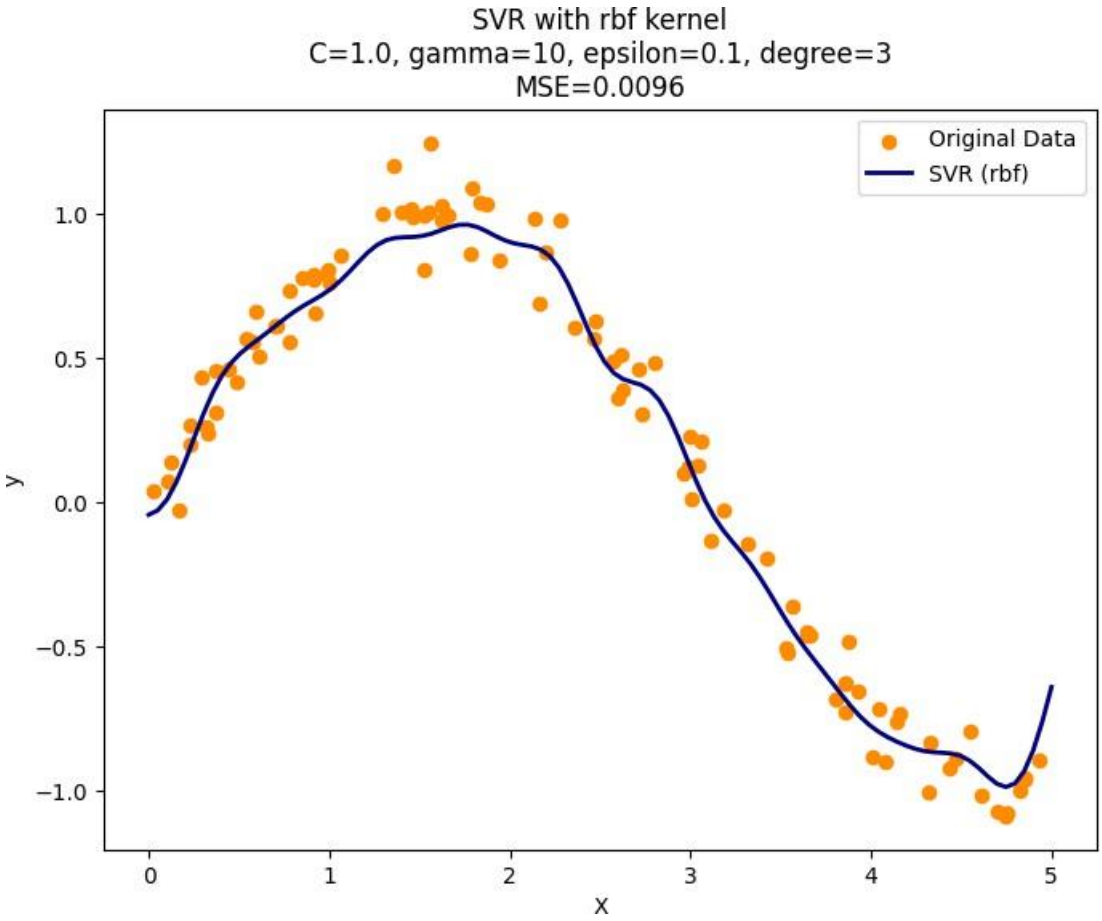

SVR with rbf kernel
C=1.0, gamma=0.1, epsilon=0.1, degree=3
MSE=0.0129
Original Data
SVR (rbf)
1.0
0.5
0.0
−0.5
−1.0
y
0
1
2
3
4
5
X



[11] :
```
# Small gamma (Smooth curve, underfitting)
plot_svr(kernel='rbf', C=1.0, gamma=0.01)

# Medium gamma (Balanced fit)
plot_svr(kernel='rbf', C=1.0, gamma=0.1)

# High gamma (Overfitting, very flexible)
plot_svr(kernel='rbf', C=1.0, gamma=10)
```


SVR with rbf kernel
C=1.0, gamma=0.01, epsilon=0.1, degree=3
MSE=0.1806
Original Data
SVR (rbf)
1.0
0.5
0.0
−0.5
−1.0
y
0
1
2
3
4
5
X


SVR with rbf kernel
C=1.0, gamma=0.1, epsilon=0.1, degree=3
MSE=0.0129
Original Data
SVR (rbf)
1.0
0.5
0.0
−0.5
−1.0
y
0
1
2
3
4
5
X



Result:

Insights from SVR Exploration 1 Effect of Kernel Choice Linear Kernel: Works well if data is almost linear, but fails for complex relationships.

Polynomial Kernel: Can fit more complex data but requires tuning degree.

RBF Kernel: Works best for general non-linear patterns.

Sigmoid Kernel: Less common, can sometimes behave like RBF.

2 Effect of C (Regularization) Small C → Simpler model, higher bias, avoids overfitting.

Large C → Tries to fit data more precisely, risks overfitting.

3 Effect of Gamma (for RBF & Poly) Small gamma → Broad, smooth function, underfits.

Large gamma → More flexible, but risks overfitting.

4 Effect of Epsilon (Tolerance) Small epsilon → More precise fit, sensitive to noise.

Large epsilon → Smoother, ignores small variations.

Single layered Perceptron

March 19th, 2025

AIM: Perform single layer perceptron on lineaerly separable and non-linearly separable data

ALGORITHM:

Step 1: Import Necessary Libraries Import numpy, matplotlib.pyplot, sklearn.model_selection, sklearn.linear_model, and sklearn.metrics for implementation and evaluation.

Step 2: Generate Linearly and Non-Linearly Separable Data Create linearly separable datasets (e.g., AND, OR) and non-linearly separable datasets (e.g., XOR, NOR)

Step 3: Train Perceptron Using Sklearn Use Perceptron() from sklearn.linear_model.

Fit the model on training data.

Evaluate performance using accuracy metrics.

Step 4: Perform Hyperparameter Tuning Using GridSearchCV Define a hyperparameter grid for learning rate, max iterations, etc.

Use GridSearchCV to find optimal parameters.

Step 5: Implement Custom Perceptron (Perceptron1) Manually implement weight update rule

Train the model using batch updates for multiple epochs.

Fit on linearly separable data and verify convergence.

Compare with MLP

DESCRIPTION:

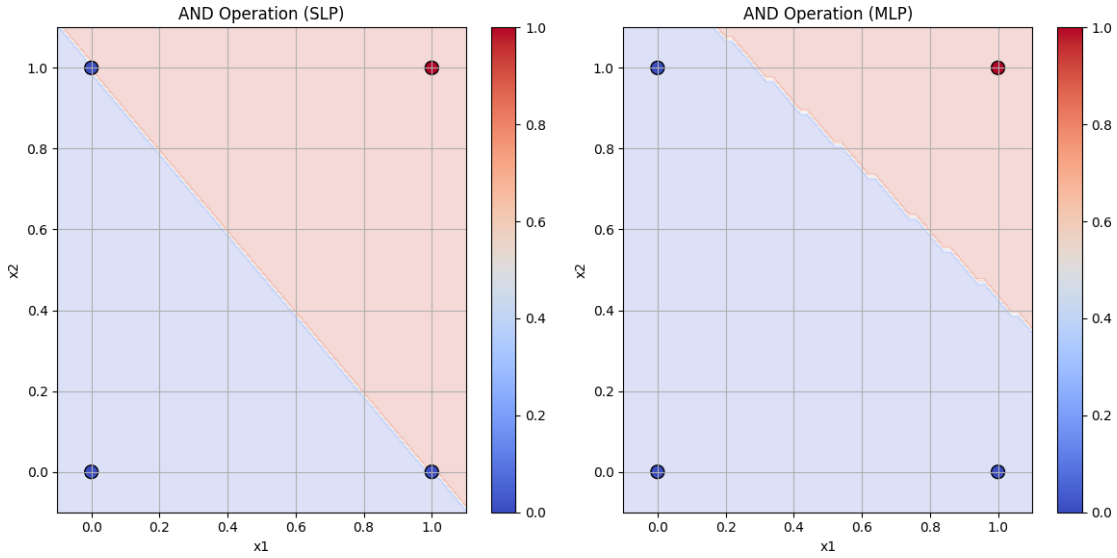
A single-layer perceptron is a basic neural network with just one layer of neurons. It takes input values, applies weights, and passes the result through an activation function to produce an output. This model is used for binary classification when the data is linearly separable. It learns by adjusting weights to minimize classification errors. While simple, it cannot handle problems that require non-linear boundaries.

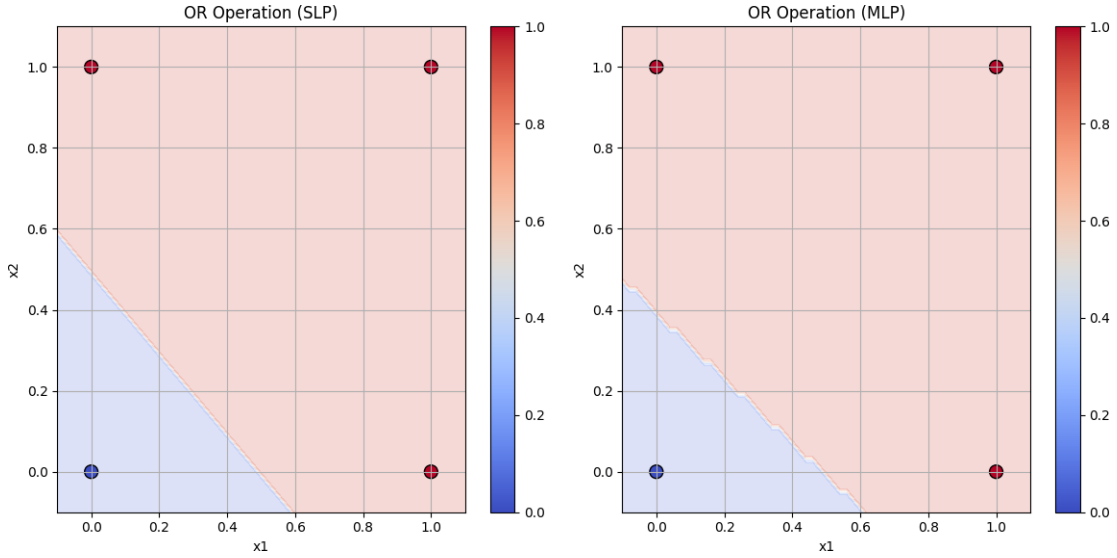
```
[ ]: import pandas as pd
     import numpy as np
```

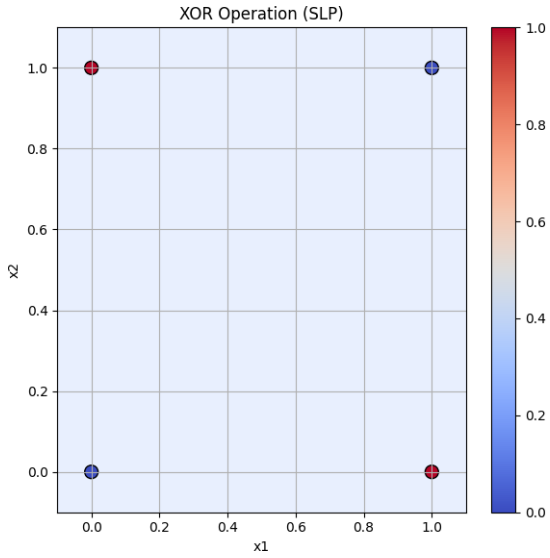
[ ]:
```
import matplotlib.pyplot as plt
from sklearn.linear_model import Perceptron
from sklearn.neural_network import MLPClassifier
```

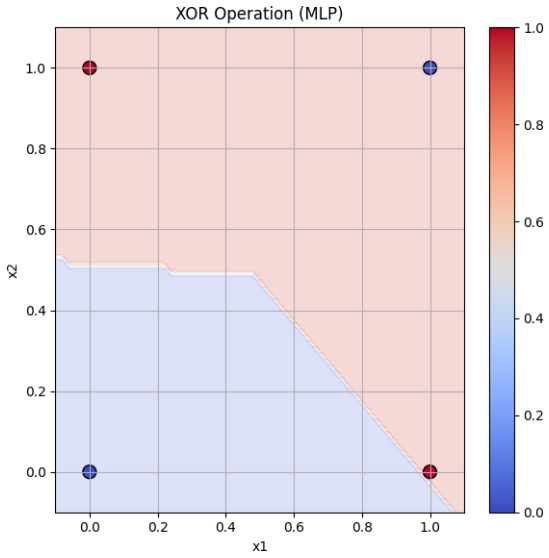
[ ]:
```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.linear_model import Perceptron
from sklearn.neural_network import MLPClassifier

# Boolean values for x1 and x2
x1 = [0, 0, 1, 1]
x2 = [0, 1, 0, 1]
X = np.array([[x1[i], x2[i]] for i in range(4)])

# Define the logical operations
operations = {
    'AND': lambda x1, x2: x1 & x2,
    'OR': lambda x1, x2: x1 | x2,
    'XOR': lambda x1, x2: x1 ^ x2,
    'NAND': lambda x1, x2: not (x1 & x2),
    'NOR': lambda x1, x2: not (x1 | x2),
    'XNOR': lambda x1, x2: not (x1 ^ x2),
}

 # Iterate through each operation and plot
for op_name, op_func in operations.items():
    # Compute the result of the operation
    y = [op_func(a, b) for a, b in zip(x1, x2)]
    y = np.array(y)

    # --- Single-layer Perceptron ---
    perceptron = Perceptron(max_iter=1000)
    perceptron.fit(X, y)

    # --- Multi-layer Perceptron ---
    mlp = MLPClassifier(hidden_layer_sizes=(5,), max_iter=1000)
    mlp.fit(X, y)

    # Create a meshgrid for plotting the decision boundary
    h = .02  # Step size in the mesh
    x_min, x_max = -0.1, 1.1
    y_min, y_max = -0.1, 1.1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))

    # Predict for each point in the meshgrid for both models
    Z_perceptron = perceptron.predict(np.c_[xx.ravel(), yy.ravel()])
```

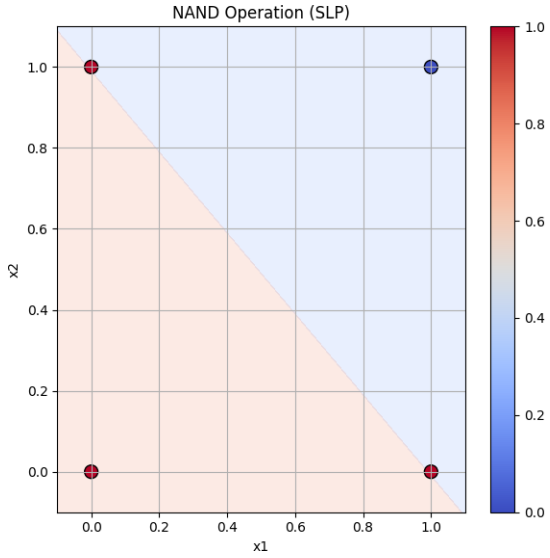
```
    Z_mlp = mlp.predict(np.c_[xx.ravel(), yy.ravel()])
    Z_perceptron = Z_perceptron.reshape(xx.shape)
    Z_mlp = Z_mlp.reshape(xx.shape)

    # Create the plot
    plt.figure(figsize=(12, 6))

    # Plot decision boundary for SLP (Perceptron)
    plt.subplot(1, 2, 1)
    plt.contourf(xx, yy, Z_perceptron, alpha=0.2, cmap='coolwarm')
    plt.scatter(x1, x2, c=y, s=100, cmap='coolwarm', edgecolors='k',␣
␣label='Data Points')
    plt.title(f'{op_name} Operation (SLP)')
    plt.xlabel('x1')
    plt.ylabel('x2')
    plt.colorbar()
    plt.grid(True)

    # Plot decision boundary for MLP
    plt.subplot(1, 2, 2)
    plt.contourf(xx, yy, Z_mlp, alpha=0.2, cmap='coolwarm')
    plt.scatter(x1, x2, c=y, s=100, cmap='coolwarm', edgecolors='k',␣
␣label='Data Points')
    plt.title(f'{op_name} Operation (MLP)')
    plt.xlabel('x1')
    plt.ylabel('x2')
    plt.colorbar()
    plt.grid(True)

    # Show the plot for both models
    plt.tight_layout()
    plt.show()
```

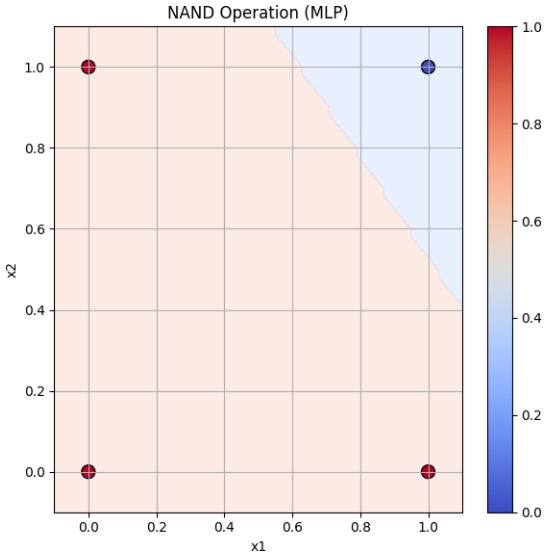

AND Operation (SLP)
AND Operation (MLP)
x1
x2
0.0
0.2
0.4
0.6
0.8
1.0

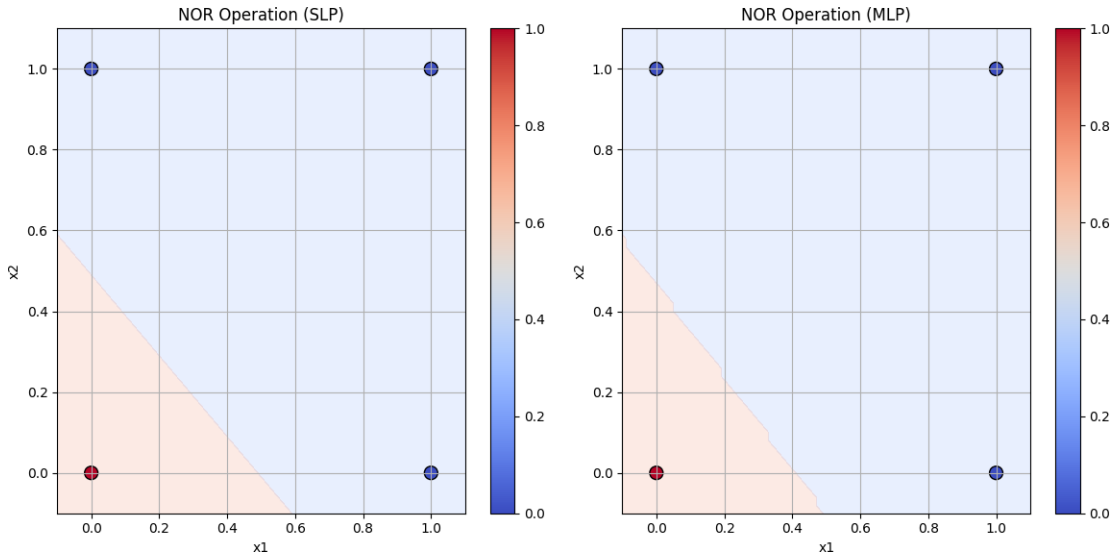

OR Operation (SLP)
OR Operation (MLP)
x1
x2
0.0
0.2
0.4
0.6
0.8
1.0

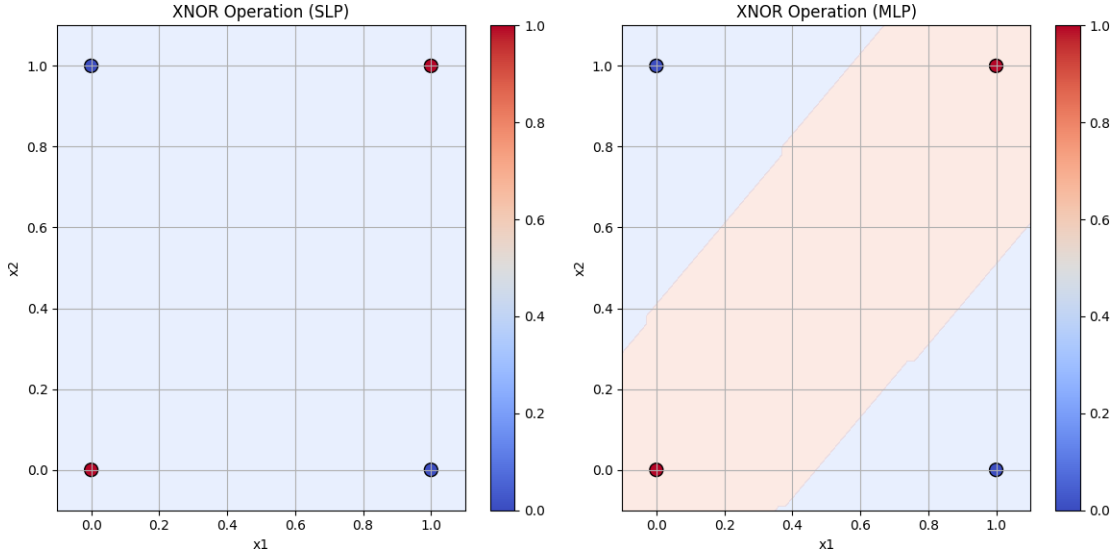

XOR Operation (SLP)
x1
x2


XOR Operation (MLP)
x1
x2


NAND Operation (SLP)
x1
x2


NAND Operation (MLP)
x1
x2





[ ]:
```
from sklearn.model_selection import GridSearchCV
```

```
# Define the hyperparameter grid with different activation functions
param_grid = {
    'eta0': [0.001, 0.01, 0.1, 1],
    'max_iter': [100, 1000, 5000],
    'alpha': [0.0001, 0.001, 0.01],
    'tol': [1e-4, 1e-3, 1e-2],
}

# Initialize the Perceptron model
perceptron = Perceptron()

# Set up GridSearchCV to search over the hyperparameter grid
grid_search = GridSearchCV(estimator=perceptron, param_grid=param_grid, cv=2)

# Fit the grid search to the training data
grid_search.fit(X,y)

# Output the best hyperparameters found
print("Best hyperparameters:", grid_search.best_params_)
```

```
Best hyperparameters: {'alpha': 0.0001, 'eta0': 0.001, 'max_iter': 100, 'tol': 0.0001}
```

[ ]:
```
thclass Perceptron1:
    def __init__(self, input_size, learning_rate=0.1, epochs=1000):
        self.learning_rate = learning_rate
        self.epochs = epochs
        self.weights = np.zeros(input_size + 1)  # +1 for the bias term

    # Activation function (step function)
    def activation(self, x):
        return 1 if x >= 0 else 0


    # Prediction function
    def predict(self, X):
        # Add bias term (1 for each input)
        X_with_bias = np.c_[np.ones(X.shape[0]), X]  # Adding bias column
        linear_output = np.dot(X_with_bias, self.weights)
        return np.array([self.activation(x) for x in linear_output])

    # Training the perceptron
    def train(self, X, y):
        # Add bias term to input data
        X_with_bias = np.c_[np.ones(X.shape[0]), X]  # Adding bias column
        for epoch in range(self.epochs):
            for i in range(len(X)):
```

```
                # Prediction for the current input
                prediction = self.activation(np.dot(X_with_bias[i], self.
weights))

                # Update weights based on the error
                error = y[i] - prediction
                self.weights += self.learning_rate * error * X_with_bias[i]

            # Optionally print the error or weights at each epoch
            if epoch % 100 == 0:
                predictions = self.predict(X)
                accuracy = np.mean(predictions == y) * 100
                print(f'Epoch {epoch}: Accuracy = {accuracy:.2f}%')
```

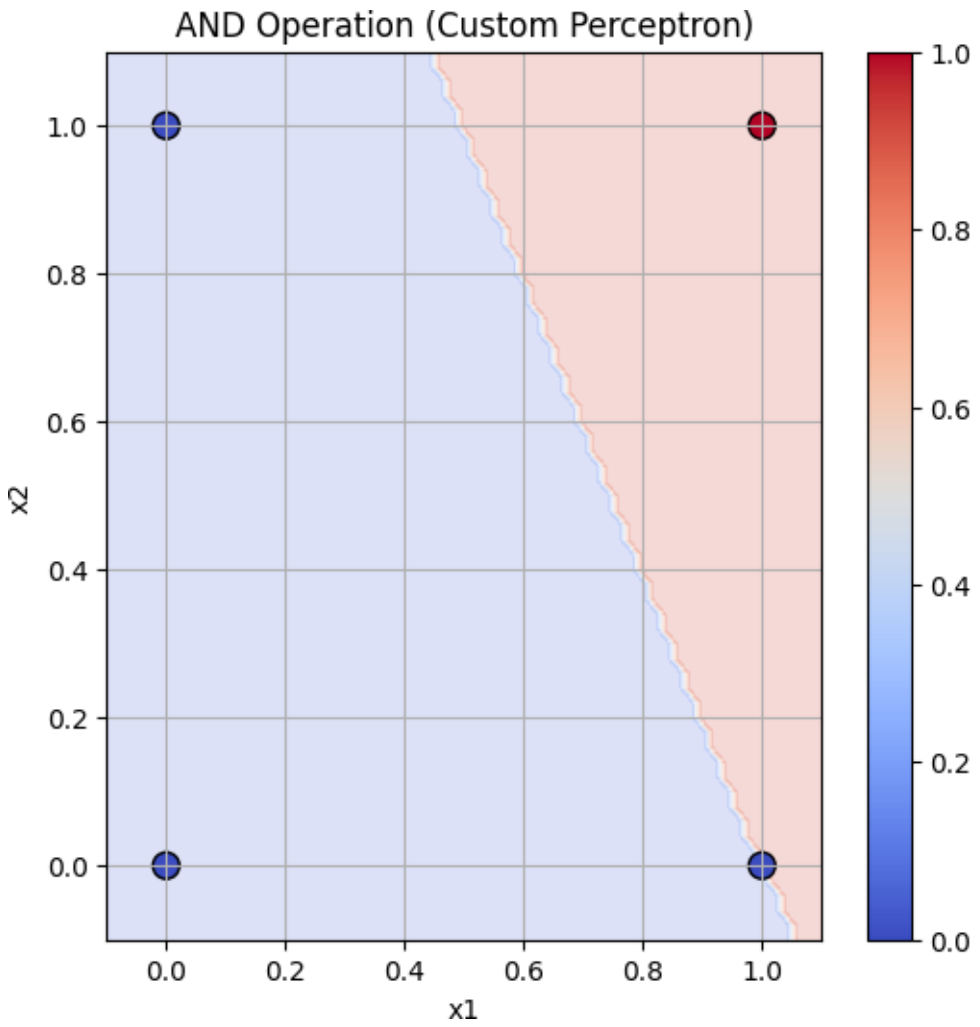
[ ]:
```
class MLP1:
    def __init__(self, input_size, hidden_size, output_size, learning_rate=0.1,
epochs=10000):
        # Initialize weights and biases
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.learning_rate = learning_rate
        self.epochs = epochs

        # Weights and biases for input to hidden layer
        self.W1 = np.random.randn(input_size, hidden_size) * 0.1
        self.b1 = np.zeros(hidden_size)

        # Weights and biases for hidden to output layer
        self.W2 = np.random.randn(hidden_size, output_size) * 0.1
        self.b2 = np.zeros(output_size)

    # Activation function: Sigmoid
    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

    # Derivative of the Sigmoid function
    def sigmoid_derivative(self, x):
        return x * (1 - x)

    # Forward propagation
    def forward(self, X):
        self.z1 = np.dot(X, self.W1) + self.b1  # Input to hidden layer
        self.a1 = self.sigmoid(self.z1)          # Hidden layer activation
        self.z2 = np.dot(self.a1, self.W2) + self.b2 # Hidden to output layer
        self.a2 = self.sigmoid(self.z2)              # Output layer activation
        return self.a2
```

```
    # Backward propagation
    def backward(self, X, y):
        # Output layer error
        self.error = y - self.a2
        self.dz2 = self.error * self.sigmoid_derivative(self.a2)

        # Hidden layer error
        self.dz1 = self.dz2.dot(self.W2.T) * self.sigmoid_derivative(self.a1)

        # Update weights and biases
        self.W2 += self.a1.T.dot(self.dz2) * self.learning_rate
        self.b2 += np.sum(self.dz2, axis=0) * self.learning_rate
        self.W1 += X.T.dot(self.dz1) * self.learning_rate
        self.b1 += np.sum(self.dz1, axis=0) * self.learning_rate

    # Training the MLP
    def train(self, X, y):
        for epoch in range(self.epochs):
            # Forward propagation
            self.forward(X)

            # Backward propagation
            self.backward(X, y)

            # Optionally print loss or accuracy
            if epoch % 1000 == 0:
                loss = np.mean(np.square(y - self.a2))     # Mean squared error
                print(f'Epoch {epoch}, Loss: {loss:.4f}')
```

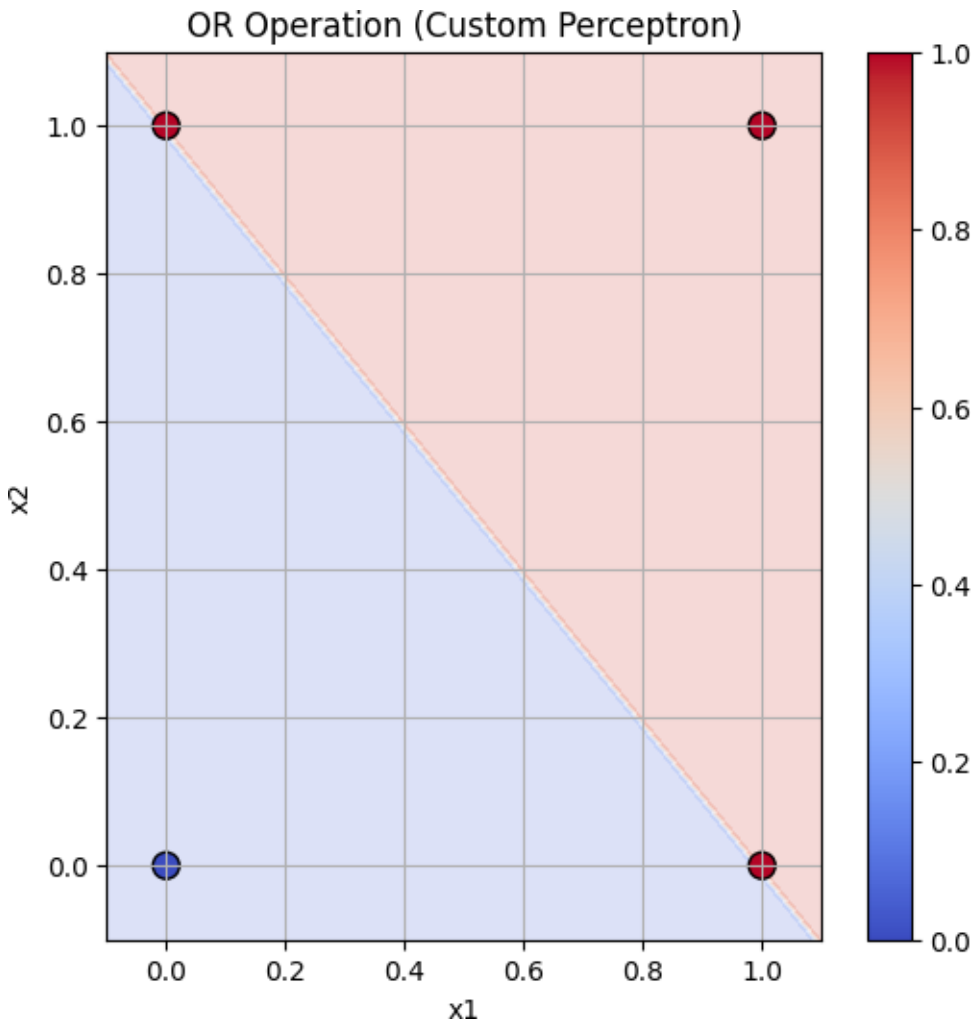
[ ]:
```
for op_name, op_func in operations.items():
    # Compute the result of the operation
    y = np.array([op_func(a, b) for a, b in zip(x1, x2)])

    # Create the Perceptron model
    perceptron = Perceptron1(input_size=2, learning_rate=0.1, epochs=1000)
    perceptron.train(X, y)

    # Make predictions and plot decision boundary
    predictions = perceptron.predict(X)

    # Create a meshgrid for plotting decision boundary
    h = .02  # Step size in the mesh
    x_min, x_max = -0.1, 1.1
    y_min, y_max = -0.1, 1.1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))
```

```
    # Predict for every point in the meshgrid
    Z = perceptron.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    # Plotting
    plt.figure(figsize=(6, 6))
    plt.contourf(xx, yy, Z, alpha=0.2, cmap='coolwarm')
    plt.scatter(x1, x2, c=y, s=100, cmap='coolwarm', edgecolors='k',␣
label='Data Points')
    plt.title(f'{op_name} Operation (Custom Perceptron)')
    plt.xlabel('x1')
    plt.ylabel('x2')
    plt.colorbar()
    plt.grid(True)
    plt.show()
```

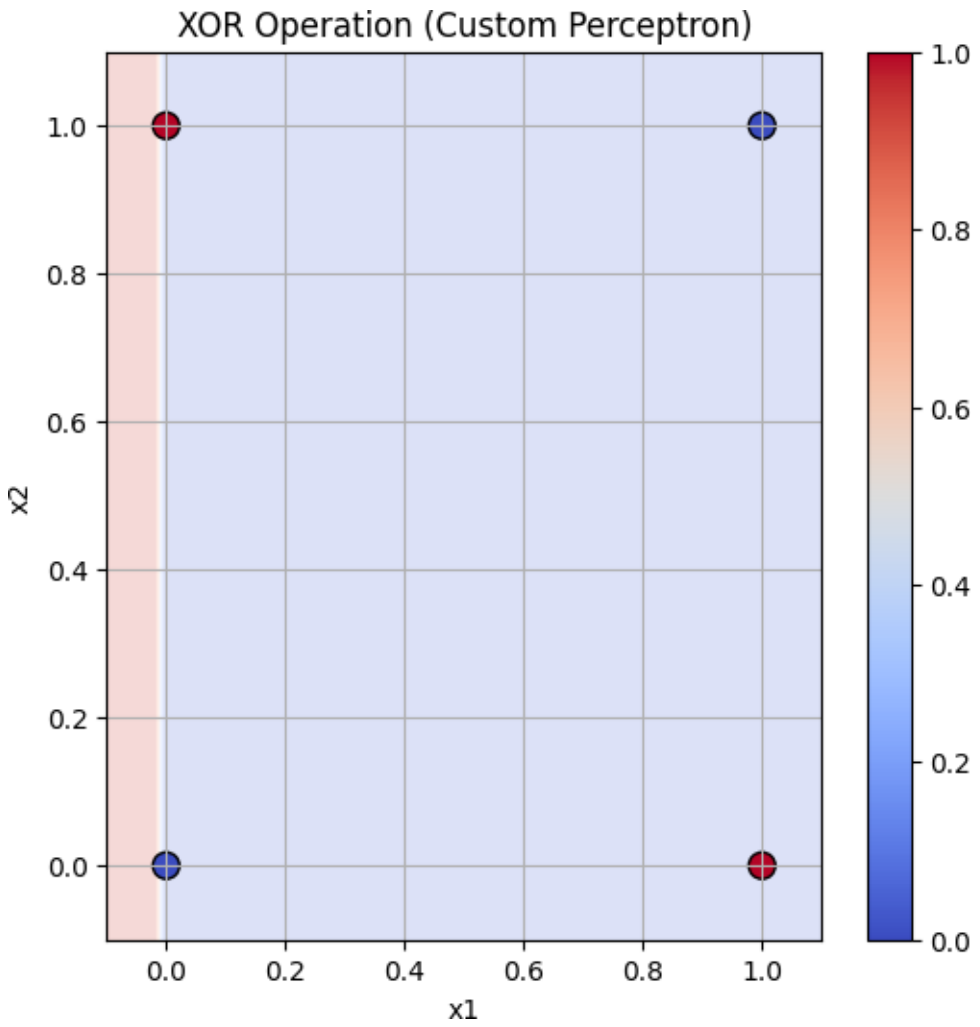
```
Epoch 0: Accuracy = 25.00%
Epoch 100: Accuracy = 100.00%
Epoch 200: Accuracy = 100.00%
Epoch 300: Accuracy = 100.00%
Epoch 400: Accuracy = 100.00%
Epoch 500: Accuracy = 100.00%
Epoch 600: Accuracy = 100.00%
Epoch 700: Accuracy = 100.00%
Epoch 800: Accuracy = 100.00%
Epoch 900: Accuracy = 100.00%
```



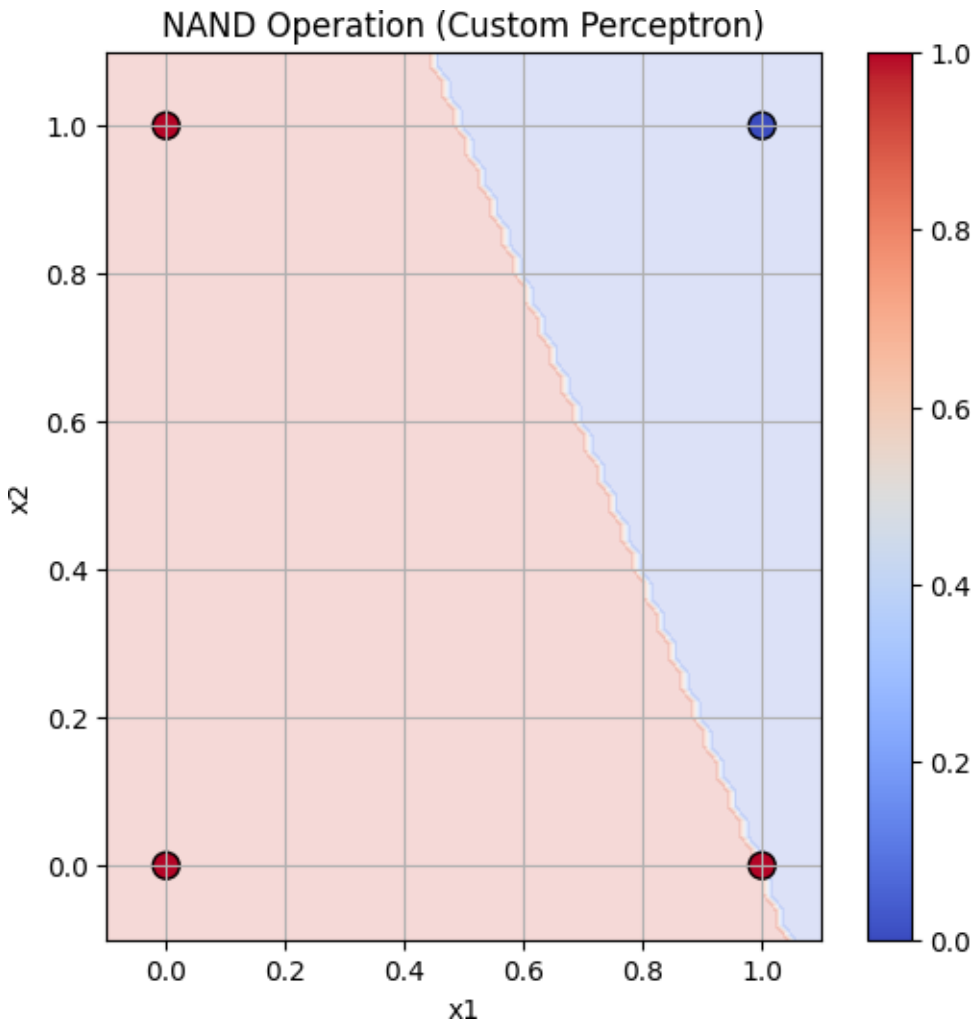

```
Epoch 0: Accuracy = 75.00%
Epoch 100: Accuracy = 100.00%
Epoch 200: Accuracy = 100.00%
Epoch 300: Accuracy = 100.00%
Epoch 400: Accuracy = 100.00%
Epoch 500: Accuracy = 100.00%
Epoch 600: Accuracy = 100.00%
Epoch 700: Accuracy = 100.00%
Epoch 800: Accuracy = 100.00%
Epoch 900: Accuracy = 100.00%
```



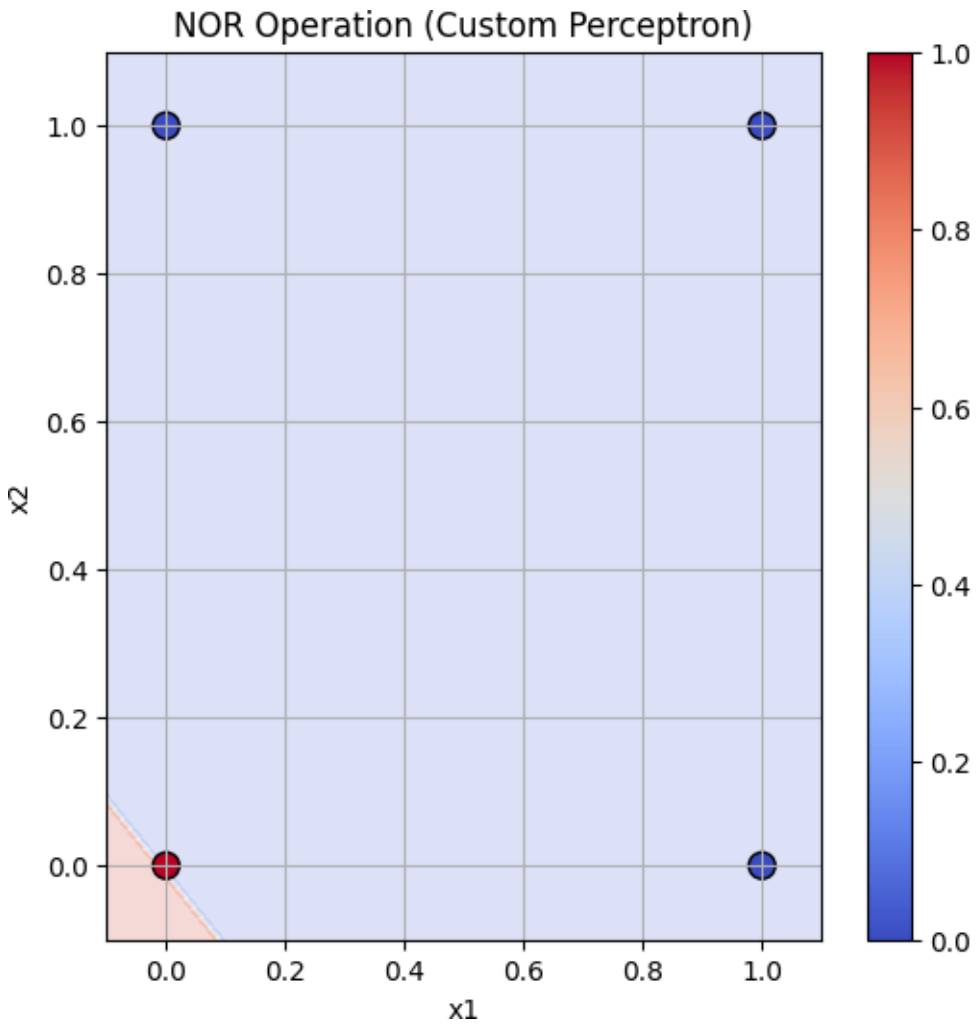
Epoch 0: Accuracy = 50.00%
Epoch 100: Accuracy = 50.00%
Epoch 200: Accuracy = 50.00%
Epoch 300: Accuracy = 50.00%
Epoch 400: Accuracy = 50.00%
Epoch 500: Accuracy = 50.00%
Epoch 600: Accuracy = 50.00%
Epoch 700: Accuracy = 50.00%
Epoch 800: Accuracy = 50.00%
Epoch 900: Accuracy = 50.00%



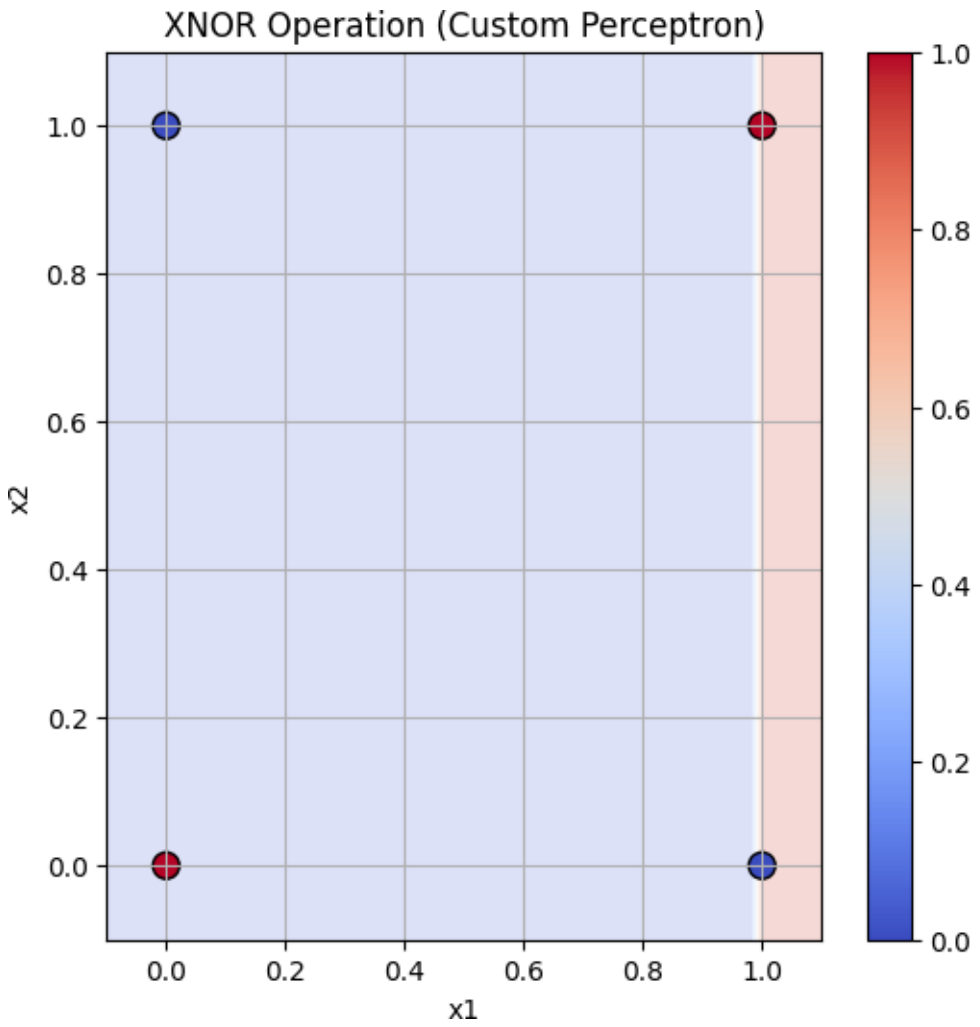

```
Epoch 0: Accuracy = 25.00%
Epoch 100: Accuracy = 100.00%
Epoch 200: Accuracy = 100.00%
Epoch 300: Accuracy = 100.00%
Epoch 400: Accuracy = 100.00%
Epoch 500: Accuracy = 100.00%
Epoch 600: Accuracy = 100.00%
Epoch 700: Accuracy = 100.00%
Epoch 800: Accuracy = 100.00%
Epoch 900: Accuracy = 100.00%
```




```
Epoch 0: Accuracy = 75.00%
Epoch 100: Accuracy = 100.00%
Epoch 200: Accuracy = 100.00%
Epoch 300: Accuracy = 100.00%
Epoch 400: Accuracy = 100.00%
Epoch 500: Accuracy = 100.00%
Epoch 600: Accuracy = 100.00%
Epoch 700: Accuracy = 100.00%
Epoch 800: Accuracy = 100.00%
Epoch 900: Accuracy = 100.00%
```



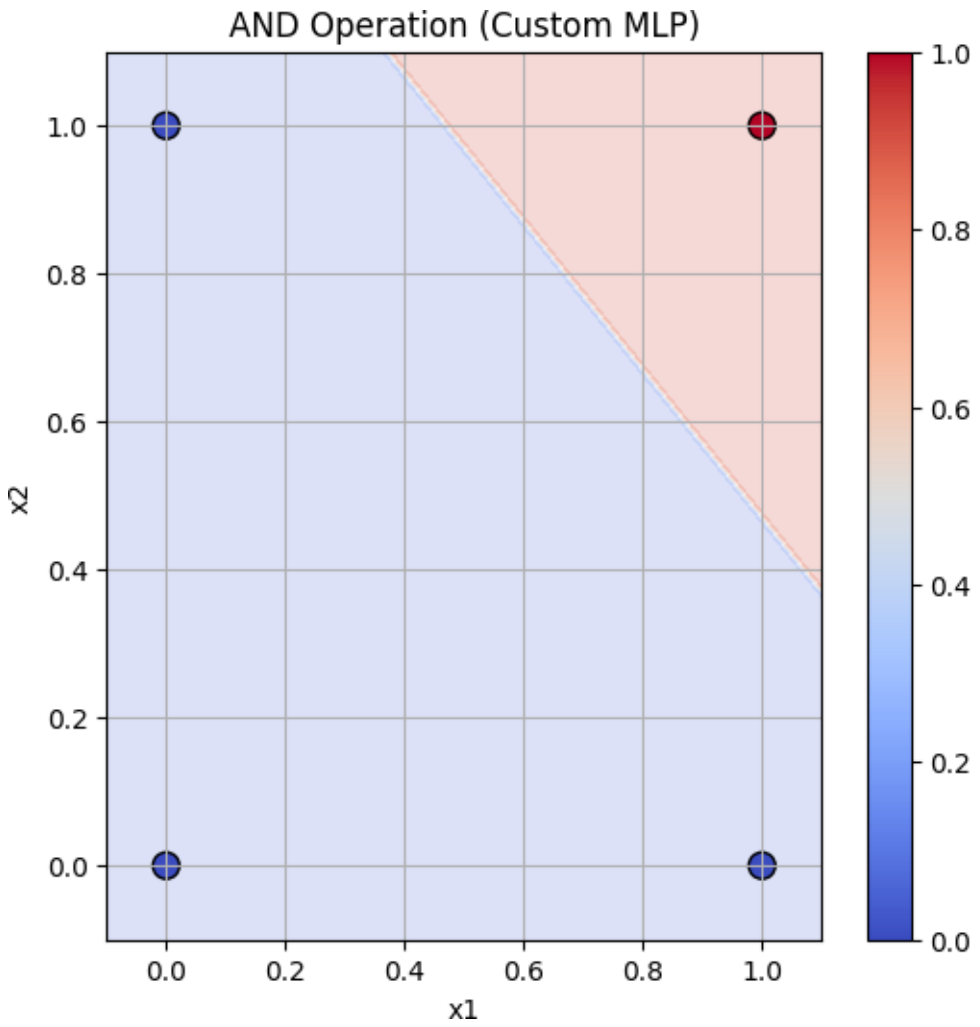

```
Epoch 0: Accuracy = 50.00%
Epoch 100: Accuracy = 50.00%
Epoch 200: Accuracy = 50.00%
Epoch 300: Accuracy = 50.00%
Epoch 400: Accuracy = 50.00%
Epoch 500: Accuracy = 50.00%
Epoch 600: Accuracy = 50.00%
Epoch 700: Accuracy = 50.00%
Epoch 800: Accuracy = 50.00%
Epoch 900: Accuracy = 50.00%
```



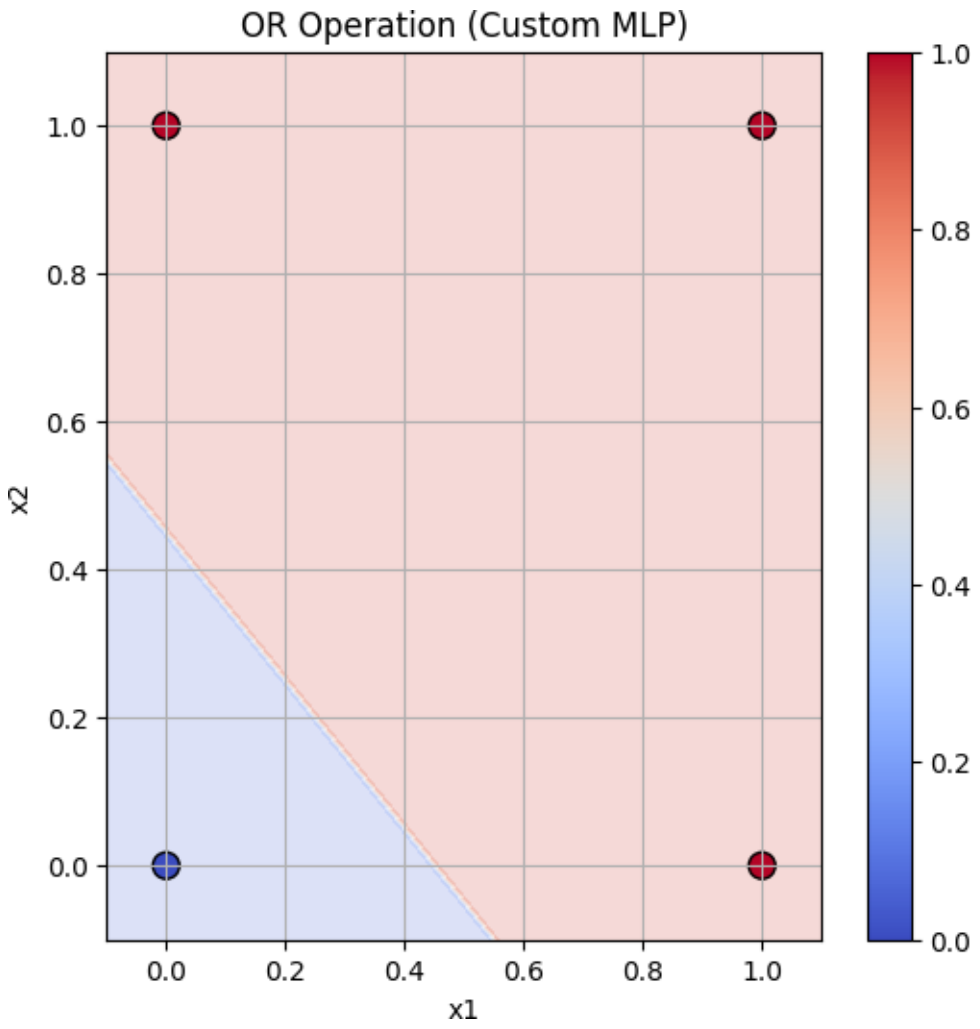
[ ]:
```
for op_name, op_func in operations.items():
    # Compute the result of the operation
    y = np.array([op_func(a, b) for a, b in zip(x1, x2)])
    y = y.reshape(-1, 1)  # Reshape the output to be a column vector for MLP

    # Create the MLP model
    mlp = MLP1(input_size=2, hidden_size=15, output_size=1, learning_rate=0.1,
epochs=10000)
    mlp.train(X, y)

    # Make predictions and plot decision boundary
    predictions = mlp.forward(X)
    predictions = np.round(predictions).flatten()

    # Create a meshgrid for plotting decision boundary
```

```
    h = .02  # Step size in the mesh
    x_min, x_max = -0.1, 1.1
    y_min, y_max = -0.1, 1.1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))

    # Predict for every point in the meshgrid
    Z = mlp.forward(np.c_[xx.ravel(), yy.ravel()])
    Z = np.round(Z).reshape(xx.shape)

    # Plotting
    plt.figure(figsize=(6, 6))
    plt.contourf(xx, yy, Z, alpha=0.2, cmap='coolwarm')
    plt.scatter(x1, x2, c=y.flatten(), s=100, cmap='coolwarm', edgecolors='k',
label='Data Points')
    plt.title(f'{op_name} Operation (Custom MLP)')
    plt.xlabel('x1')
    plt.ylabel('x2')
    plt.colorbar()
    plt.grid(True)
    plt.show()
```

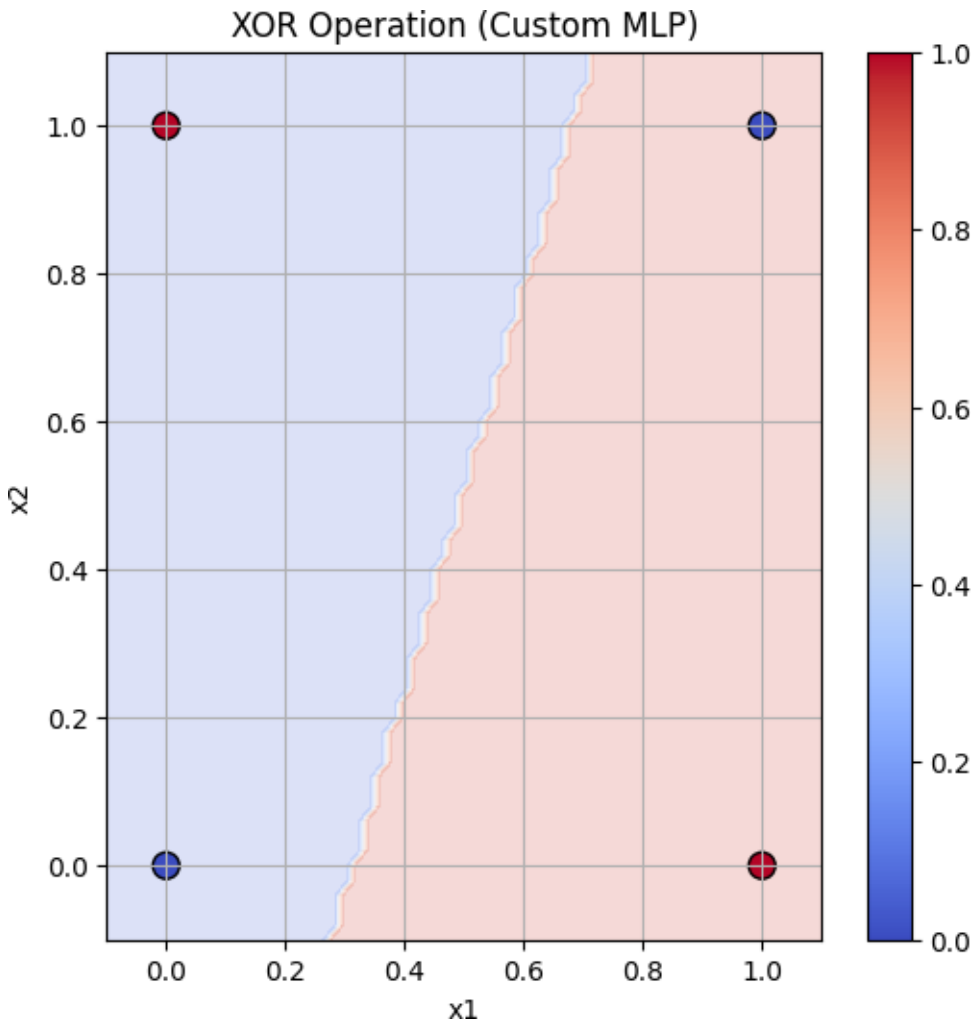
```
Epoch 0, Loss: 0.2865
Epoch 1000, Loss: 0.0755
Epoch 2000, Loss: 0.0113
Epoch 3000, Loss: 0.0043
Epoch 4000, Loss: 0.0024
Epoch 5000, Loss: 0.0016
Epoch 6000, Loss: 0.0012
Epoch 7000, Loss: 0.0009
Epoch 8000, Loss: 0.0008
Epoch 9000, Loss: 0.0006
```



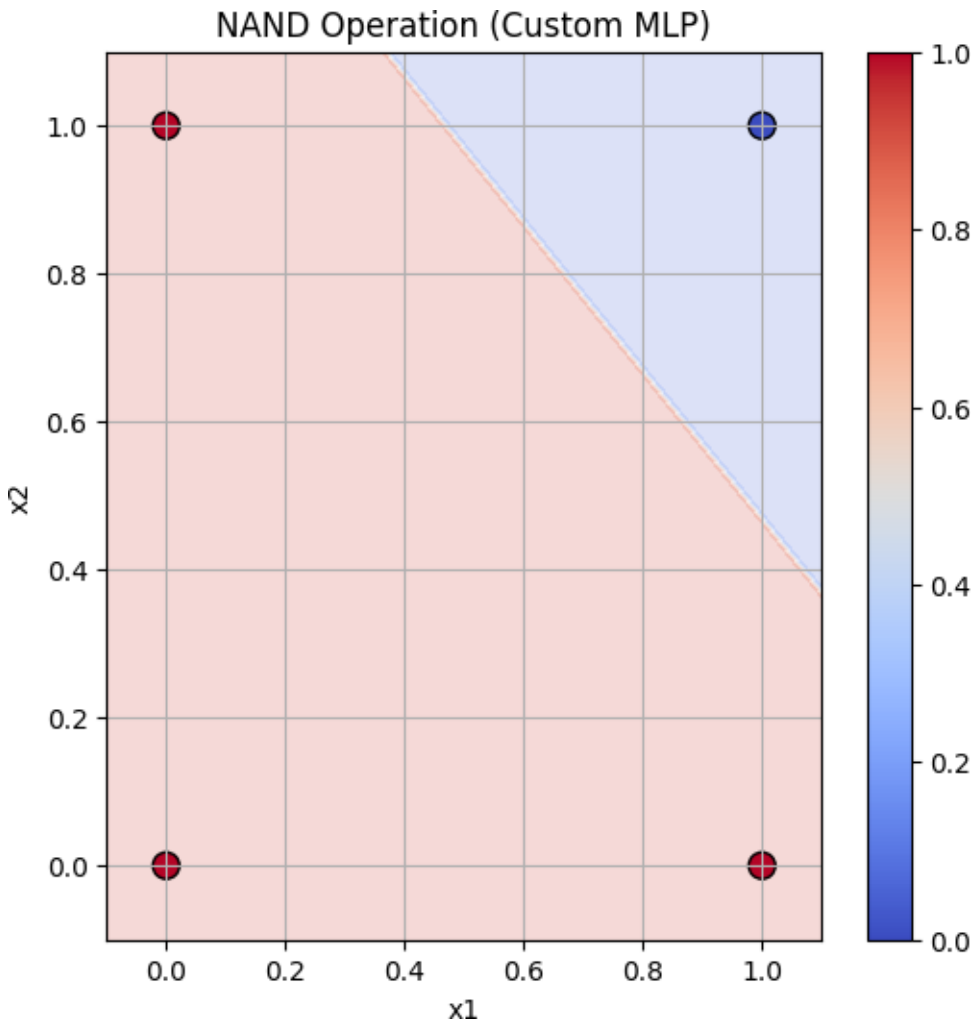

```
Epoch 0, Loss: 0.2740
Epoch 1000, Loss: 0.0425
Epoch 2000, Loss: 0.0052
Epoch 3000, Loss: 0.0022
Epoch 4000, Loss: 0.0013
Epoch 5000, Loss: 0.0009
Epoch 6000, Loss: 0.0007
Epoch 7000, Loss: 0.0005
Epoch 8000, Loss: 0.0005
Epoch 9000, Loss: 0.0004
```



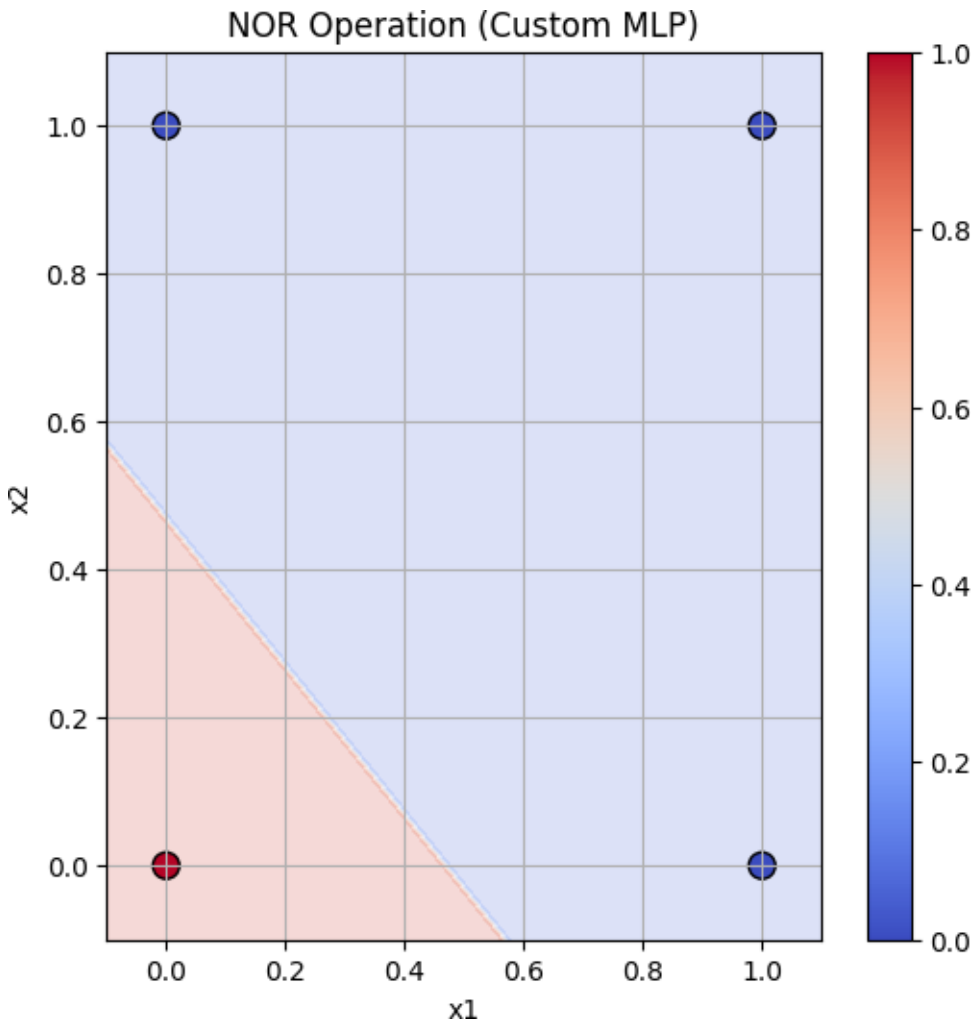

```
Epoch 0, Loss: 0.2502
Epoch 1000, Loss: 0.2500
Epoch 2000, Loss: 0.2500
Epoch 3000, Loss: 0.2500
Epoch 4000, Loss: 0.2500
Epoch 5000, Loss: 0.2500
Epoch 6000, Loss: 0.2500
Epoch 7000, Loss: 0.2500
Epoch 8000, Loss: 0.2500
Epoch 9000, Loss: 0.2500
```



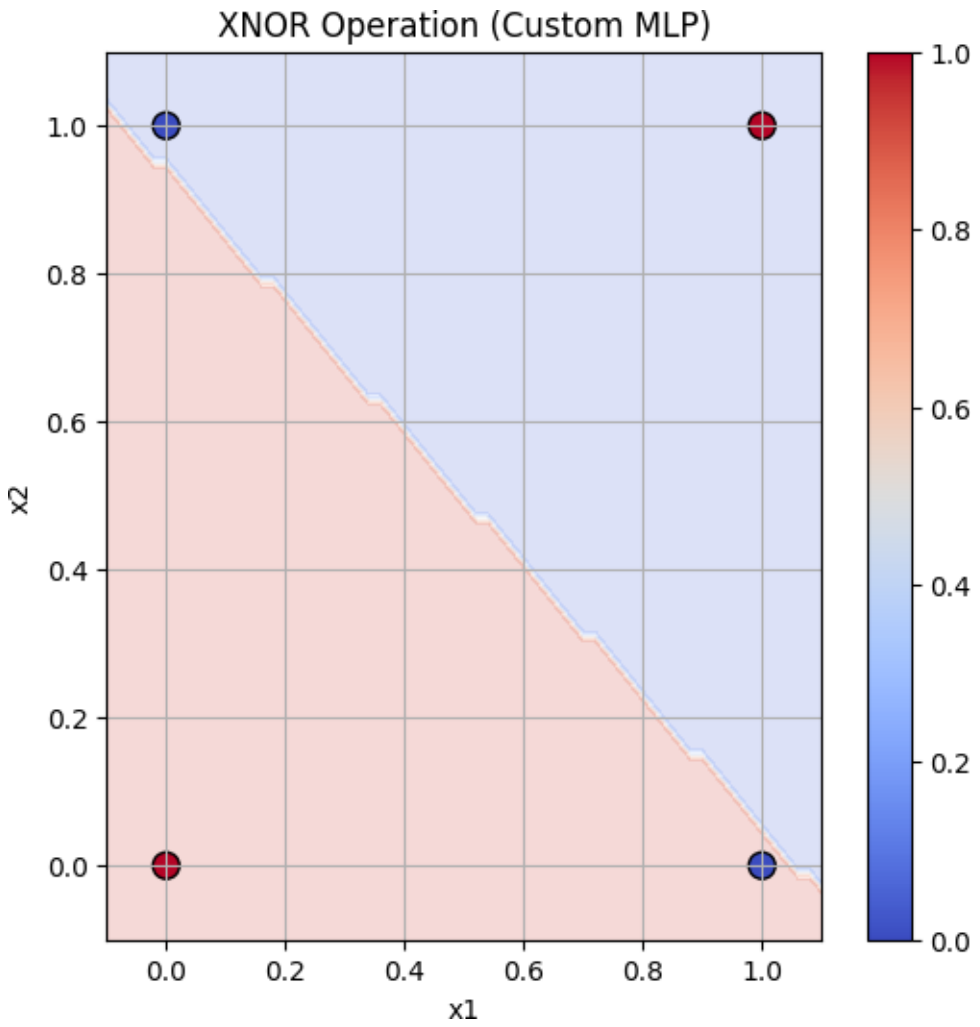

```
Epoch 0, Loss: 0.2197
Epoch 1000, Loss: 0.0583
Epoch 2000, Loss: 0.0097
Epoch 3000, Loss: 0.0039
Epoch 4000, Loss: 0.0023
Epoch 5000, Loss: 0.0015
Epoch 6000, Loss: 0.0011
Epoch 7000, Loss: 0.0009
Epoch 8000, Loss: 0.0007
Epoch 9000, Loss: 0.0006
```




```
Epoch 0, Loss: 0.2270
Epoch 1000, Loss: 0.0907
Epoch 2000, Loss: 0.0075
Epoch 3000, Loss: 0.0027
Epoch 4000, Loss: 0.0015
Epoch 5000, Loss: 0.0010
Epoch 6000, Loss: 0.0008
Epoch 7000, Loss: 0.0006
Epoch 8000, Loss: 0.0005
Epoch 9000, Loss: 0.0004
```




```
Epoch 0, Loss: 0.2516
Epoch 1000, Loss: 0.2500
Epoch 2000, Loss: 0.2500
Epoch 3000, Loss: 0.2500
Epoch 4000, Loss: 0.2500
Epoch 5000, Loss: 0.2500
Epoch 6000, Loss: 0.2500
Epoch 7000, Loss: 0.2500
Epoch 8000, Loss: 0.2500
Epoch 9000, Loss: 0.2500
```



Result: Single layered perceptron struggles with non-linearly separable data. Both in-build SLP and custom SLP have similar performances. While sklearn's MLP can deal with our non-linearly separable data, our custom MLP has failed to do so. Indicating some drawbacks in our custom model,

Multi layered Perceptron

April 2nd, 2025

Aim: Perform multi layer perceptron on lineaerly separable and non-linearly separable data

Description: A multilayer perceptron (MLP) is a neural network with one or more hidden layers between the input and output layers. Each layer is made of neurons that apply activation functions to learn patterns in data. Unlike a single-layer perceptron, MLPs can model complex, non-linear relationships. They are trained using backpropagation to minimize prediction errors. MLPs are widely used for classification and regression tasks. Their layered structure makes them powerful for solving real-world problems.

Algorithm:

Step 1: Import all necessary libraries for data manipulation, visualization, preprocessing, and modeling.

Step 2: Read the dataset from the appropriate source into a usable format such as a DataFrame.

Step 3: View basic information about the dataset.

- Check for missing values and data types.
- Understand the distribution of features and target.
- Visualize relationships between variables.

Step 4: Preprocess the Data

- Normalize or standardize the input features.
- Split the dataset into training and testing sets.

Step 5: Build the MLP Model

- Define the architecture of the Multi-Layer Perceptron with input, hidden, and output layers.
- Specify activation functions, loss function, and optimizer.

Step 6: Train the MLP Model

Step 7: Evaluate the Model

[1]:
```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from keras.models import Sequential
from keras.layers import Dense
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.datasets import load_iris
```

[5]:
```
df=pd.read_csv("//home//aia3_ml//Downloads//archive  (12)//pcos_dataset.csv");
```

[6]:
```
df.head()
```

[6] :

	Age	BMI	Menstrual_Irregularity	Testosterone_Level(ng/dL)	\
0	24	34.7	1	25.2	
1	37	26.4	0	57.1	
2	32	23.6	0	92.7	
3	28	28.8	0	63.1	
4	25	22.1	1	59.8	

	Antral_Follicle_Count	PCOS_Diagnosis
0	20	0
1	25	0
2	28	0
3	26	0
4	8	0

[7] :
```
#Printing some info about the dataset
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 6 columns):
 #   Column                    Non-Null Count  Dtype
---  ------                    --------------  -----
 0   Age                       1000 non-null   int64
 1   BMI                       1000 non-null   float64
 2   Menstrual_Irregularity    1000 non-null   int64
 3   Testosterone_Level(ng/dL) 1000 non-null   float64
 4   Antral_Follicle_Count     1000 non-null   int64
 5   PCOS_Diagnosis            1000 non-null   int64
dtypes: float64(2), int64(4)
memory usage: 47.0 KB
```

[8] :
```
#describe the dataset
df.describe()
```

[8]:

	Age	BMI	Menstrual_Irregularity	\
count	1000.000000	1000.00000	1000.000000	
mean	31.771000	26.38700	0.530000	
std	8.463462	4.93554	0.499349	
min	18.000000	18.10000	0.000000	
25%	24.000000	21.90000	0.000000	
50%	32.000000	26.40000	1.000000	
75%	39.000000	30.50000	1.000000	
max	45.000000	35.00000	1.000000	

	Testosterone_Level(ng/dL)	Antral_Follicle_Count	PCOS_Diagnosis
count	1000.000000	1000.000000	1000.000000
mean	60.159500	17.469000	0.199000
std	23.160204	7.069301	0.399448
min	20.000000	5.000000	0.000000
25%	41.700000	12.000000	0.000000
50%	60.000000	18.000000	0.000000
75%	80.300000	23.250000	0.000000
max	99.800000	29.000000	1.000000

[9]:
```
#Check for NULL values
df.isna().sum().sum()
```

[9]: 0

[10]:
```
#Finding pairwise correlation btw columns
df.corr()
```

[10]:

	Age	BMI	Menstrual_Irregularity	\
Age	1.000000	-0.049455	0.032300	
BMI	-0.049455	1.000000	0.031189	
Menstrual_Irregularity	0.032300	0.031189	1.000000	
Testosterone_Level(ng/dL)	-0.050129	0.003811	0.042694	
Antral_Follicle_Count	0.017841	0.030724	0.035851	
PCOS_Diagnosis	-0.064675	0.377852	0.469376	

	Testosterone_Level(ng/dL)	Antral_Follicle_Count	\
Age	-0.050129	0.017841	
BMI	0.003811	0.030724	
Menstrual_Irregularity	0.042694	0.035851	
Testosterone_Level(ng/dL)	1.000000	0.011976	
Antral_Follicle_Count	0.011976	1.000000	
PCOS_Diagnosis	0.200817	0.192014	

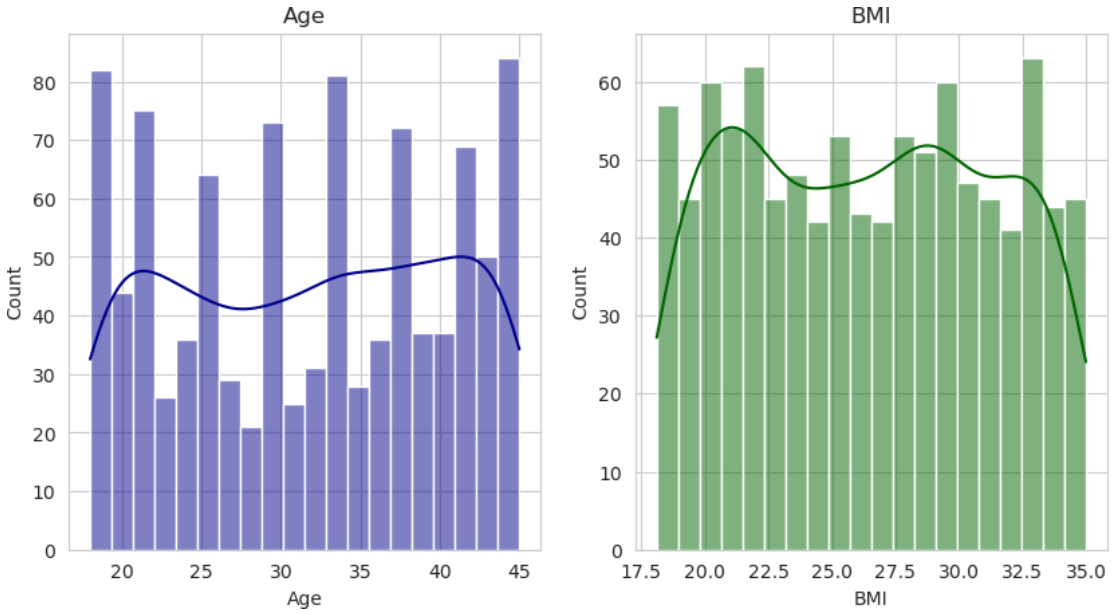
	PCOS_Diagnosis
Age	-0.064675
BMI	0.377852

Menstrual_Irregularity	0.469376
Testosterone_Level(ng/dL)	0.200817
Antral_Follicle_Count	0.192014
PCOS_Diagnosis	1.000000

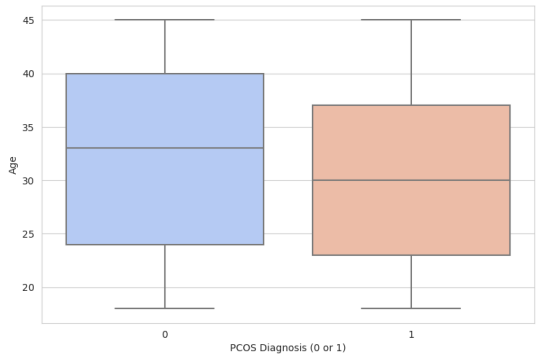
[11] :
```
#Find shape and duplicated rows count
print("Shape:",df.shape)
print("No. of duplicated rows:",df.duplicated().sum())
```

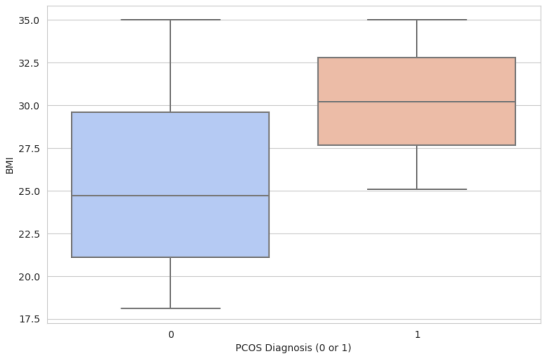
Shape: (1000, 6)
No. of duplicated rows: 0

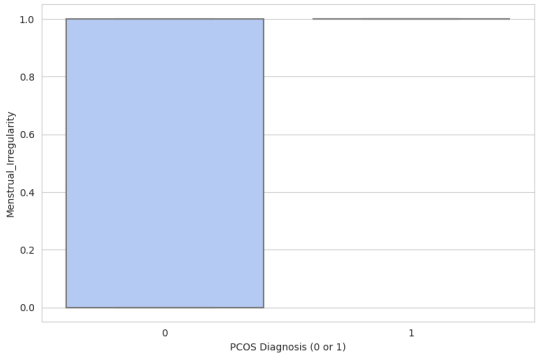
[12] :
```
#Visualizing the data
#Plotting the distribution of age nad BMI
sns.set_style('whitegrid')
plt.figure(figsize=(10,5))
plt.subplot(1,2,1)
sns.histplot(df['Age'],bins=20,kde=True,color='darkblue')
plt.title('Age')

plt.subplot(1,2,2)
sns.histplot(df['BMI'],bins=20,kde=True,color='darkgreen')
plt.title('BMI')
plt.show()
```

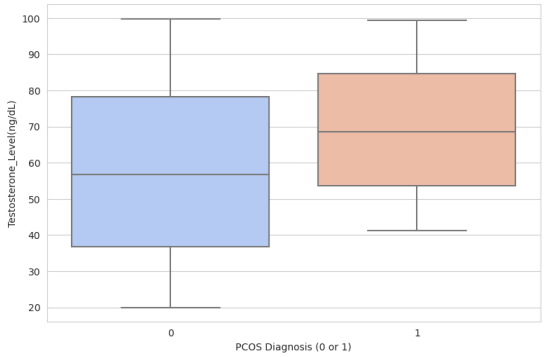


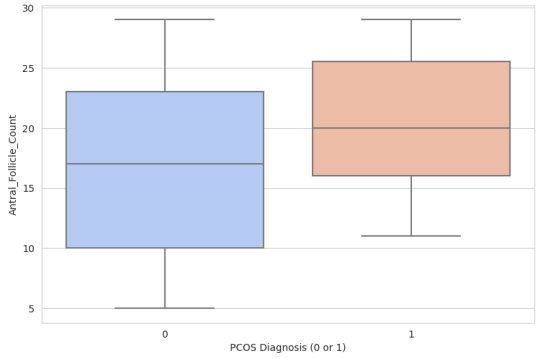
[31]:
```
#Boxplot of PCOS VS Features
plt.figure(figsize=(20,20))
for i in range(1,len(df.columns)):
    plt.subplot(3,2,i)
    sns.boxplot(x=df['PCOS_Diagnosis'],y=df[df.columns[i-1]],palette='coolwarm')
    plt.xlabel("PCOS Diagnosis (0 or 1)")
    plt.ylabel(f"{df.columns[i-1]}")
plt.show()
```



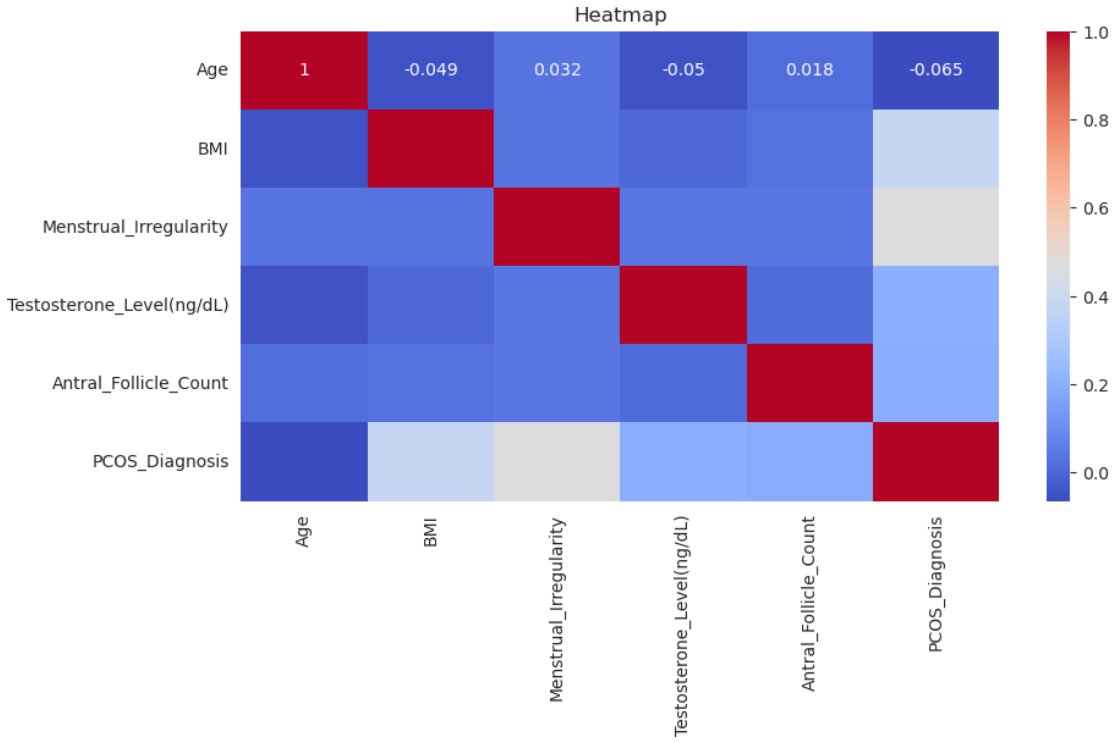




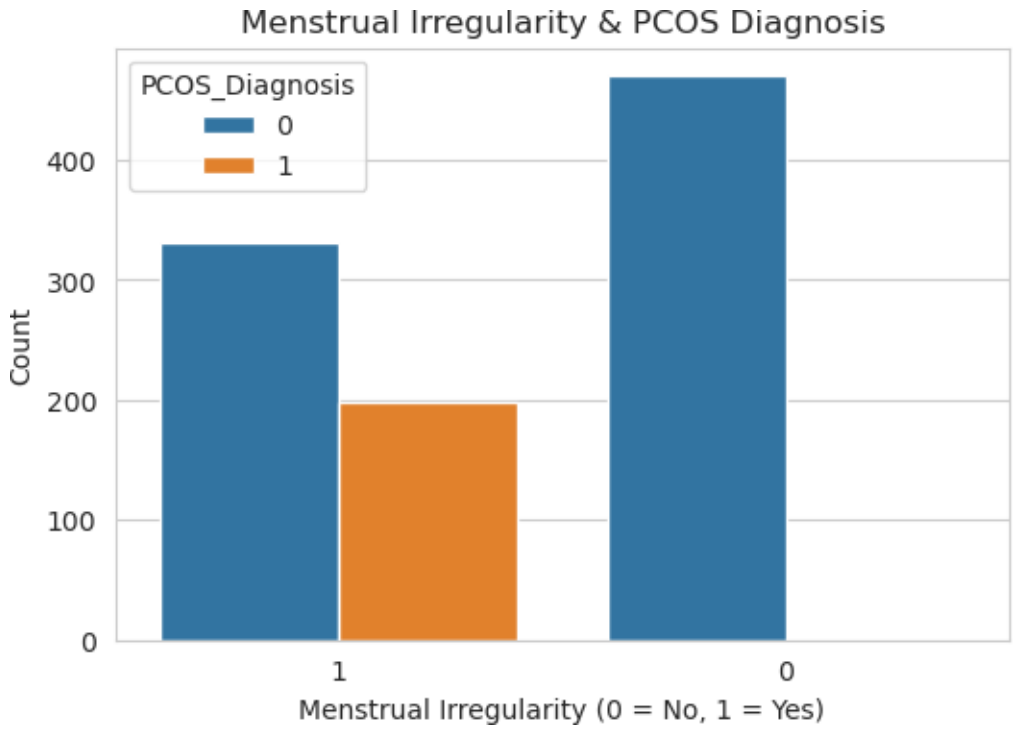




[13] :
```
#Correlation  heatmap
plt.figure(figsize=(10,5))
sns.heatmap(df.corr(),annot=True,cmap='coolwarm')
plt.title('Heatmap')
plt.show()
```



[14] :
```
# Bar Chart: Menstrual Irregularity vs. PCOS
plt.figure(figsize=(6, 4))
sns.countplot(x=df['Menstrual_Irregularity'].astype('str'),
  hue=df['PCOS_Diagnosis'].astype('str'))
plt.xlabel("Menstrual Irregularity (0 = No, 1 = Yes)")
plt.ylabel("Count")
plt.title("Menstrual Irregularity & PCOS Diagnosis")
plt.show()
```



[15] :
```
#Checking distribution of target variable
df['PCOS_Diagnosis'].value_counts()
```

[15]:
```
PCOS_Diagnosis
0    801
1    199
Name: count, dtype: int64
```

[94]:
```
simple_mlp = Sequential()
simple_mlp.add(Dense(60, input_dim=5,activation='relu'))
simple_mlp.add(Dense(60,activation='relu'))
simple_mlp.add(Dense(50,activation='relu'))
simple_mlp.add(Dense(40,activation='relu'))
simple_mlp.add(Dense(30,activation='relu'))
simple_mlp.add(Dense(1,activation='sigmoid'))
simple_mlp.compile(loss='binary_crossentropy',
  optimizer='adam',metrics=['accuracy'])
```

[95]:
```
simple_mlp.summary()
```

Model: "sequential_12"

Layer (type)	Output Shape	Param #
dense_54 (Dense)	(None, 60)	360
dense_55 (Dense)	(None, 60)	3,660
dense_56 (Dense)	(None, 50)	3,050
dense_57 (Dense)	(None, 40)	2,040
dense_58 (Dense)	(None, 30)	1,230
dense_59 (Dense)	(None, 1)	31

Total params: 10,371 (40.51 KB)

Trainable params: 10,371 (40.51 KB)

Non-trainable params: 0 (0.00 B)

[96]:
```
#Train test split
X=df.drop(columns='PCOS_Diagnosis')
y=df['PCOS_Diagnosis']
X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.
  2,random_state=42,stratify=y)
```

[103]:
```
simple_mlp.fit(X_train, y_train, epochs=100, batch_size=100)
```

```
Epoch 1/100
8/8               0s 3ms/step -
accuracy: 0.9514 - loss: 0.1159
Epoch 2/100
8/8               0s 3ms/step -
accuracy: 0.9670 - loss: 0.0939
Epoch 3/100
8/8               0s 3ms/step -
accuracy: 0.9617 - loss: 0.0920
Epoch 4/100
8/8               0s 2ms/step -
```

accuracy: 0.9506 - loss: 0.1083
Epoch 5/100
8/8 **0s** 3ms/step -
accuracy: 0.9634 - loss: 0.0982
Epoch 6/100
8/8 **0s** 3ms/step -
accuracy: 0.9622 - loss: 0.0983
Epoch 7/100
8/8 **0s** 2ms/step -
accuracy: 0.9563 - loss: 0.1001
Epoch 8/100
8/8 **0s** 2ms/step -
accuracy: 0.9570 - loss: 0.0977
Epoch 9/100
8/8 **0s** 3ms/step -
accuracy: 0.9689 - loss: 0.0858
Epoch 10/100
8/8 **0s** 3ms/step -
accuracy: 0.9533 - loss: 0.1020
Epoch 11/100
8/8 **0s** 4ms/step -
accuracy: 0.9537 - loss: 0.1191
Epoch 12/100
8/8 **0s** 3ms/step -
accuracy: 0.9584 - loss: 0.0943
Epoch 13/100
8/8 **0s** 3ms/step -
accuracy: 0.9671 - loss: 0.0858
Epoch 14/100
8/8 **0s** 3ms/step -
accuracy: 0.9708 - loss: 0.0737
Epoch 15/100
8/8 **0s** 3ms/step -
accuracy: 0.9624 - loss: 0.0764
Epoch 16/100
8/8 **0s** 2ms/step -
accuracy: 0.9597 - loss: 0.0823
Epoch 17/100
8/8 **0s** 3ms/step -
accuracy: 0.9551 - loss: 0.0918
Epoch 18/100
8/8 **0s** 2ms/step -
accuracy: 0.9512 - loss: 0.1144
Epoch 19/100
8/8 **0s** 2ms/step -
accuracy: 0.9151 - loss: 0.2322
Epoch 20/100
8/8 **0s** 3ms/step -

accuracy: 0.9032 - loss: 0.2314
Epoch 21/100
8/8 **0s** 3ms/step -
accuracy: 0.8998 - loss: 0.2337
Epoch 22/100
8/8 **0s** 3ms/step -
accuracy: 0.9398 - loss: 0.1308
Epoch 23/100
8/8 **0s** 4ms/step -
accuracy: 0.9016 - loss: 0.2075
Epoch 24/100
8/8 **0s** 2ms/step -
accuracy: 0.9488 - loss: 0.1220
Epoch 25/100
8/8 **0s** 2ms/step -
accuracy: 0.9515 - loss: 0.1220
Epoch 26/100
8/8 **0s** 3ms/step -
accuracy: 0.9577 - loss: 0.1177
Epoch 27/100
8/8 **0s** 3ms/step -
accuracy: 0.9492 - loss: 0.1153
Epoch 28/100
8/8 **0s** 3ms/step -
accuracy: 0.9625 - loss: 0.0906
Epoch 29/100
8/8 **0s** 3ms/step -
accuracy: 0.9647 - loss: 0.0877
Epoch 30/100
8/8 **0s** 3ms/step -
accuracy: 0.9617 - loss: 0.0833
Epoch 31/100
8/8 **0s** 2ms/step -
accuracy: 0.9473 - loss: 0.1067
Epoch 32/100
8/8 **0s** 3ms/step -
accuracy: 0.9564 - loss: 0.1058
Epoch 33/100
8/8 **0s** 3ms/step -
accuracy: 0.9666 - loss: 0.0966
Epoch 34/100
8/8 **0s** 3ms/step -
accuracy: 0.9558 - loss: 0.1023
Epoch 35/100
8/8 **0s** 3ms/step -
accuracy: 0.9649 - loss: 0.0881
Epoch 36/100
8/8 **0s** 3ms/step -

accuracy: 0.9747 - loss: 0.0712
Epoch 37/100
8/8 **0s** 3ms/step -
accuracy: 0.9762 - loss: 0.0651
Epoch 38/100
8/8 **0s** 3ms/step -
accuracy: 0.9730 - loss: 0.0698
Epoch 39/100
8/8 **0s** 3ms/step -
accuracy: 0.9741 - loss: 0.0681
Epoch 40/100
8/8 **0s** 2ms/step -
accuracy: 0.9745 - loss: 0.0731
Epoch 41/100
8/8 **0s** 3ms/step -
accuracy: 0.9806 - loss: 0.0660
Epoch 42/100
8/8 **0s** 2ms/step -
accuracy: 0.9652 - loss: 0.0834
Epoch 43/100
8/8 **0s** 3ms/step -
accuracy: 0.9626 - loss: 0.0761
Epoch 44/100
8/8 **0s** 3ms/step -
accuracy: 0.9679 - loss: 0.0794
Epoch 45/100
8/8 **0s** 3ms/step -
accuracy: 0.9708 - loss: 0.0702
Epoch 46/100
8/8 **0s** 3ms/step -
accuracy: 0.9636 - loss: 0.0796
Epoch 47/100
8/8 **0s** 3ms/step -
accuracy: 0.9611 - loss: 0.0934
Epoch 48/100
8/8 **0s** 2ms/step -
accuracy: 0.9658 - loss: 0.0912
Epoch 49/100
8/8 **0s** 3ms/step -
accuracy: 0.9627 - loss: 0.1081
Epoch 50/100
8/8 **0s** 3ms/step -
accuracy: 0.9650 - loss: 0.0813
Epoch 51/100
8/8 **0s** 3ms/step -
accuracy: 0.9720 - loss: 0.0691
Epoch 52/100
8/8 **0s** 3ms/step -

accuracy: 0.9648 - loss: 0.0799
Epoch 53/100
8/8 **0s** 3ms/step -
accuracy: 0.9559 - loss: 0.1022
Epoch 54/100
8/8 **0s** 3ms/step -
accuracy: 0.9473 - loss: 0.1160
Epoch 55/100
8/8 **0s** 2ms/step -
accuracy: 0.9569 - loss: 0.0827
Epoch 56/100
8/8 **0s** 3ms/step -
accuracy: 0.9756 - loss: 0.0619
Epoch 57/100
8/8 **0s** 2ms/step -
accuracy: 0.9770 - loss: 0.0697
Epoch 58/100
8/8 **0s** 2ms/step -
accuracy: 0.9776 - loss: 0.0657
Epoch 59/100
8/8 **0s** 2ms/step -
accuracy: 0.9668 - loss: 0.0828
Epoch 60/100
8/8 **0s** 2ms/step -
accuracy: 0.9599 - loss: 0.0899
Epoch 61/100
8/8 **0s** 3ms/step -
accuracy: 0.9788 - loss: 0.0727
Epoch 62/100
8/8 **0s** 2ms/step -
accuracy: 0.9601 - loss: 0.0783
Epoch 63/100
8/8 **0s** 2ms/step -
accuracy: 0.9582 - loss: 0.1020
Epoch 64/100
8/8 **0s** 3ms/step -
accuracy: 0.9809 - loss: 0.0750
Epoch 65/100
8/8 **0s** 3ms/step -
accuracy: 0.9664 - loss: 0.0762
Epoch 66/100
8/8 **0s** 2ms/step -
accuracy: 0.9619 - loss: 0.0871
Epoch 67/100
8/8 **0s** 2ms/step -
accuracy: 0.9571 - loss: 0.1001
Epoch 68/100
8/8 **0s** 2ms/step -

accuracy: 0.9347 - loss: 0.1309
Epoch 69/100
8/8 **0s** 2ms/step -
accuracy: 0.9630 - loss: 0.0901
Epoch 70/100
8/8 **0s** 2ms/step -
accuracy: 0.9684 - loss: 0.0781
Epoch 71/100
8/8 **0s** 2ms/step -
accuracy: 0.9770 - loss: 0.0650
Epoch 72/100
8/8 **0s** 3ms/step -
accuracy: 0.9832 - loss: 0.0513
Epoch 73/100
8/8 **0s** 3ms/step -
accuracy: 0.9864 - loss: 0.0456
Epoch 74/100
8/8 **0s** 2ms/step -
accuracy: 0.9729 - loss: 0.0622
Epoch 75/100
8/8 **0s** 2ms/step -
accuracy: 0.9692 - loss: 0.0703
Epoch 76/100
8/8 **0s** 2ms/step -
accuracy: 0.9795 - loss: 0.0617
Epoch 77/100
8/8 **0s** 2ms/step -
accuracy: 0.9780 - loss: 0.0592
Epoch 78/100
8/8 **0s** 2ms/step -
accuracy: 0.9857 - loss: 0.0515
Epoch 79/100
8/8 **0s** 2ms/step -
accuracy: 0.9647 - loss: 0.0829
Epoch 80/100
8/8 **0s** 4ms/step -
accuracy: 0.9835 - loss: 0.0555
Epoch 81/100
8/8 **0s** 2ms/step -
accuracy: 0.9758 - loss: 0.0646
Epoch 82/100
8/8 **0s** 2ms/step -
accuracy: 0.9771 - loss: 0.0640
Epoch 83/100
8/8 **0s** 2ms/step -
accuracy: 0.9743 - loss: 0.0698
Epoch 84/100
8/8 **0s** 3ms/step -

accuracy: 0.9733 - loss: 0.0739
Epoch 85/100
8/8 **0s** 2ms/step -
accuracy: 0.9659 - loss: 0.0735
Epoch 86/100
8/8 **0s** 3ms/step -
accuracy: 0.9833 - loss: 0.0541
Epoch 87/100
8/8 **0s** 2ms/step -
accuracy: 0.9871 - loss: 0.0478
Epoch 88/100
8/8 **0s** 2ms/step -
accuracy: 0.9695 - loss: 0.0686
Epoch 89/100
8/8 **0s** 2ms/step -
accuracy: 0.9785 - loss: 0.0545
Epoch 90/100
8/8 **0s** 2ms/step -
accuracy: 0.9607 - loss: 0.0724
Epoch 91/100
8/8 **0s** 2ms/step -
accuracy: 0.9766 - loss: 0.0646
Epoch 92/100
8/8 **0s** 3ms/step -
accuracy: 0.9788 - loss: 0.0589
Epoch 93/100
8/8 **0s** 2ms/step -
accuracy: 0.9864 - loss: 0.0514
Epoch 94/100
8/8 **0s** 3ms/step -
accuracy: 0.9891 - loss: 0.0433
Epoch 95/100
8/8 **0s** 3ms/step -
accuracy: 0.9899 - loss: 0.0452
Epoch 96/100
8/8 **0s** 2ms/step -
accuracy: 0.9847 - loss: 0.0458
Epoch 97/100
8/8 **0s** 2ms/step -
accuracy: 0.9882 - loss: 0.0374
Epoch 98/100
8/8 **0s** 4ms/step -
accuracy: 0.9782 - loss: 0.0455
Epoch 99/100
8/8 **0s** 2ms/step -
accuracy: 0.9873 - loss: 0.0448
Epoch 100/100
8/8 **0s** 2ms/step -

accuracy: 0.9843 - loss: 0.0498

[103]: <keras.src.callbacks.history.History at 0x7ff87461c210>

[107]:
```
y_pred = simple_mlp.predict(X_test)
```

7/7 **0s** 2ms/step

[108]:
```
y_pred=[1 if i>0.5 else 0 for i in y_pred]
```

[109]:
```
print(accuracy_score(y_test,y_pred)*100)
```

94.5

RESULT: MLP applied successfully and gives excellent performance. However it is incredibly time consuming.