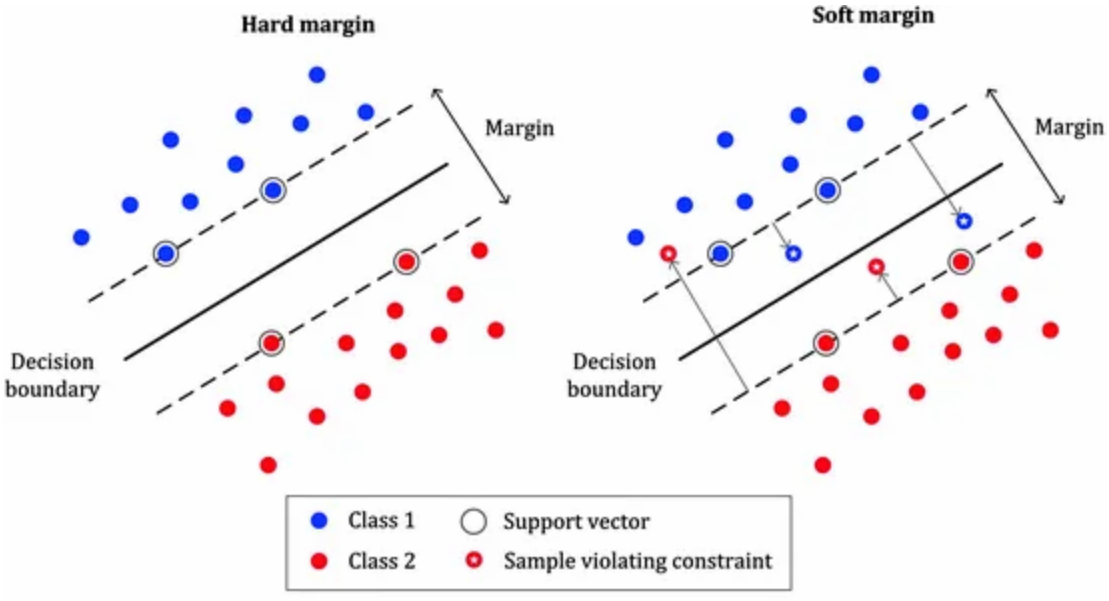
Support Vector Machine

Support Vector Machines (SVMs) are a popular supervised learning algorithm used for classification and regression tasks in machine learning. They are particularly powerful in high-dimensional spaces and are widely used for their effectiveness in both linearly separable and non-linearly separable data.

Key Concepts of SVM

1. **Objective**:
    - SVM aims to find a hyperplane that best separates the data into different classes.
    - In high-dimensional spaces, the hyperplane is generalized to a decision boundary.
2. **Maximum Margin**:
    - The goal is to maximize the margin, which is the distance between the hyperplane and the closest data points (called support vectors) from each class.
    - A larger margin ensures better generalization on unseen data.
3. **Kernel Trick**:
    - For non-linearly separable data, SVM uses kernel functions to transform the input data into a higher-dimensional space where a linear hyperplane can separate the classes.
    - Common kernel functions include:
        - Linear kernel
        - Polynomial kernel
        - Radial Basis Function (RBF) or Gaussian kernel
        - Sigmoid kernel
4. **Soft Margin**:
    - Real-world data often has noise, overlaps, or outliers that make perfect separation impossible.
    - The soft margin approach allows some misclassifications by introducing a penalty for incorrect classifications. This is controlled by the regularization parameter CCC, which balances margin size and classification accuracy.
5. **Support Vectors**:
    - Support vectors are the data points closest to the hyperplane and are critical in defining its position and orientation.



Advantages of SVM

1. **Effective in High-Dimensional Spaces**:
    - SVM performs well in cases where the number of dimensions exceeds the number of samples.
2. **Robust to Overfitting**:
    - Especially effective in scenarios with a clear margin of separation.
3. **Versatile**:
    - SVM can be adapted for non-linear classification with appropriate kernels.

Disadvantages of SVM

1. **Computationally Intensive**:
    - Training can be slow for large datasets, particularly with non-linear kernels.
2. **Requires Careful Parameter Tuning**:
    - The choice of kernel and hyperparameters like CCC and kernel parameters significantly affects performance.
3. **Less Effective with Overlapping Classes**:
    - May struggle when classes are not well separated or when there's significant noise.

Applications

1. **Text Categorization**: Spam detection, sentiment analysis.
2. **Image Recognition**: Face detection, object classification.

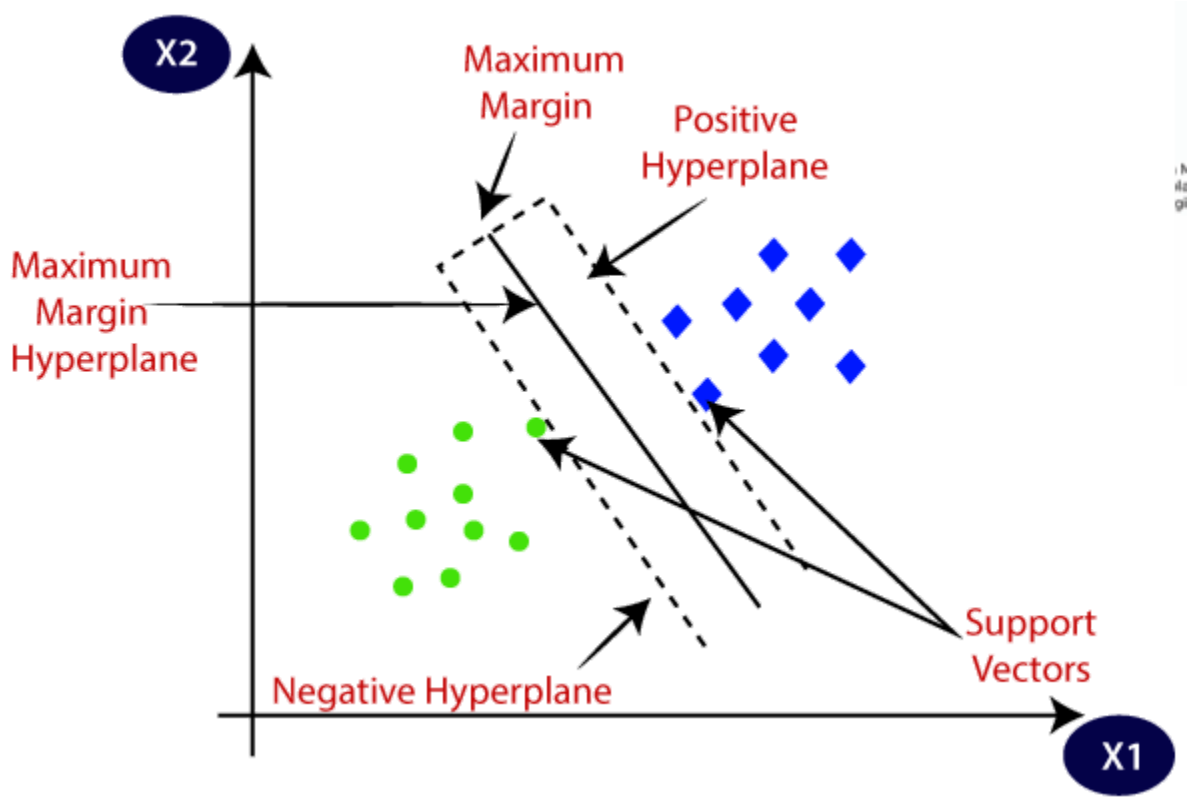
3. **Bioinformatics**: Gene expression classification.
4. **Finance**: Fraud detection, credit scoring.

Support Vector Machine Algorithm

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

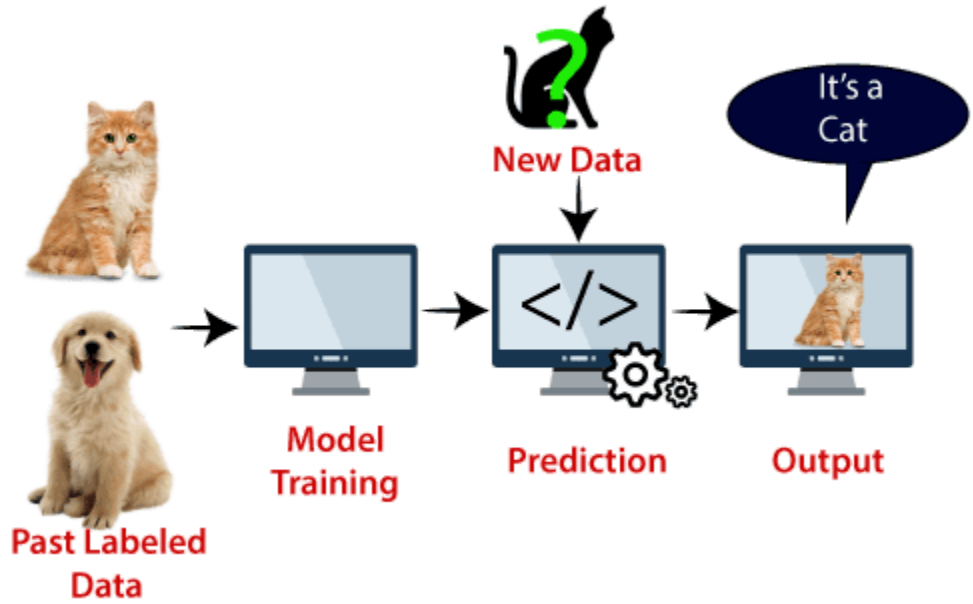
The goal of the SVM algorithm is to create the best line or decision boundary that can segregate n-dimensional space into classes so that we can easily put the new data point in the correct category in the future. This best decision boundary is called a hyperplane.

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



Example: SVM can be understood with the example that we have used in the KNN classifier. Suppose we see a strange cat that also has some features of dogs, so if we want a model that can accurately identify whether it is a cat or dog, so such a model can be created by using the SVM algorithm. We will first train our model with lots of images of cats and dogs so that it can learn about different features of cats and dogs, and then we test it with this strange creature. So as support vector

creates a decision boundary between these two data (cat and dog) and choose extreme cases (support vectors), it will see the extreme case of cat and dog. On the basis of the support vectors, it will classify it as a cat. Consider the below diagram:



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

Types of SVM

SVM can be of two types:

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

Hyperplane and Support Vectors in the SVM algorithm:

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

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

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

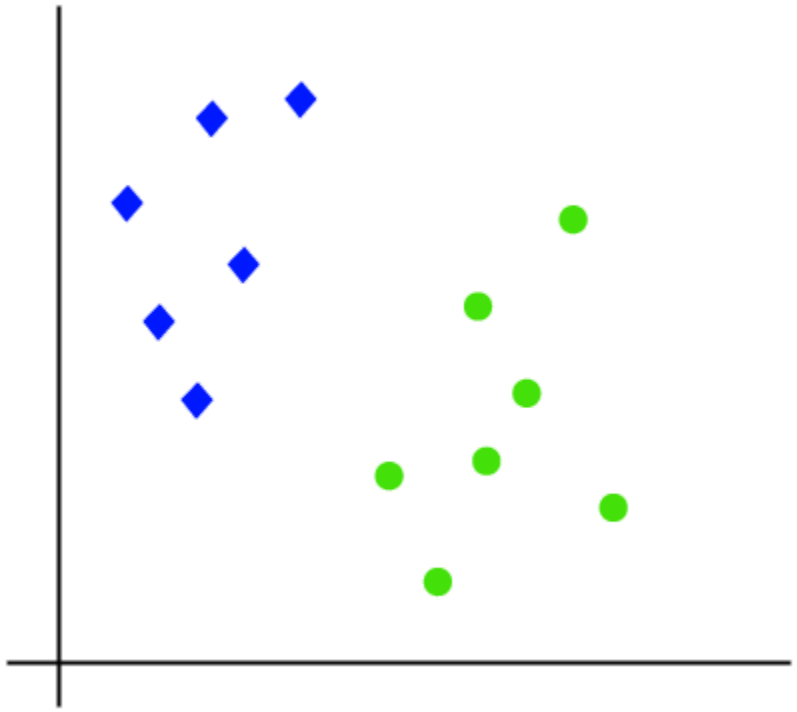
Support Vectors:

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

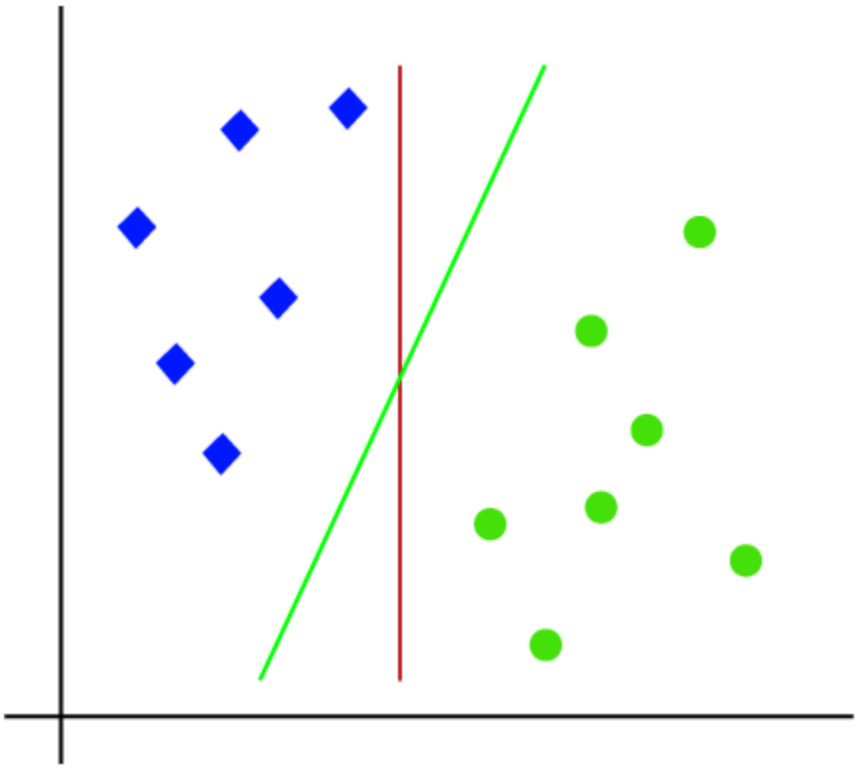
How does SVM works?

Linear SVM:

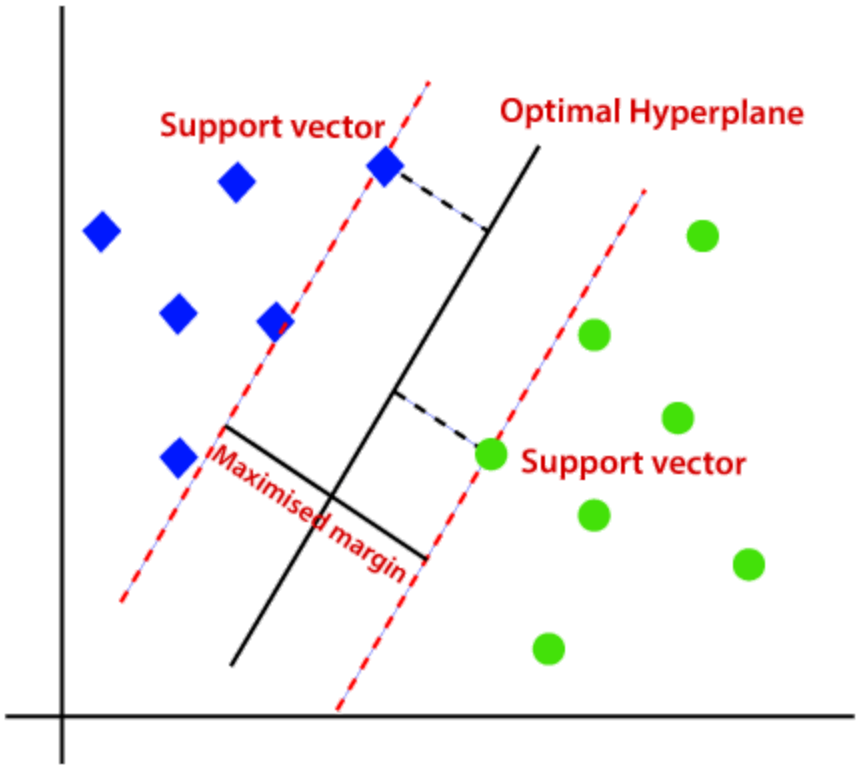
The working of the SVM algorithm can be understood by using an example. Suppose we have a dataset that has two tags (green and blue), and the dataset has two features x1 and x2. We want a classifier that can classify the pair(x1, x2) of coordinates in either green or blue. Consider the below image:



So as it is 2-d space so by just using a straight line, we can easily separate these two classes. But there can be multiple lines that can separate these classes. Consider the below image:



Hence, the SVM algorithm helps to find the best line or decision boundary; this best boundary or region is called as a hyperplane. SVM algorithm finds the closest point of the lines from both the classes. These points are called **support vectors.** The distance between the vectors and the hyperplane is called as **margin**. And the goal of SVM is to maximize this margin. The hyperplane with maximum margin is called the **optimal hyperplane**.




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

# Generate a synthetic dataset
X, y = make_blobs(n_samples=200, centers=2, random_state=42,
cluster_std=1.5)

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3, random_state=42)

# Function to plot decision boundaries for linear SVM
def plot_linear_svm(X, y, clf):
    """
    Plot decision boundary for a linear SVM.

    Parameters:
        X (ndarray): Input features, shape (n_samples, 2).
```

```
        y (ndarray): Target labels, shape (n_samples,).
        clf (SVC): Trained linear SVM classifier.
    """
    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, 0.01),
                         np.arange(y_min, y_max, 0.01))

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

    plt.contourf(xx, yy, Z, alpha=0.8, cmap=plt.cm.coolwarm)
    plt.scatter(X[:, 0], X[:, 1], c=y, edgecolors='k',
cmap=plt.cm.coolwarm)
    plt.title("Linear SVM Decision Boundary")
    plt.show()

# Train a linear SVM classifier
linear_svm = SVC(kernel='linear', C=1, random_state=42)
linear_svm.fit(X_train, y_train)

# Make predictions
y_pred = linear_svm.predict(X_test)

# Evaluate the model
print("Classification Report:")
print(classification_report(y_test, y_pred))
print(f"Accuracy: {accuracy_score(y_test, y_pred):.2f}")

# Plot the decision boundary
plot_linear_svm(X, y, linear_svm)
```

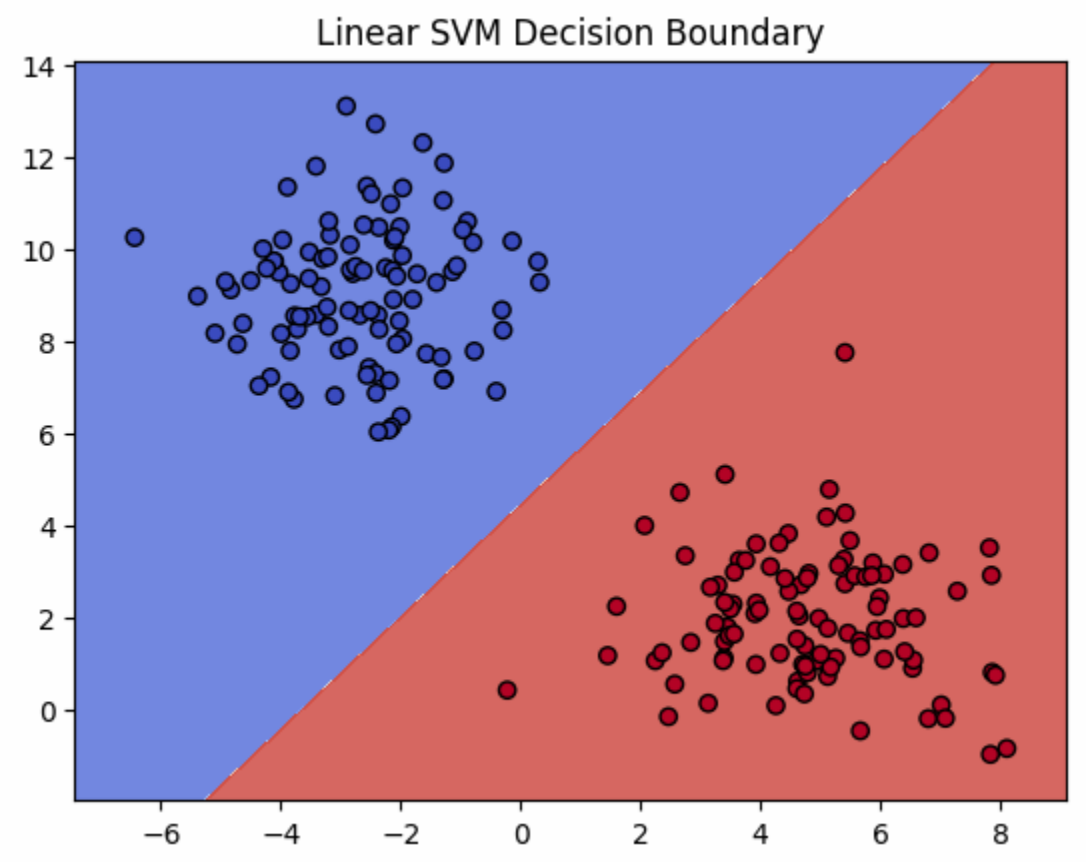
Output

Classification Report:

	precision	recall	f1-score	support
0	1.00	1.00	1.00	37
1	1.00	1.00	1.00	23
accuracy			1.00	60
macro avg	1.00	1.00	1.00	60

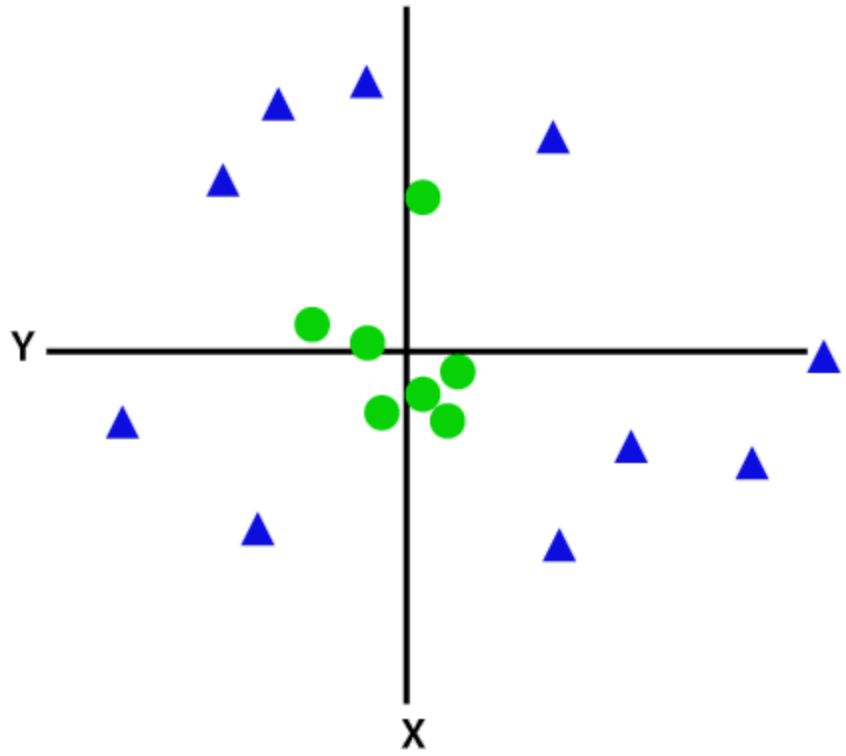
weighted avg 1.00 1.00 1.00 60

Accuracy: 1.00



Non-Linear SVM:

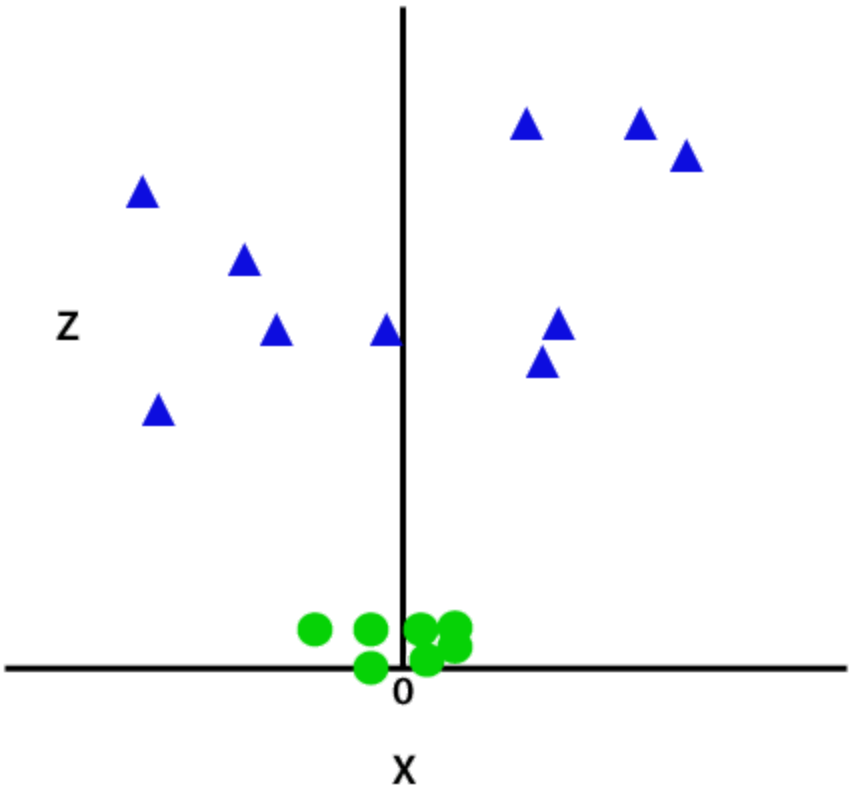
If data is linearly arranged, then we can separate it by using a straight line, but for non-linear data, we cannot draw a single straight line. Consider the below image:



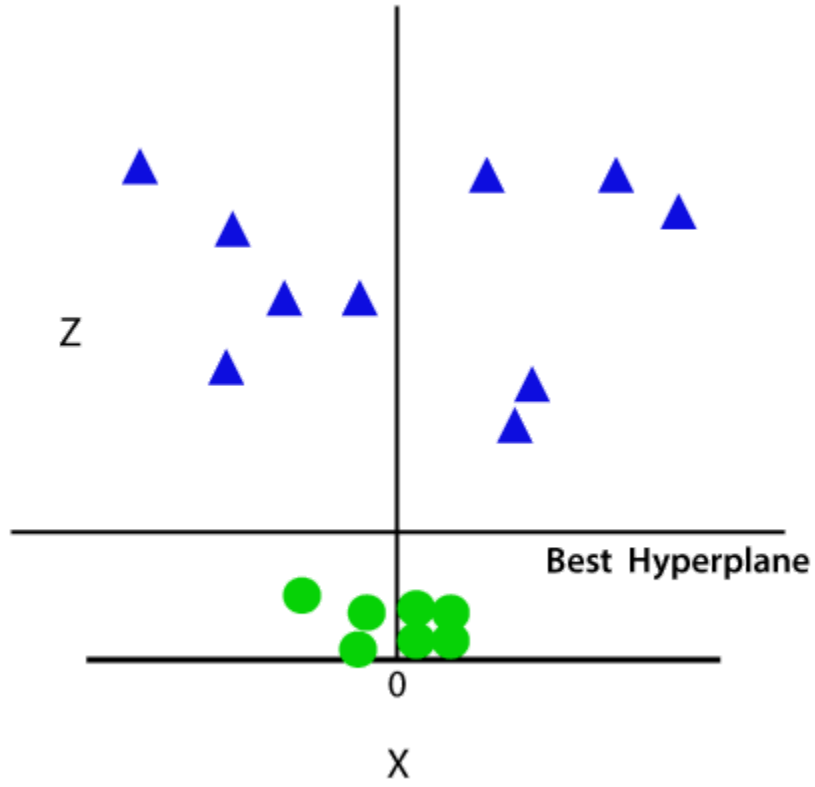
So to separate these data points, we need to add one more dimension. For linear data, we have used two dimensions x and y, so for non-linear data, we will add a third dimension z. It can be calculated as:

$z=x^2+y^2$

By adding the third dimension, the sample space will become as below image:



So now, SVM will divide the datasets into classes in the following way. Consider the below image:

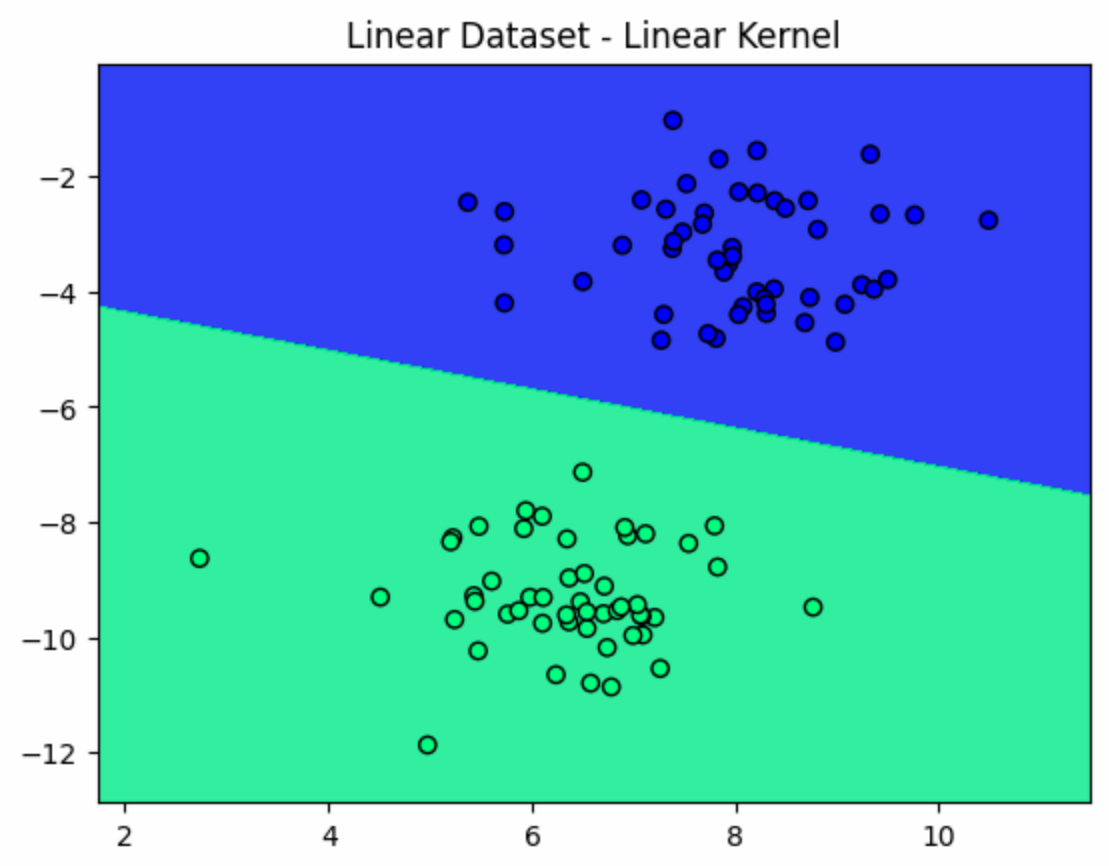


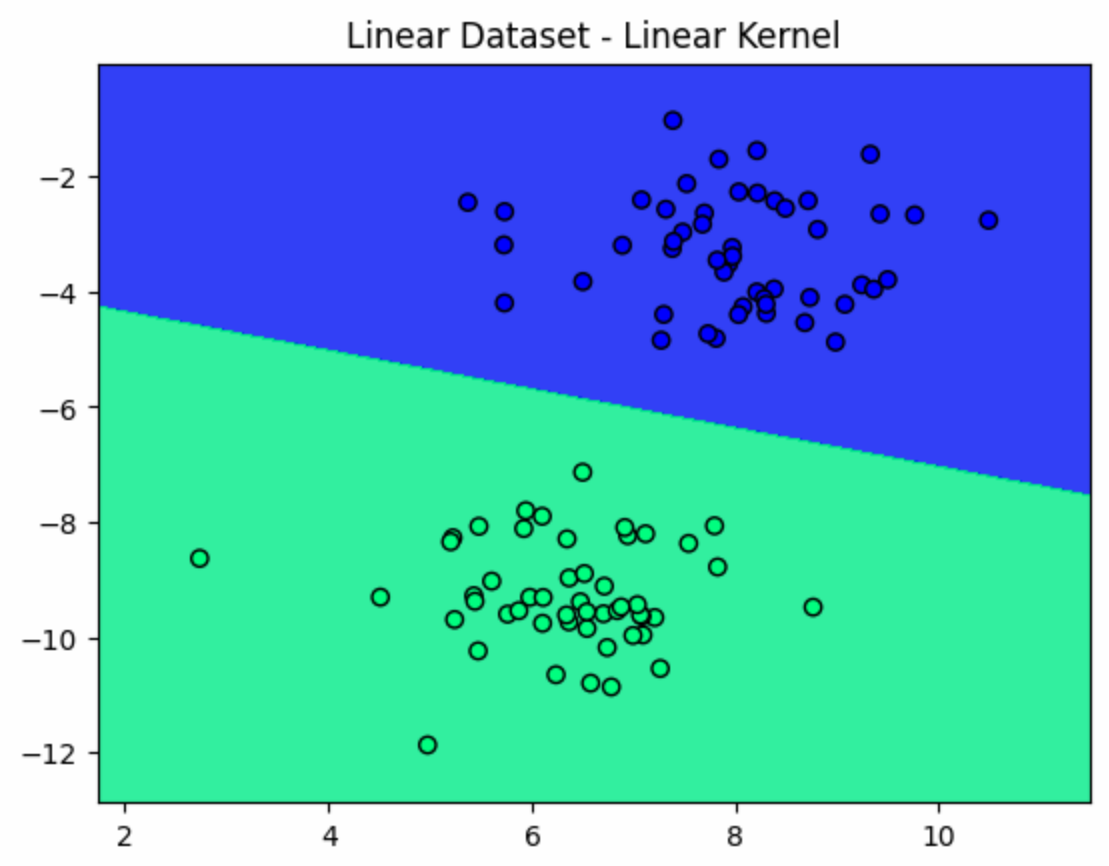
Kernel Trick:

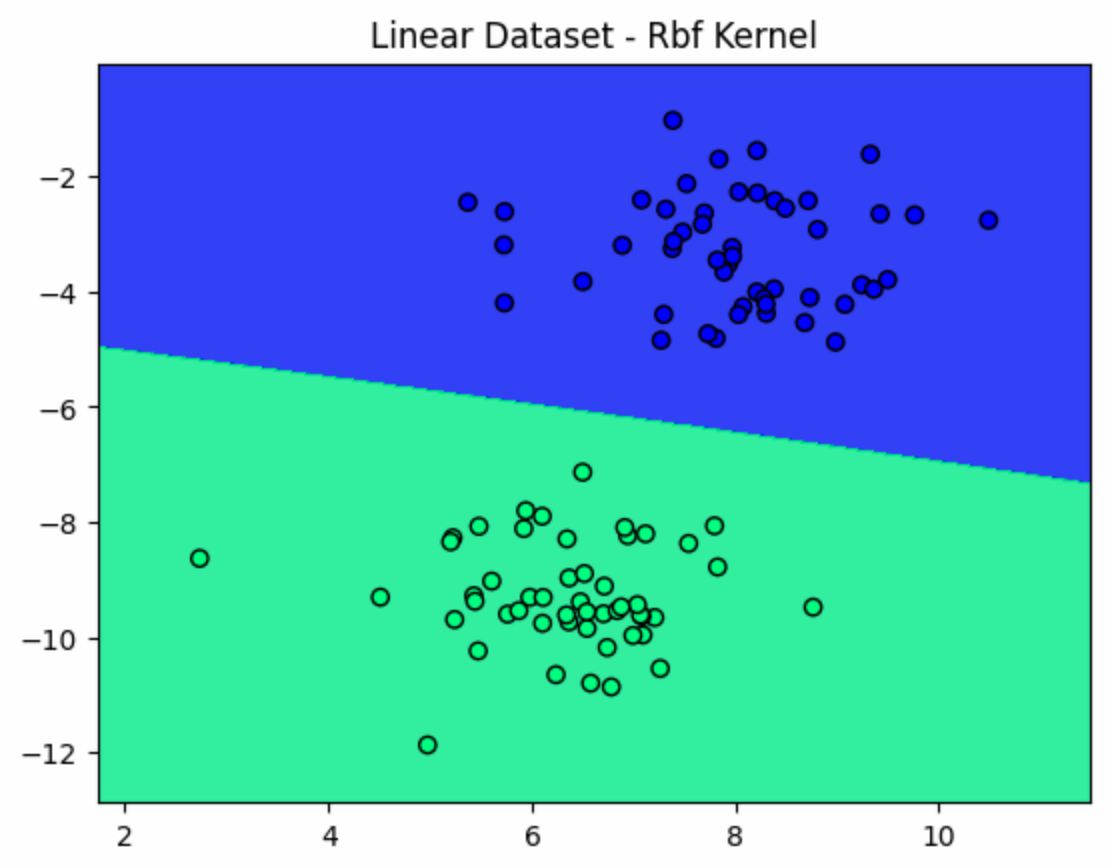
- For non-linearly separable data, SVM uses kernel functions to transform the input data into a higher-dimensional space where a linear hyperplane can separate the classes.
- Common kernel functions include:
  - Linear kernel
  - Polynomial kernel
  - Radial Basis Function (RBF) or Gaussian kernel
  - Sigmoid kernel

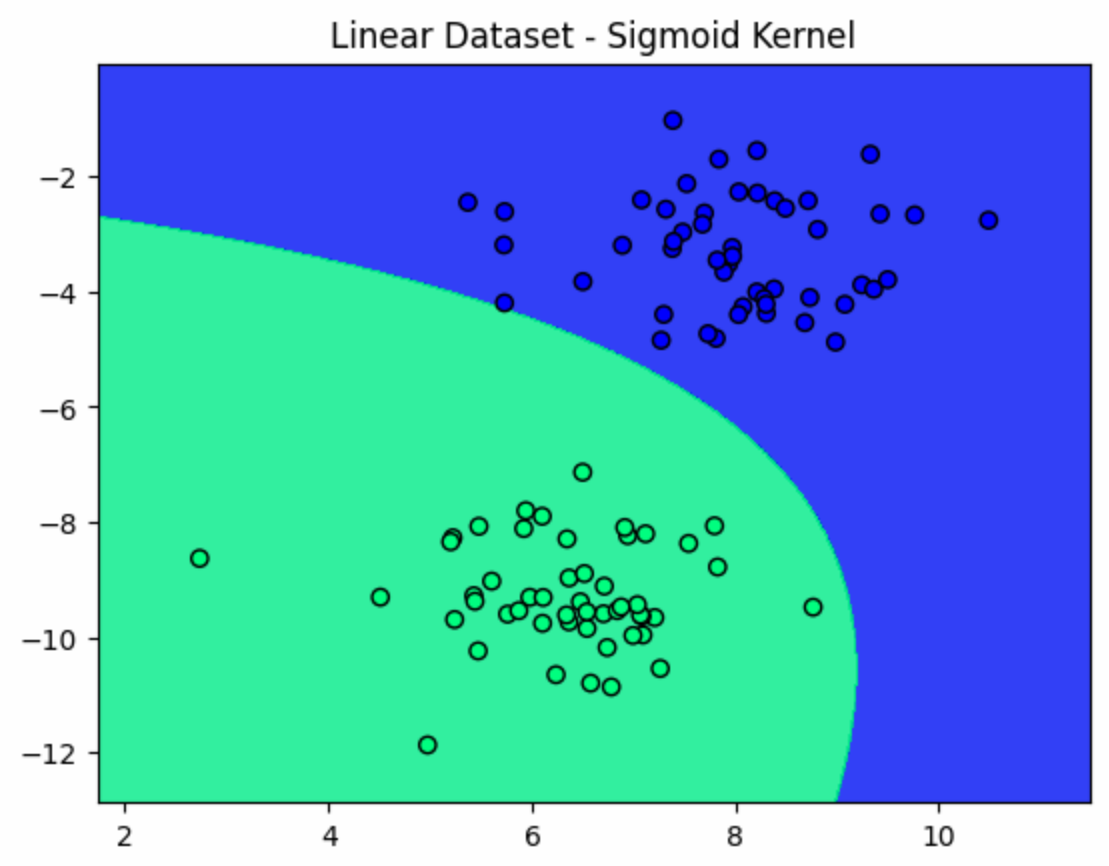
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.svm import SVC
from sklearn.datasets import make_blobs, make_circles, make_moons

# Function to plot SVM results
def plot_svm(X, y, clf, title):
    """
    Plot decision boundary, margins, and dataset points for an SVM classifier.

    Parameters:
        X (ndarray): Input features, shape (n_samples, 2).
        y (ndarray): Target labels, shape (n_samples,).
        clf (SVC): Trained SVM classifier.
        title (str): Title for the plot.
    """
    # Determine plot boundaries by finding the min and max values of X and y coordinates
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1

    # Generate a grid of points (xx, yy) to evaluate the model
    xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.01),  # Create grid in X range
                         np.arange(y_min, y_max, 0.01))  # Create grid in Y range

    # Predict class labels for all points in the grid
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])  # Flatten grid and predict labels
    Z = Z.reshape(xx.shape)  # Reshape predictions to match grid dimensions

    # Plot the decision boundary and dataset points
    plt.contourf(xx, yy, Z, alpha=0.8, cmap='winter')  # Plot decision regions
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap='winter', edgecolors='k')  # Plot data points
```

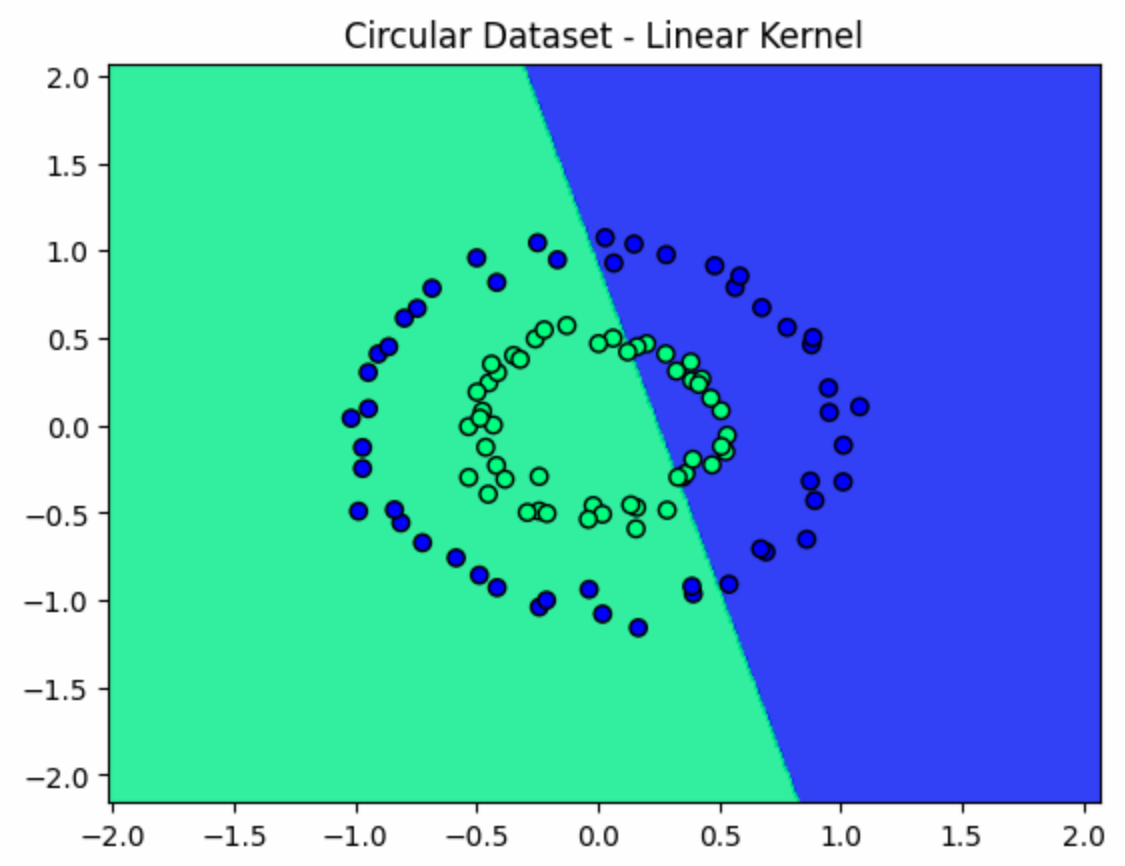
```
    plt.title(title)  # Set the title of the plot
    plt.show()  # Display the plot

# Generate different datasets
datasets = {
    'Linear Dataset': make_blobs(n_samples=100, centers=2, random_state=6),  # Linearly separable data
    'Circular Dataset': make_circles(n_samples=100, factor=0.5, noise=0.05, random_state=0),  # Circular data
    'Moons Dataset': make_moons(n_samples=100, noise=0.1, random_state=0)  # Moon-shaped data
}

# Define SVM kernels to evaluate
kernels = ['linear', 'poly', 'rbf', 'sigmoid']  # SVM kernel types: linear, polynomial, radial basis function,
sigmoid

# Apply SVM with different kernels to each dataset
for dataset_name, (X, y) in datasets.items():  # Iterate over datasets
    for kernel in kernels:  # Iterate over kernel types
        # Create and fit the SVM model with the specified kernel
        clf = SVC(kernel=kernel, C=1, degree=3, gamma='scale', random_state=42)  # Configure SVM
        clf.fit(X, y)  # Train the SVM model on the dataset

        # Output dataset and kernel information to the console
        print(f"Dataset: {dataset_name}, Kernel: {kernel}")

        # Plot results of the SVM classifier
        plot_svm(X, y, clf, f"{dataset_name} - {kernel.capitalize()} Kernel")
```

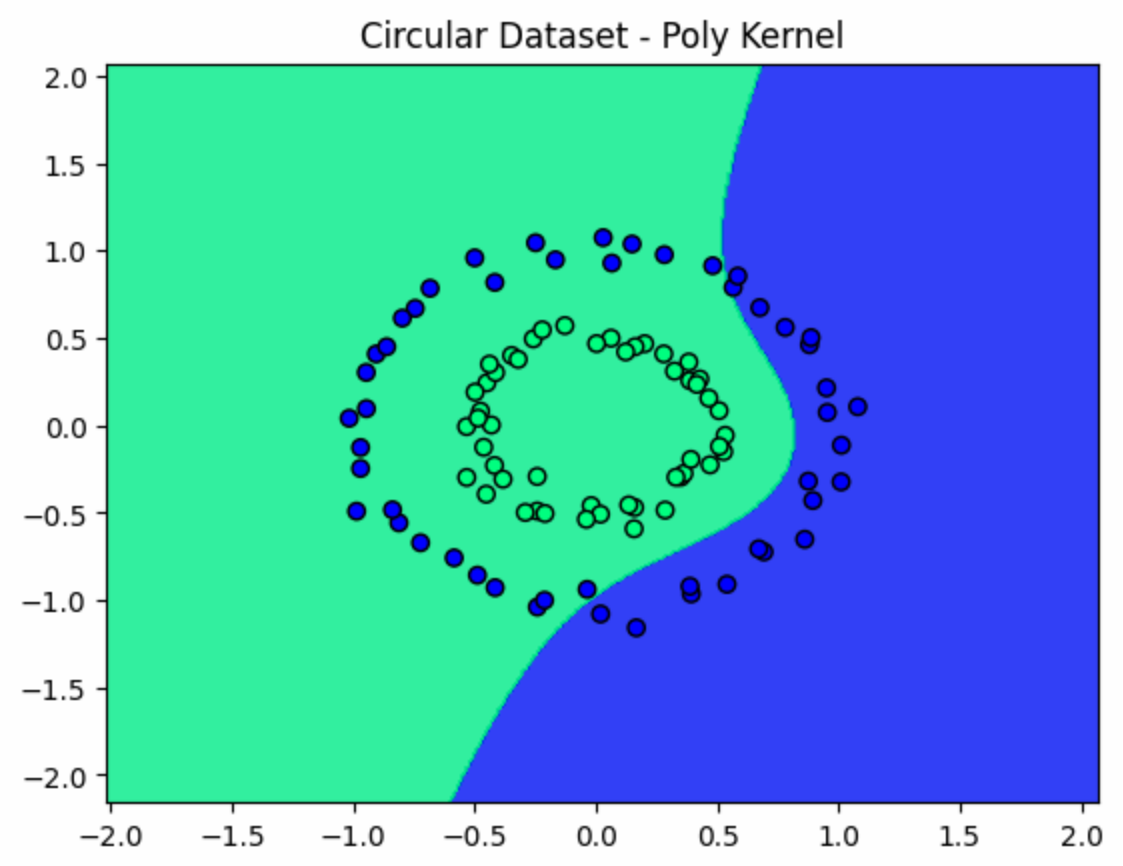

Linear Dataset - Linear Kernel
−2
−4
−6
−8
−10
−12
2
4
6
8
10

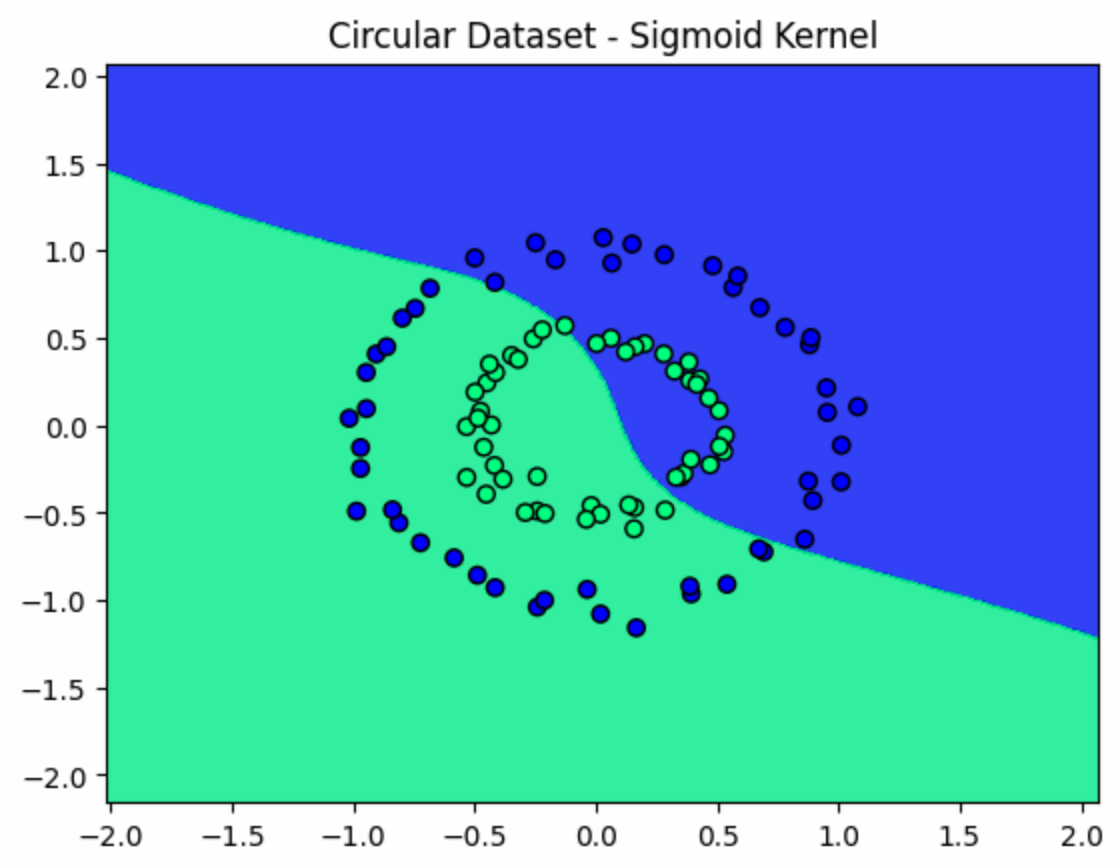

Linear Dataset - Linear Kernel
−2
−4
−6
−8
−10
−12
2
4
6
8
10

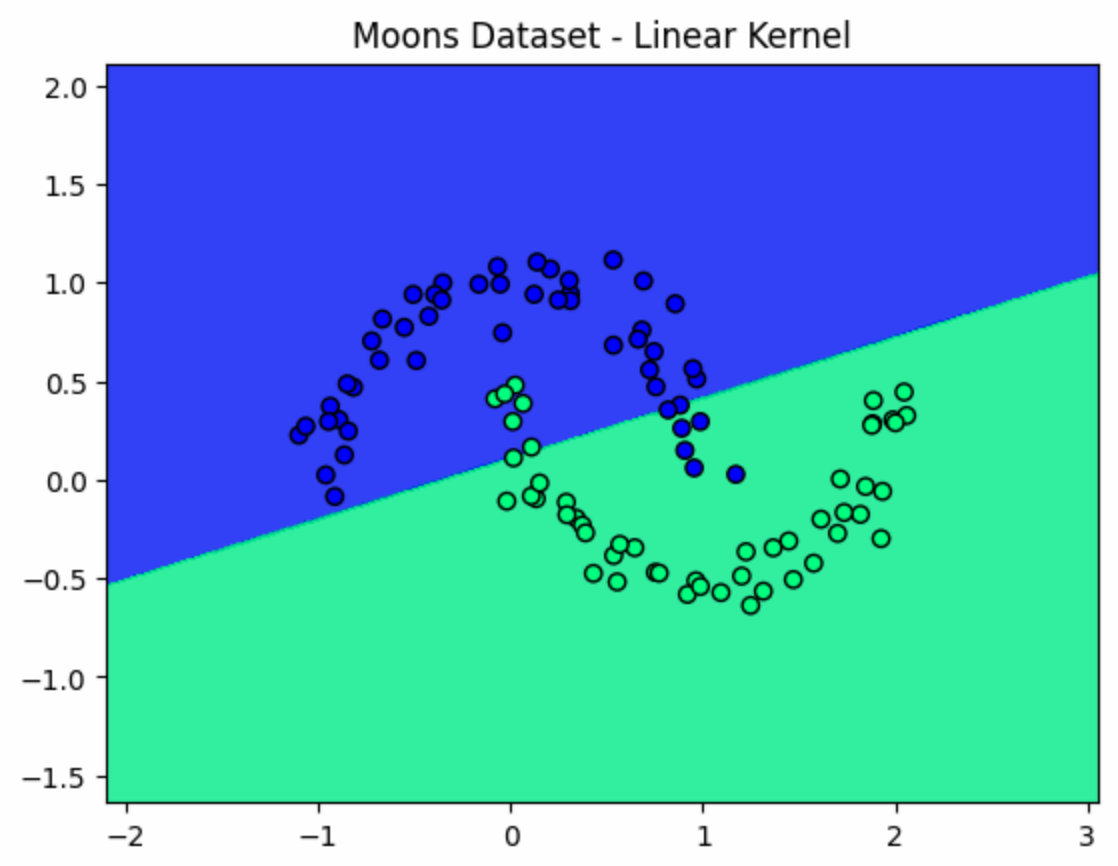

Linear Dataset - Rbf Kernel
−2
−4
−6
−8
−10
−12
2
4
6
8
10

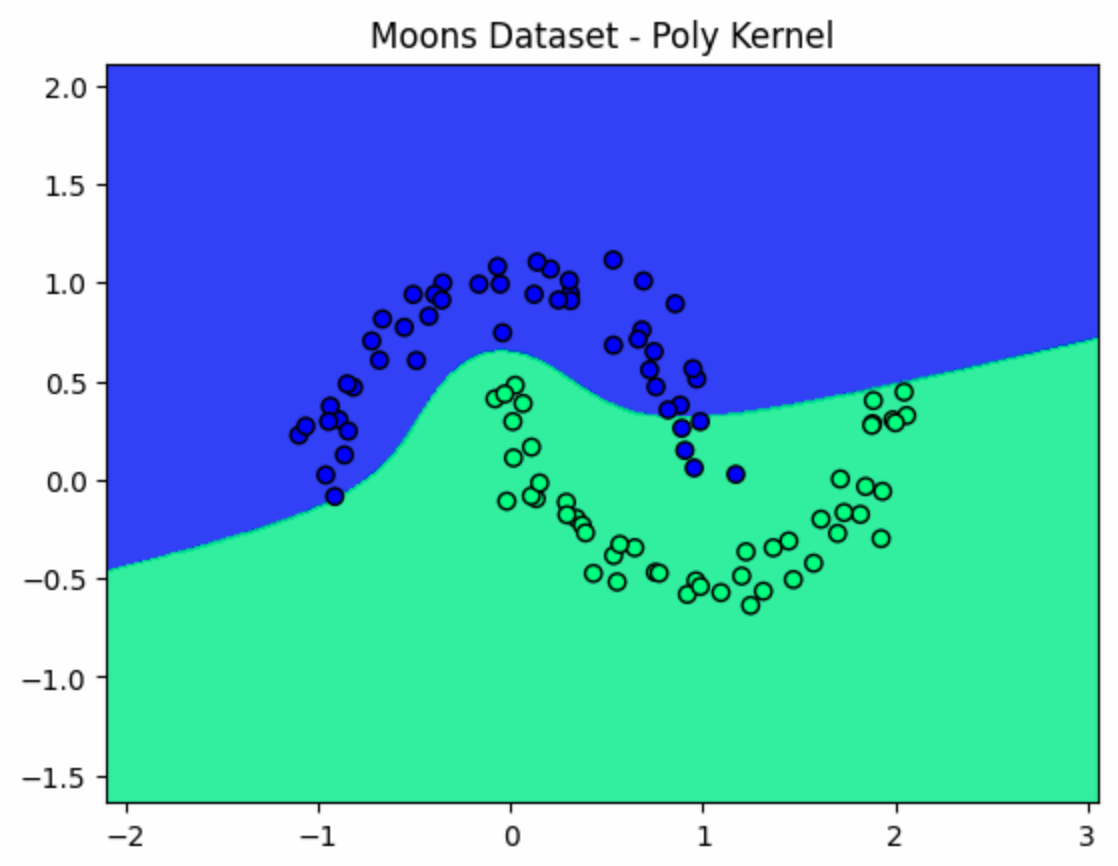

Linear Dataset - Sigmoid Kernel
−2
−4
−6
−8
−10
−12
2
4
6
8
10

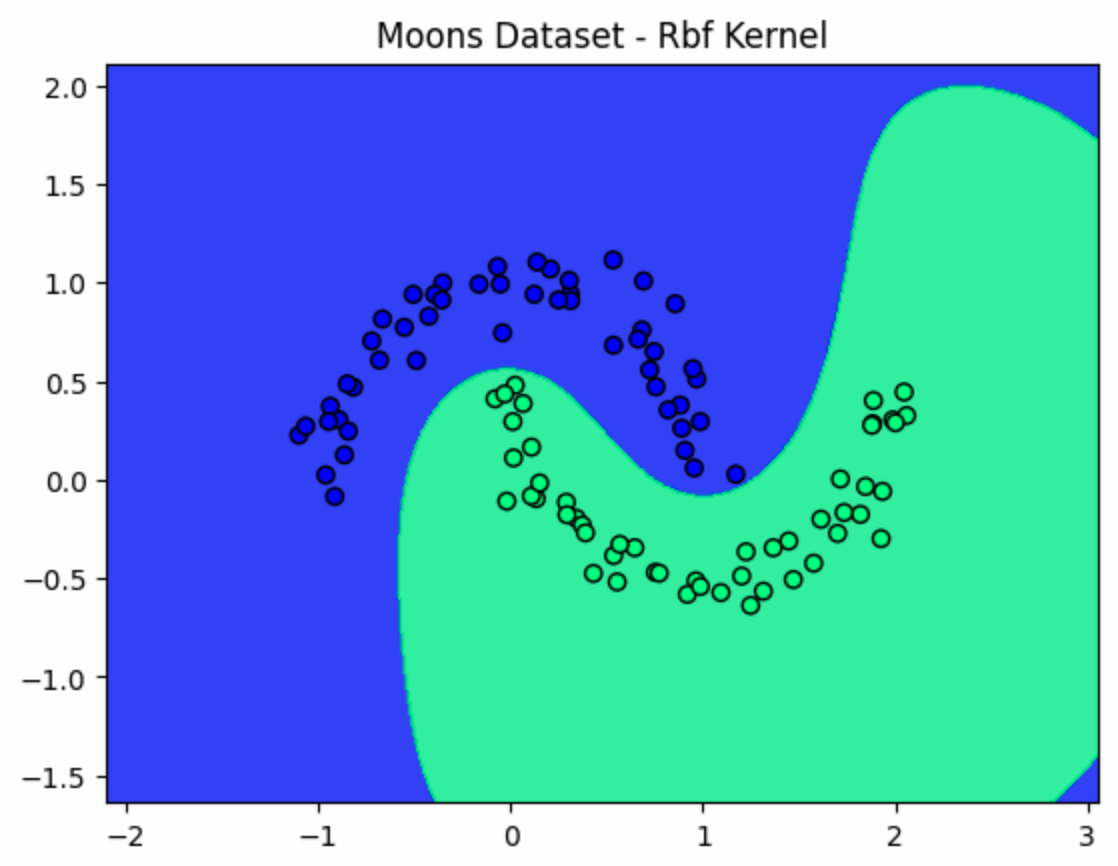

Circular Dataset - Linear Kernel
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0

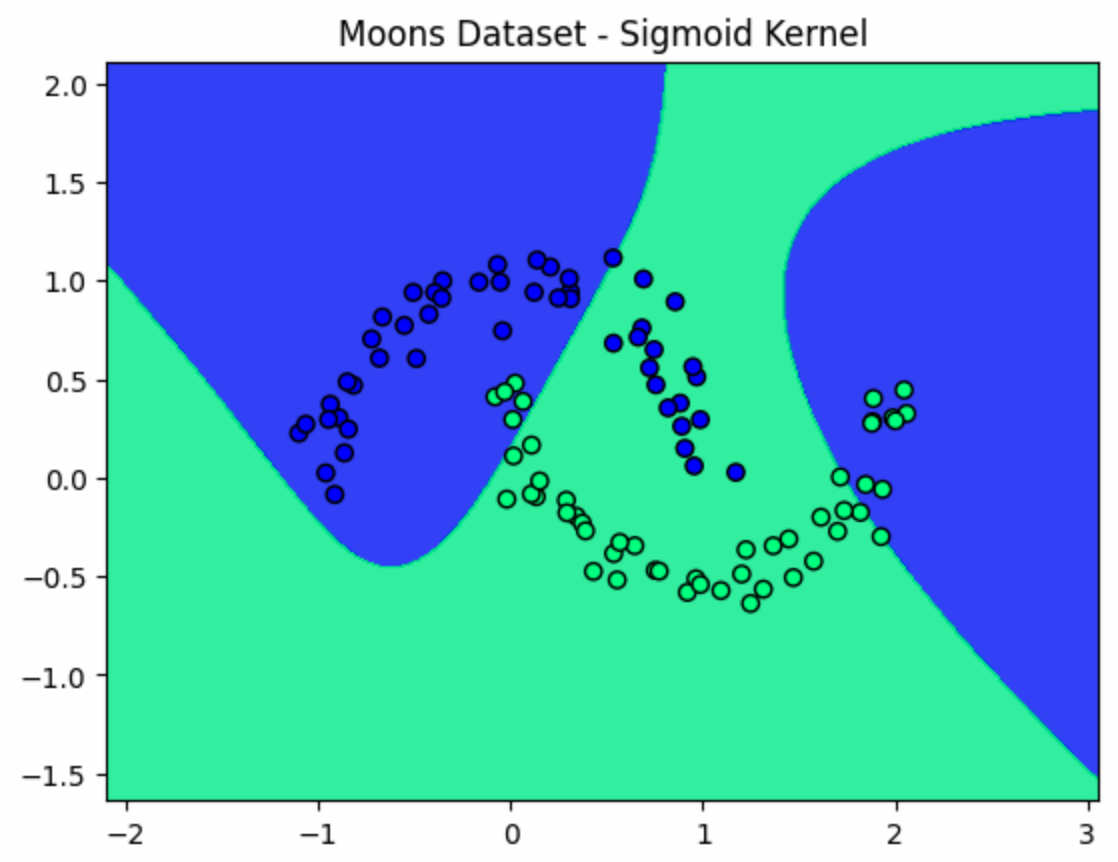

Circular Dataset - Poly Kernel
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0


Circular Dataset - Sigmoid Kernel
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0


Moons Dataset - Linear Kernel
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2
−1
0
1
2
3


Moons Dataset - Poly Kernel
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2
−1
0
1
2
3


Moons Dataset - Rbf Kernel
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2
−1
0
1
2
3



Code using poly kernel on degree 2,3 and 4

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

# Generate a synthetic dataset
X, y = make_blobs(n_samples=200, centers=3, random_state=42,
cluster_std=1.5)

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3, random_state=42)

# Function to plot decision boundaries for non-linear SVM with
polynomial kernel
def plot_poly_svm(X, y, clf, degree):
    """
    Plot decision boundary for a polynomial kernel SVM.
```

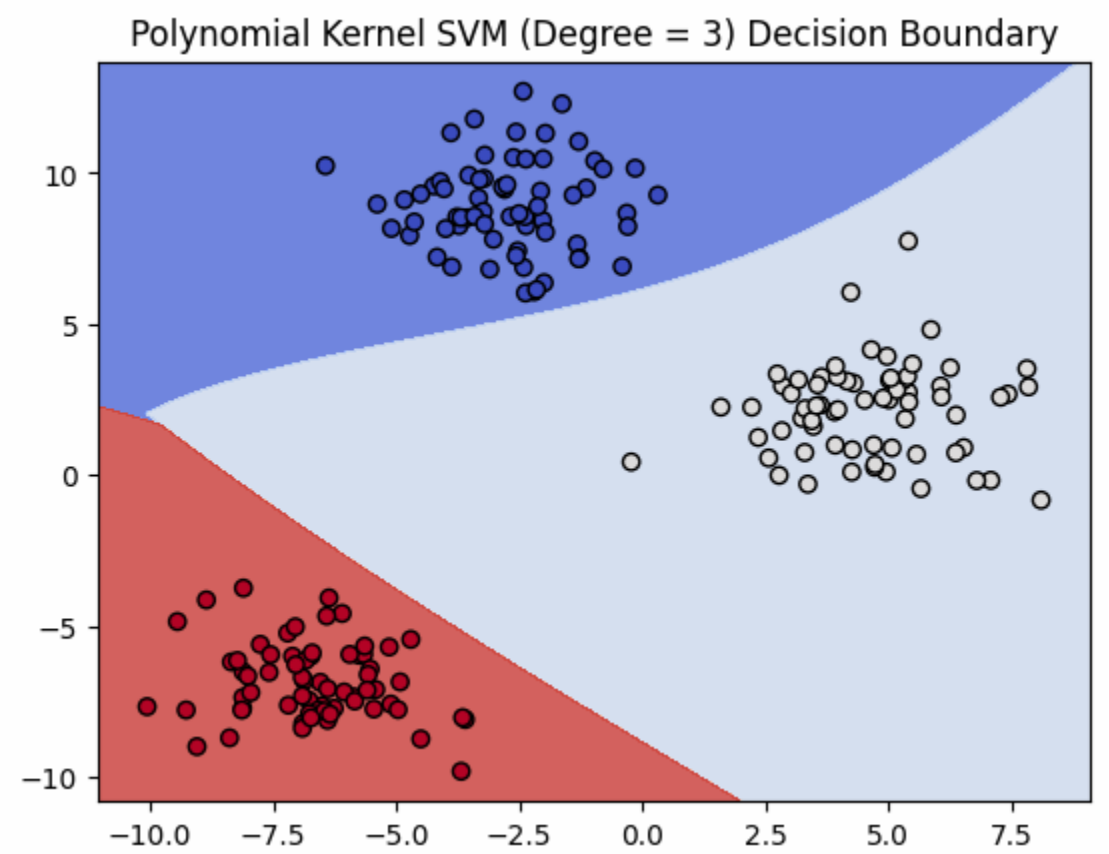
```
    Parameters:
        X (ndarray): Input features, shape (n_samples, 2).
        y (ndarray): Target labels, shape (n_samples,).
        clf (SVC): Trained polynomial kernel SVM classifier.
        degree (int): Degree of the polynomial kernel.
    """
    x_min, x_max = X[:, 0].min() - 1, X[:, 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, 0.01),
                         np.arange(y_min, y_max, 0.01))

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

    plt.contourf(xx, yy, Z, alpha=0.8, cmap=plt.cm.coolwarm)
    plt.scatter(X[:, 0], X[:, 1], c=y, edgecolors='k',
cmap=plt.cm.coolwarm)
    plt.title(f"Polynomial Kernel SVM (Degree = {degree}) Decision
Boundary")
    plt.show()

# Train and evaluate polynomial kernel SVM classifiers with different
degrees
for degree in [2, 3, 4]:
    print(f"\nTraining SVM with Polynomial Kernel (Degree =
{degree})")
    poly_svm = SVC(kernel='poly', degree=degree, C=1, random_state=42)
    # Explanation of parameter C:
    # C is the regularization parameter. A smaller C encourages the
model to find a simpler decision boundary by allowing some
misclassification.
    # A larger C tries to classify all training examples correctly,
potentially leading to overfitting.
    poly_svm.fit(X_train, y_train)

    # Make predictions
    y_pred = poly_svm.predict(X_test)

    # Evaluate the model
    print("Classification Report:")
    print(classification_report(y_test, y_pred))
    print(f"Accuracy: {accuracy_score(y_test, y_pred):.2f}")
```

```
    # Plot the decision boundary
    plot_poly_svm(X, y, poly_svm, degree)
```

Training SVM with Polynomial Kernel (Degree = 3)
Classification Report:

	precision	recall	f1-score	support
0	1.00	1.00	1.00	20
1	1.00	1.00	1.00	20
2	1.00	1.00	1.00	20
accuracy			1.00	60
macro avg	1.00	1.00	1.00	60
weighted avg	1.00	1.00	1.00	60

Accuracy: 1.00

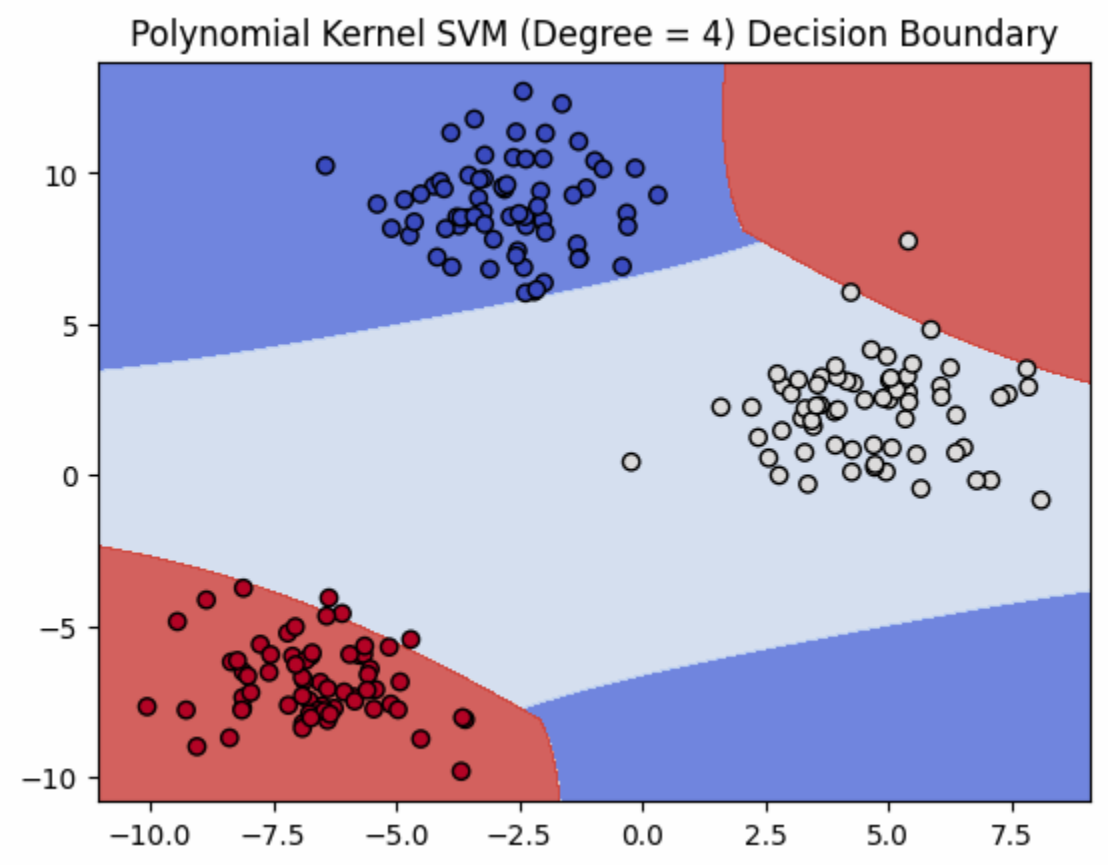


Training SVM with Polynomial Kernel (Degree = 4)
Classification Report:

	precision	recall	f1-score	support
0	1.00	1.00	1.00	20

1	0.95	0.95	0.95	20
2	0.95	0.95	0.95	20
accuracy			0.97	60
macro avg	0.97	0.97	0.97	60
weighted avg	0.97	0.97	0.97	60

Accuracy: 0.97



Code for using rbf kernel on different regularization parameter(c) and kernel coefficient(gamma)

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

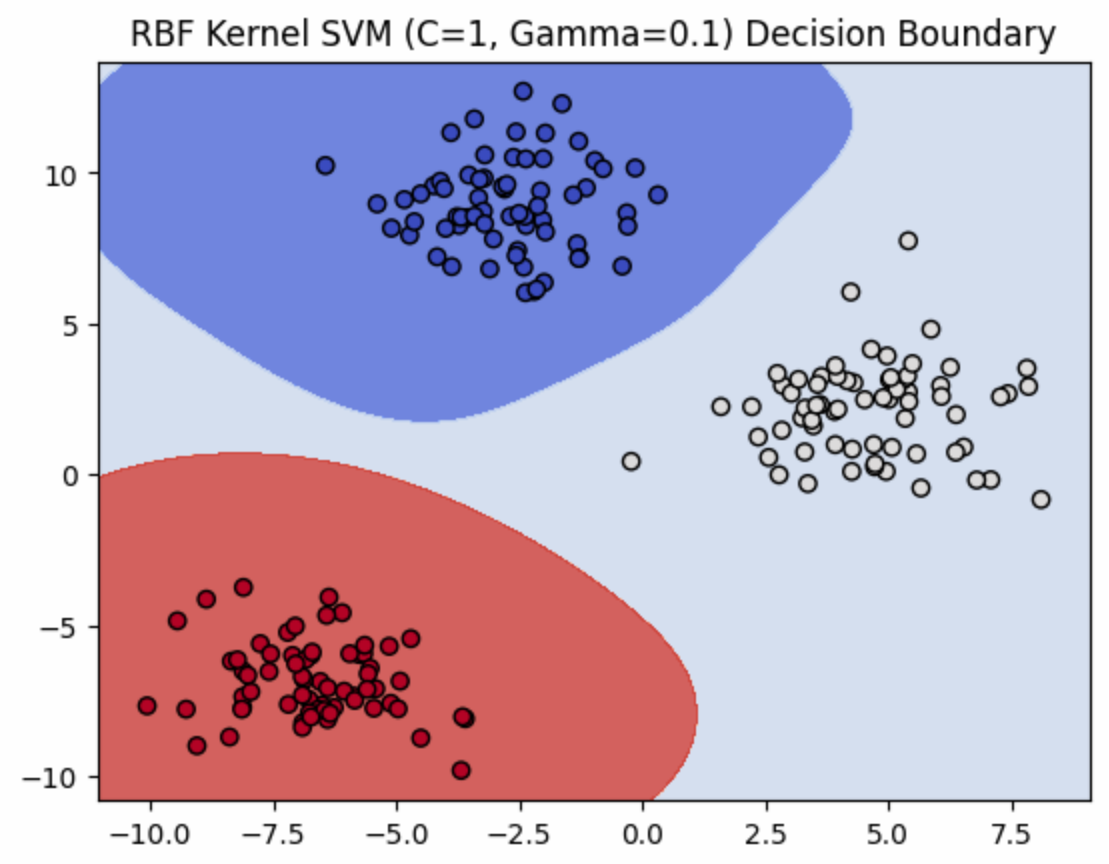
```
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, accuracy_score

# Generate a synthetic dataset
X, y = make_blobs(n_samples=200, centers=3, random_state=42,
cluster_std=1.5)

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

# Function to plot decision boundaries for SVM with RBF kernel
def plot_rbf_svm(X, y, clf, C, gamma):
    """
    Plot decision boundary for an RBF kernel SVM.

    Parameters:
        X (ndarray): Input features, shape (n_samples, 2).
        y (ndarray): Target labels, shape (n_samples,).
        clf (SVC): Trained RBF kernel SVM classifier.
        C (float): Regularization parameter.
        gamma (float): Kernel coefficient.
    """
    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, 0.01),
                         np.arange(y_min, y_max, 0.01))

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

    plt.contourf(xx, yy, Z, alpha=0.8, cmap=plt.cm.coolwarm)
    plt.scatter(X[:, 0], X[:, 1], c=y, edgecolors='k',
cmap=plt.cm.coolwarm)
    plt.title(f"RBF Kernel SVM (C={C}, Gamma={gamma}) Decision Boundary")
    plt.show()

# Train and evaluate RBF kernel SVM classifiers with different parameters
parameters = [(1, 0.1), (1, 1), (10, 0.1), (10, 1)]
```

```
for C, gamma in parameters:
    print(f"\nTraining SVM with RBF Kernel (C={C}, Gamma={gamma})")
    rbf_svm = SVC(kernel='rbf', C=C, gamma=gamma, random_state=42)
    rbf_svm.fit(X_train, y_train)

    # Make predictions
    y_pred = rbf_svm.predict(X_test)

    # Evaluate the model
    print("Classification Report:")
    print(classification_report(y_test, y_pred))
    print(f"Accuracy: {accuracy_score(y_test, y_pred):.2f}")

    # Plot the decision boundary
    plot_rbf_svm(X, y, rbf_svm, C, gamma)
```

Training SVM with RBF Kernel (C=1, Gamma=0.1)
Classification Report:

	precision	recall	f1-score	support
0	1.00	1.00	1.00	20
1	1.00	1.00	1.00	20
2	1.00	1.00	1.00	20
accuracy			1.00	60
macro avg	1.00	1.00	1.00	60
weighted avg	1.00	1.00	1.00	60

Accuracy: 1.00

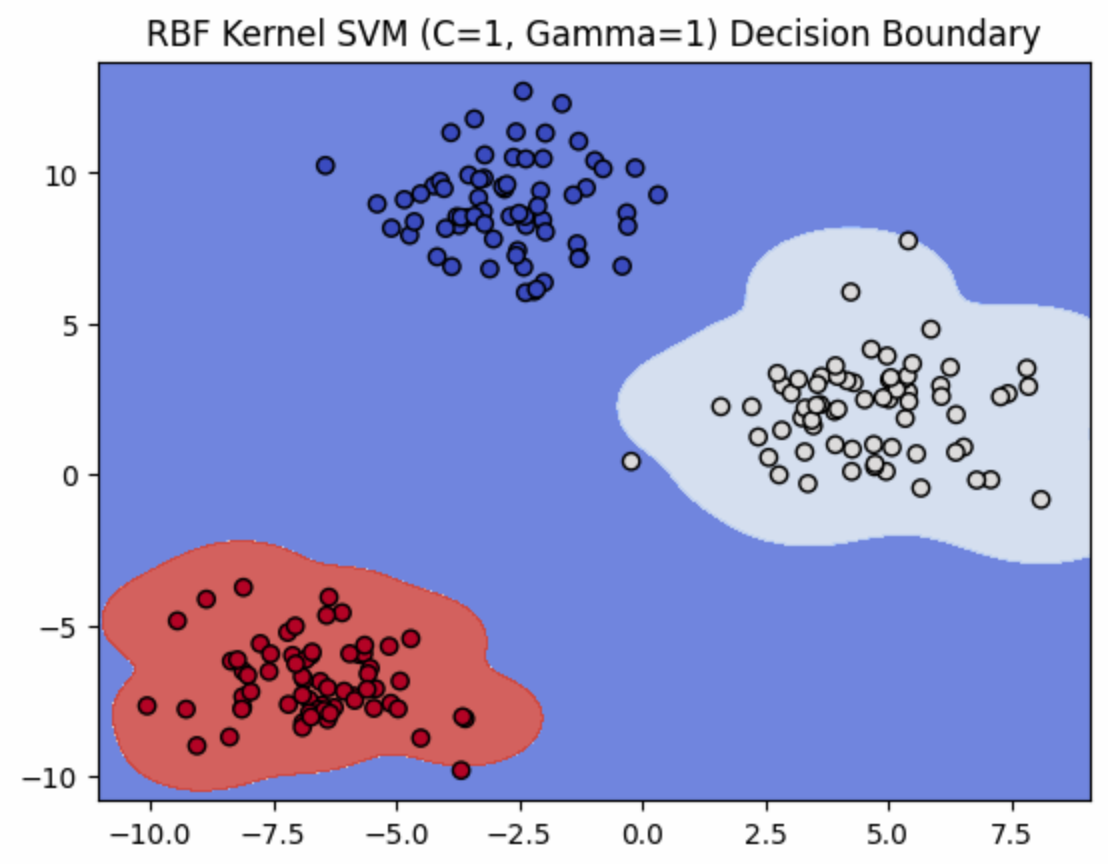


Training SVM with RBF Kernel (C=1, Gamma=1)

Classification Report:

	precision	recall	f1-score	support
0	0.91	1.00	0.95	20
1	1.00	0.95	0.97	20
2	1.00	0.95	0.97	20
accuracy			0.97	60
macro avg	0.97	0.97	0.97	60
weighted avg	0.97	0.97	0.97	60

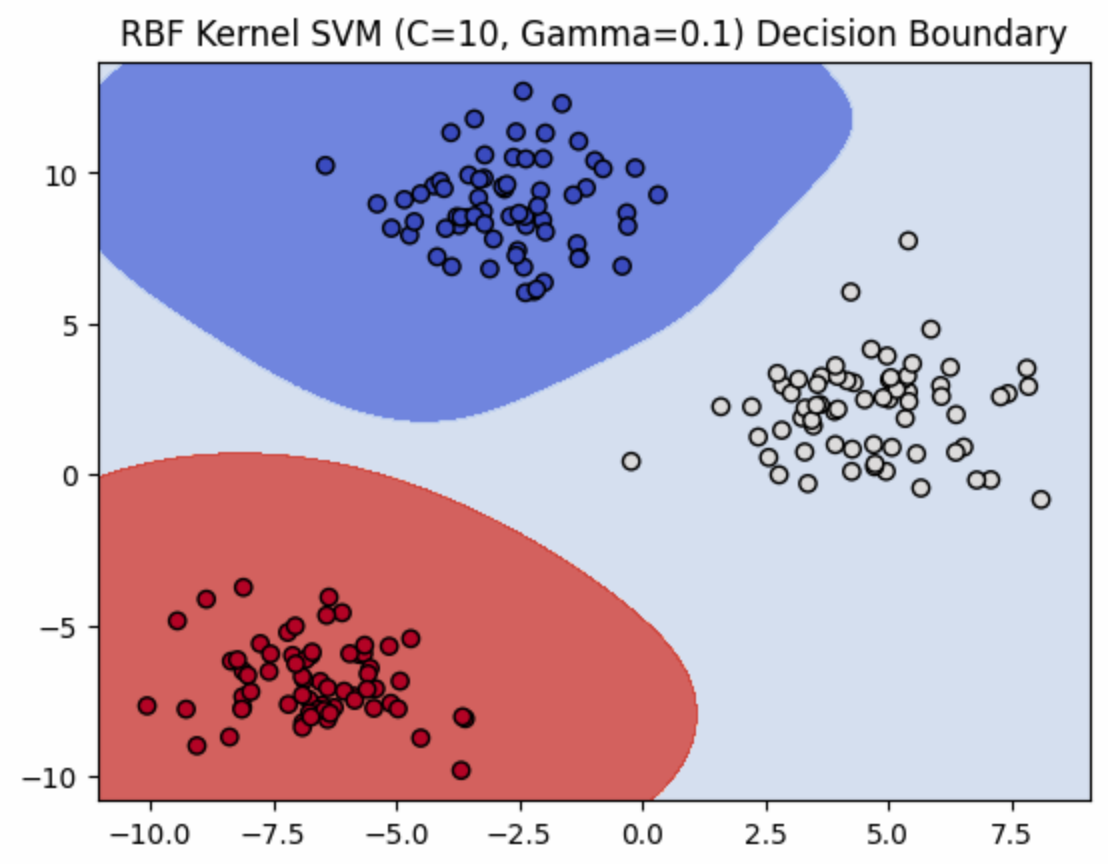
Accuracy: 0.97



Training SVM with RBF Kernel (C=10, Gamma=0.1)
Classification Report:

	precision	recall	f1-score	support
0	1.00	1.00	1.00	20
1	1.00	1.00	1.00	20
2	1.00	1.00	1.00	20
accuracy			1.00	60
macro avg	1.00	1.00	1.00	60
weighted avg	1.00	1.00	1.00	60

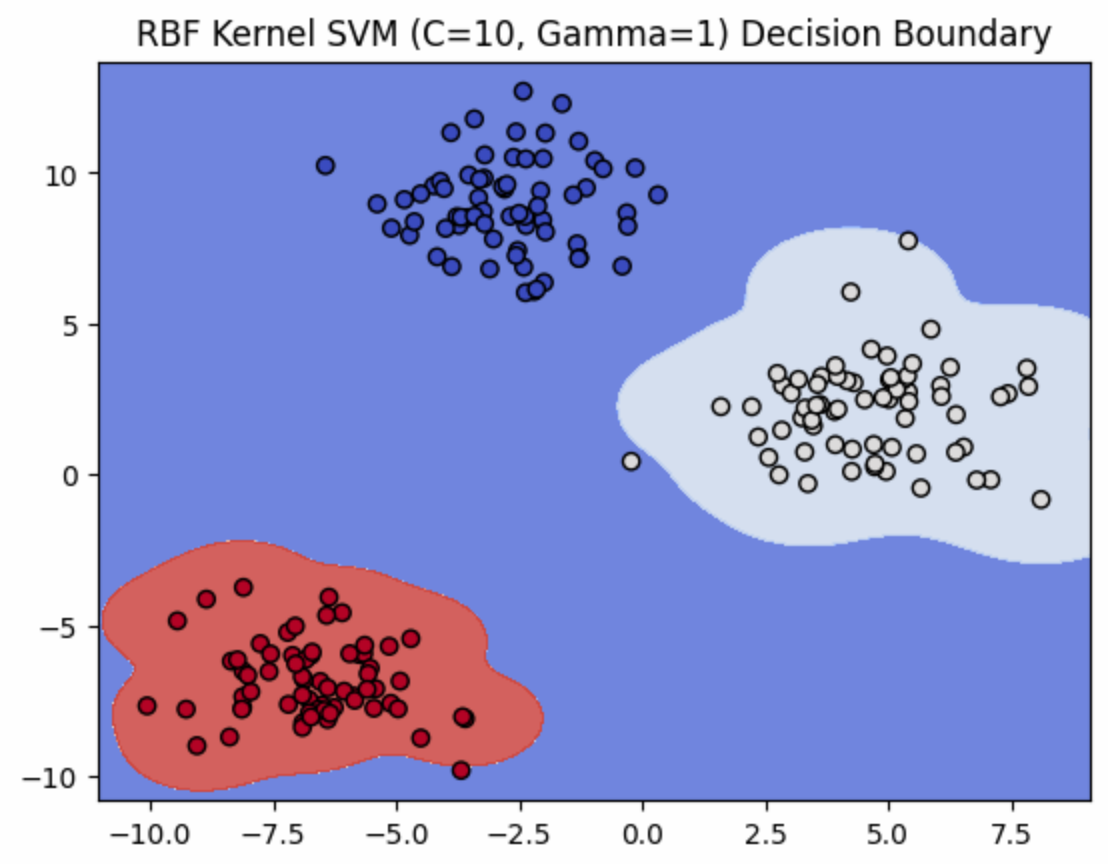
Accuracy: 1.00



Training SVM with RBF Kernel (C=10, Gamma=1)
Classification Report:

	precision	recall	f1-score	support
0	0.91	1.00	0.95	20
1	1.00	0.95	0.97	20
2	1.00	0.95	0.97	20
accuracy			0.97	60
macro avg	0.97	0.97	0.97	60
weighted avg	0.97	0.97	0.97	60

Accuracy: 0.97



Multiclass Classification Using SVM

SVM is originally designed to classify data points into two groups (binary classification). However, it can be adapted for problems involving multiple classes by breaking the multiclass problem into multiple binary classification tasks. The two most common approaches are:

1. One-to-Rest (OvR) Approach

- **Concept**: Train one binary classifier for each class, where the goal is to distinguish between that class and all the other classes combined.
- **Process**:
    - For each class, create a separate classifier. For example, if there are three classes (A, B, C), the OvR approach trains three classifiers:

        - One classifier to distinguish A from B and C.
        - One classifier to distinguish B from A and C.
        - One classifier to distinguish C from A and B.
    - For a new data point, all classifiers are evaluated, and the one with the highest confidence or score determines the class.

Process of Classifying a New Data Point

1. **Evaluate Each Classifier**:
    - In OvR, each classifier is trained to separate one class from all other classes. When given a new data point, each classifier outputs a score or confidence value indicating how likely it is that the data point belongs to the specific class it represents.
2. **Output Scores**:
    - Each classifier produces a numerical value, often referred to as the "decision score." This score represents the distance of the data point from the hyperplane (boundary). A higher positive score indicates stronger confidence that the data point belongs to that class.
3. **Choose the Class with the Highest Score**:
    - Once all classifiers have been evaluated, the class corresponding to the classifier with the highest score is assigned to the new data point. For example:
        - Classifier A: Score = 2.5
        - Classifier B: Score = 1.2
        - Classifier C: Score = 0.8
        - Since Classifier A has the highest score, the data point is classified as belonging to Class A.

- **Pros**:
    - Simple to implement.
    - Requires fewer models, so it is computationally less expensive.
- **Cons**:
    - Can struggle if there is an imbalance between the classes.
    - Results may be less accurate for some datasets.

2. One-to-One (OvO) Approach

- **Concept**: Train a binary classifier for every possible pair of classes. Each classifier focuses only on distinguishing between two specific classes.
- **Process**:
    - For a dataset with three classes (A, B, C), the OvO approach trains classifiers for:
        - A vs. B
        - A vs. C
        - B vs. C
    - For a new data point, each classifier gives a "vote" for one of its two classes. The class with the most votes across all classifiers is the predicted class.
- **Pros**:
    - More focused classifiers since each model only deals with two classes.
    - Generally performs better when class boundaries are complex.
- **Cons**:
    - Computationally expensive since it requires many classifiers.
    - Can be slower during prediction because every classifier needs to evaluate the data.

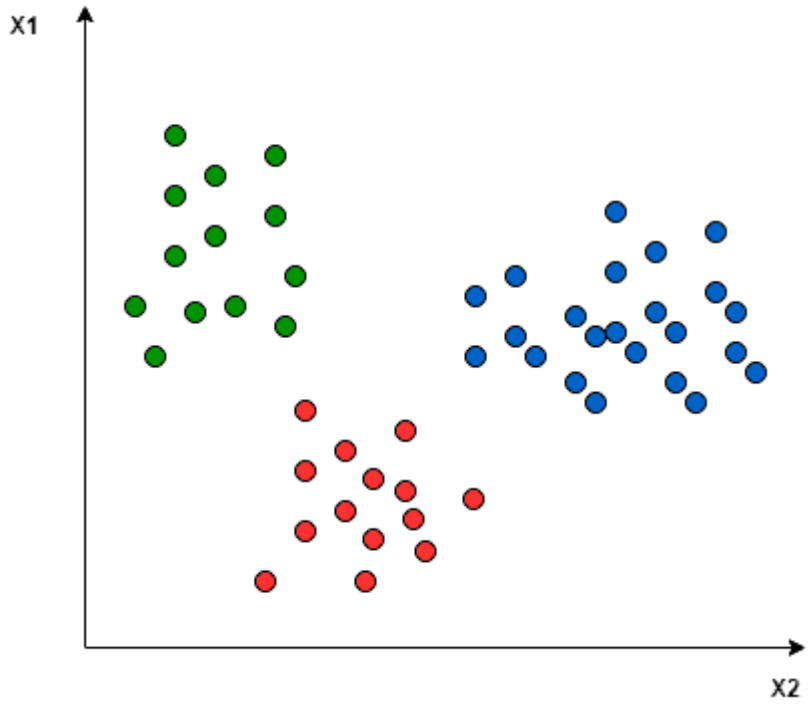
Comparison of OvR and OvO

- **OvR** is better suited for datasets with a large number of classes because it requires fewer models and is faster.
- **OvO** is often more accurate, especially when the classes are not well-separated, but it is computationally intensive and best for datasets with fewer classes.

Which to Use?

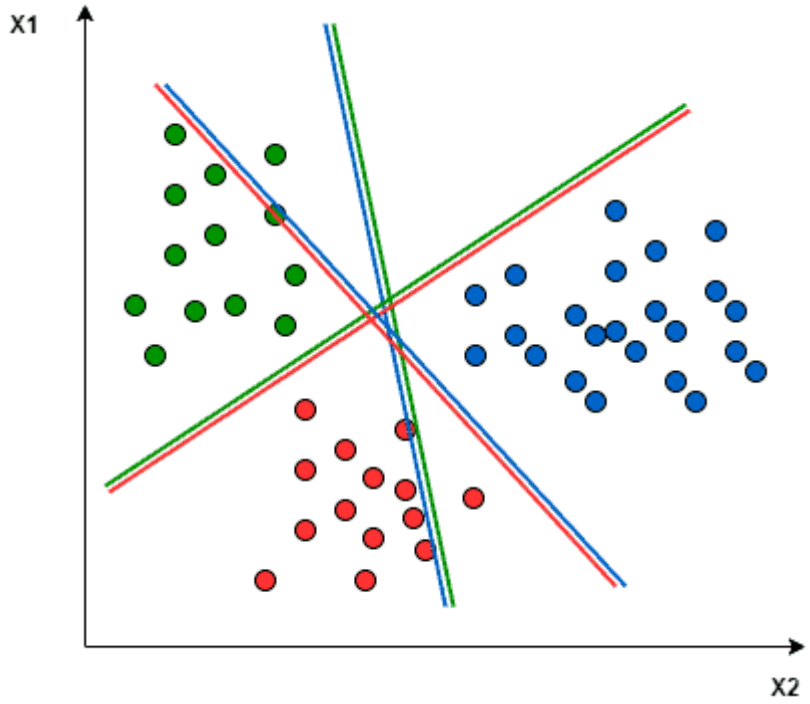
- Use **OvR** when computational efficiency is a priority or the dataset has many classes.
- Use **OvO** when accuracy is more important, or when the dataset has fewer, more distinct classes.
-

Let’s take an example of 3 classes classification problem; green, red, and blue, as the following image:

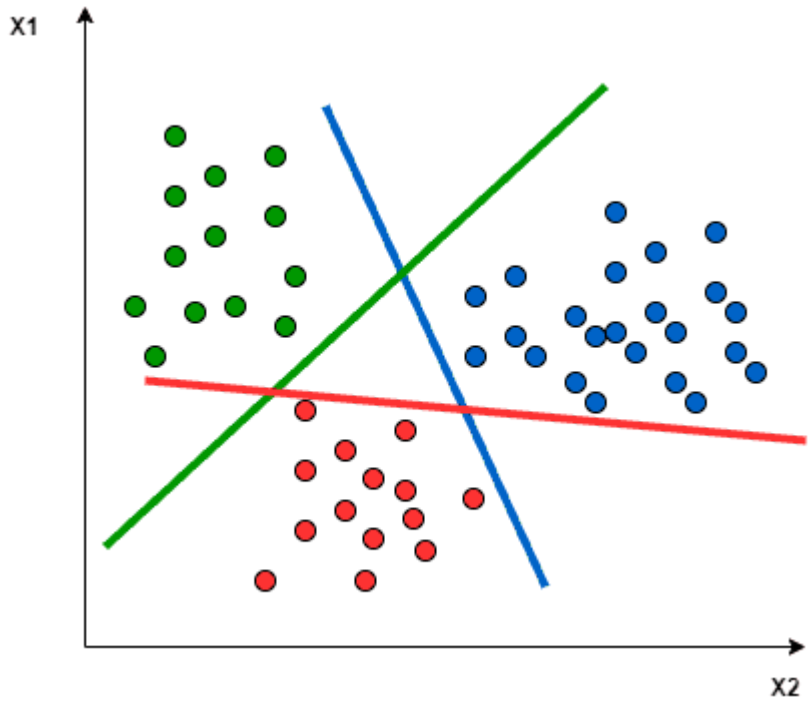


Applying the two approaches to this data set results in the followings:

In the *One-to-One* approach, we need a hyperplane to separate between every two classes, neglecting the points of the third class. This means the separation takes into account only the points of the two classes in the current split. For example, the red-blue line tries to maximize the separation only between blue and red points. It has nothing to do with green points:



In the *One-to-Rest* approach, we need a hyperplane to separate between a class and all others at once. This means the separation takes all points into account, dividing them into two groups; a group for the class points and a group for all other points. For example, the green line tries to maximize the separation between green points and all other points at once:



One of the most common real-world problems for multiclass classification using SVM is text classification. For example, classifying news articles, tweets, or scientific papers.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.svm import SVC
from sklearn.datasets import make_classification
from sklearn.multiclass import OneVsOneClassifier, OneVsRestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report

# Generate a synthetic multi-class dataset
X, y = make_classification(n_samples=300, n_features=2,
n_informative=2, n_redundant=0,
                           n_classes=3, n_clusters_per_class=1,
random_state=42)

# Split the dataset into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3, random_state=42)

# Function to plot decision boundaries for multi-class classification
def plot_decision_boundary(X, y, clf, title):
    """
```

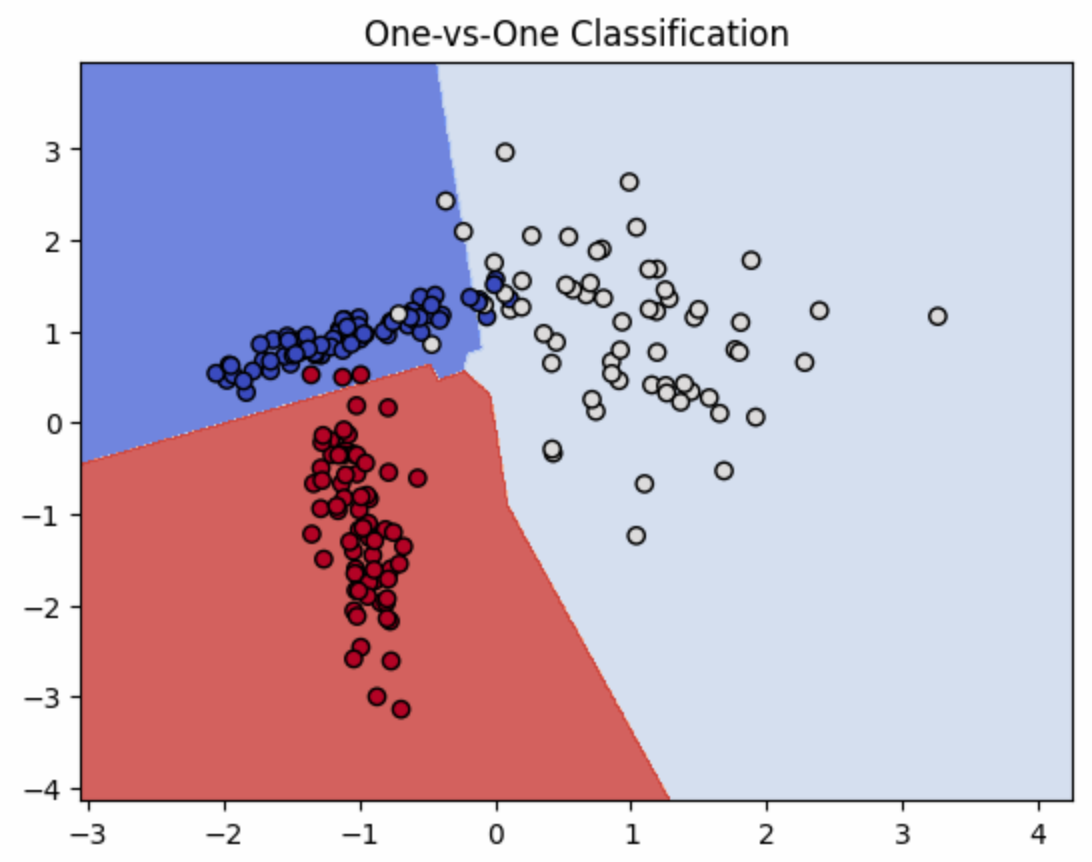
```
    Plot decision boundary for a classifier.

    Parameters:
        X (ndarray): Input features, shape (n_samples, 2).
        y (ndarray): Target labels, shape (n_samples,).
        clf: Trained classifier.
        title (str): Title for the plot.
    """
    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, 0.01),
                         np.arange(y_min, y_max, 0.01))

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

    plt.contourf(xx, yy, Z, alpha=0.8, cmap=plt.cm.coolwarm)
    plt.scatter(X[:, 0], X[:, 1], c=y, edgecolors='k',
cmap=plt.cm.coolwarm)
    plt.title(title)
    plt.show()

# One-vs-One (OvO) Classification
ovo_clf = OneVsOneClassifier(SVC(kernel='linear', C=1,
random_state=42))
ovo_clf.fit(X_train, y_train)
y_pred_ovo = ovo_clf.predict(X_test)

# Evaluate One-vs-One model
print("One-vs-One Classification Report:")
print(classification_report(y_test, y_pred_ovo))

# Plot decision boundary for One-vs-One
plot_decision_boundary(X_train, y_train, ovo_clf, "One-vs-One
Classification")

# One-vs-All (OvA) Classification
ova_clf = OneVsRestClassifier(SVC(kernel='linear', C=1,
random_state=42))
ova_clf.fit(X_train, y_train)
y_pred_ova = ova_clf.predict(X_test) # Changed nova_clf to ova_clf

# Evaluate One-vs-All model
print("One-vs-All Classification Report:")
```

```
print(classification_report(y_test, y_pred_ova))

# Plot decision boundary for One-vs-All
plot_decision_boundary(X_train, y_train, ova_clf, "One-vs-All
Classification")
```

Output

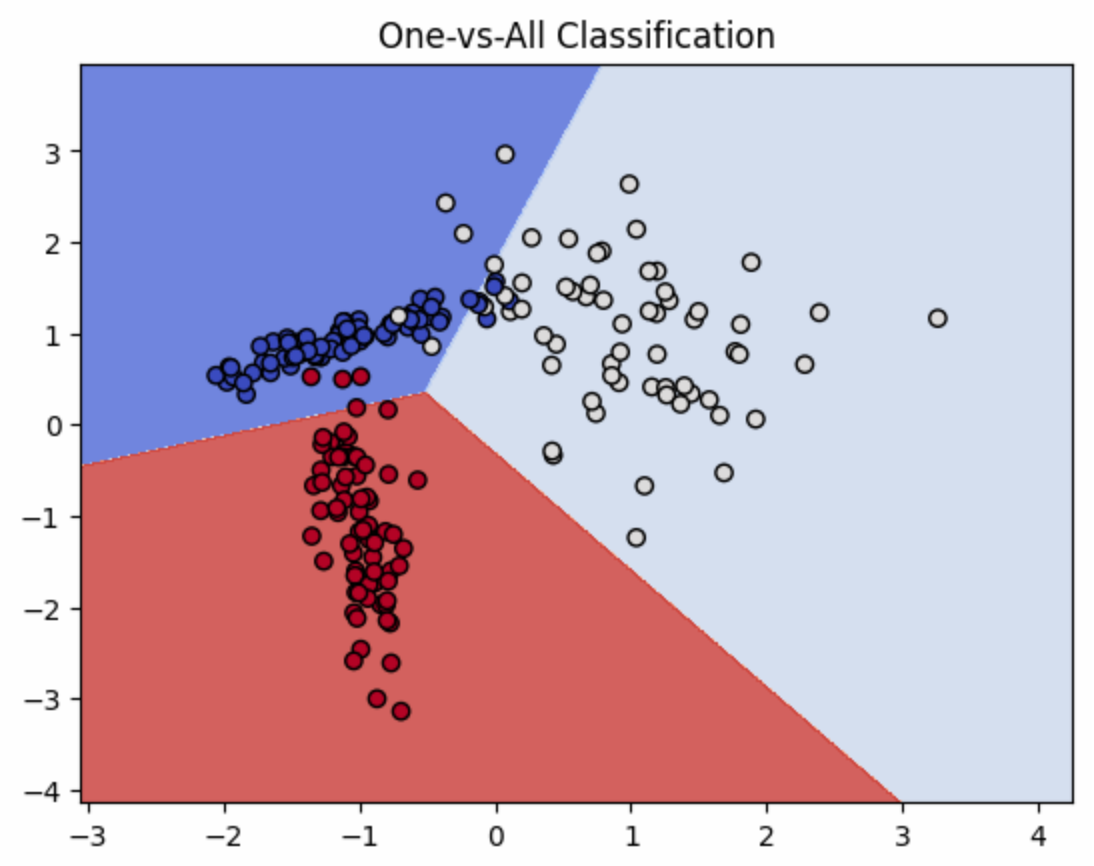
One-vs-One Classification Report:

	precision	recall	f1-score	support
0	0.77	0.89	0.83	27
1	0.92	0.89	0.91	38
2	1.00	0.88	0.94	25
accuracy			0.89	90
macro avg	0.90	0.89	0.89	90
weighted avg	0.90	0.89	0.89	90



One-vs-All Classification Report:

	precision	recall	f1-score	support
0	0.76	0.93	0.83	27
1	0.94	0.89	0.92	38
2	1.00	0.84	0.91	25
accuracy			0.89	90
macro avg	0.90	0.89	0.89	90
weighted avg	0.90	0.89	0.89	90


One-vs-All Classification
3
2
1
0
−1
−2
−3
−4
−3
−2
−1
0
1
2
3
4