

Question 1

Loading required libraries

```
In [1]: import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_moons, make_circles
from sklearn.preprocessing import StandardScaler

In [2]: # Set random seed for reproducibility
np.random.seed(42)

In [3]: # Function to generate half-circle dataset
def make_half_circle(n_samples, noise=0.05, factor=0.5):
    """Ensures each half-circle has exactly `n_samples` points."""
    while True:
        X, y = make_circles(n_samples=n_samples * 3, noise=noise, factor=factor)

        # Get upper and lower half-circle
        X_upper = X[X[:, 1] >= 0]
        X_lower = X[X[:, 1] <= 0]

        # Ensure we have enough points
        if len(X_upper) >= n_samples and len(X_lower) >= n_samples:
            X_upper = X_upper[:n_samples]
            X_lower = X_lower[:n_samples]
            break # Stop once we have exact samples

    # Assign labels: Upper = 0, Lower = 1
    y_upper = np.zeros(len(X_upper))
    y_lower = np.ones(len(X_lower))

    # Combine both halves into one dataset
    X_half_circle = np.vstack((X_upper, X_lower))
    y_half_circle = np.hstack((y_upper, y_lower))

    return X_half_circle, y_half_circle
```

Generating Datasets

```
In [4]: datasets = {
    "Half-Circles (1000)": make_half_circle(n_samples=500), # 500 per class
    "Half-Circles (10000)": make_half_circle(n_samples=5000), # 5000 per class
    "Moons (1000)": make_moons(n_samples=1000, noise=0.1, random_state=42),
    "Moons (10000)": make_moons(n_samples=10000, noise=0.1, random_state=42),
}

In [5]: # Standardizing each dataset after generation
scaler = StandardScaler()
for key in datasets.keys():
    X, y = datasets[key]
```

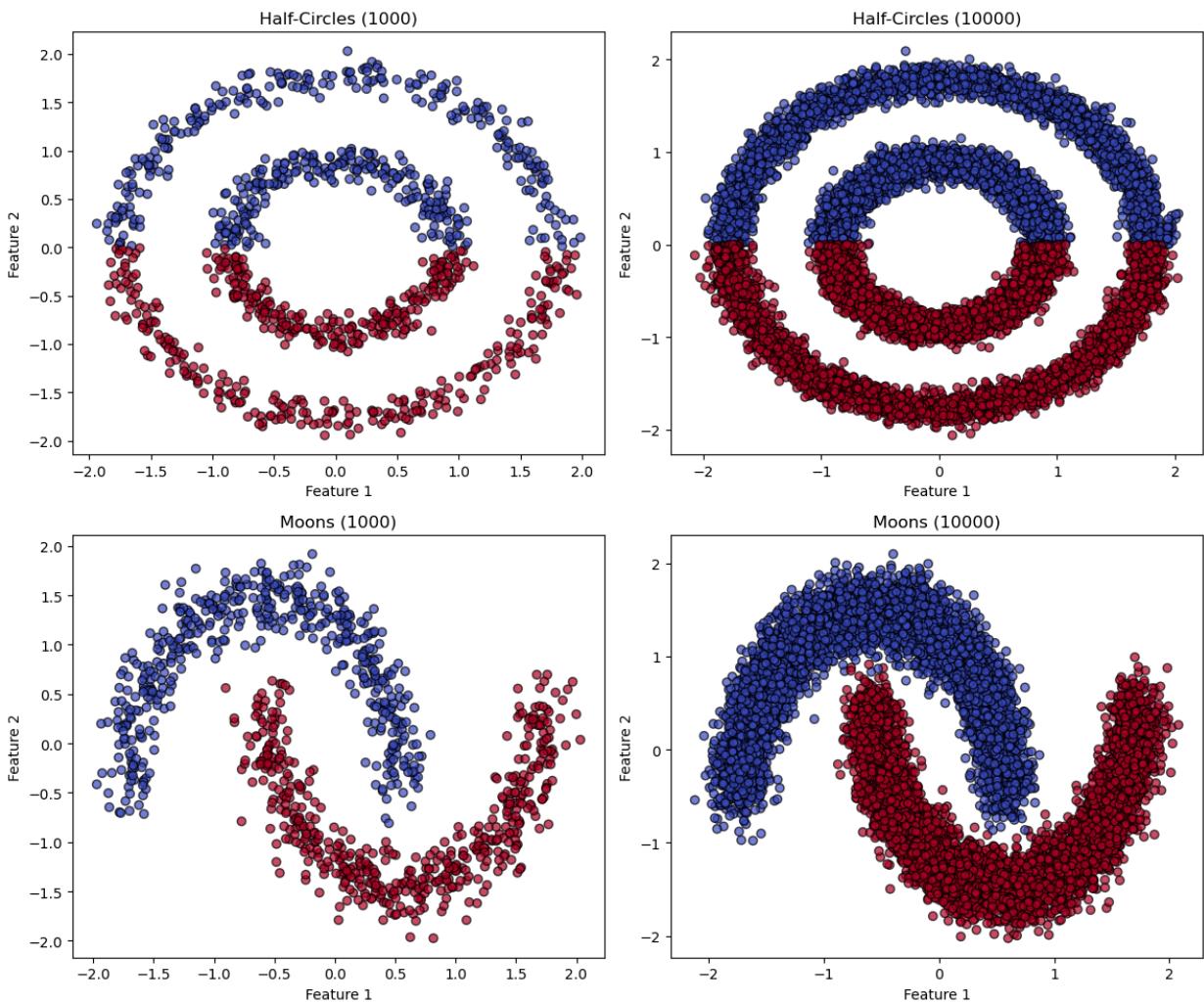
```
X = scaler.fit_transform(X) # Standardizing the features
datasets[key] = (X, y) # Updating dataset with standardized values
```

Visualization of the datasets

```
In [6]: fig, axes = plt.subplots(2, 2, figsize=(12, 10))

for ax, (title, (X, y)) in zip(axes.ravel(), datasets.items()):
    ax.scatter(X[:, 0], X[:, 1], c=y, cmap='coolwarm', edgecolors='k', alpha=0.8)
    ax.set_title(title)
    ax.set_xlabel("Feature 1")
    ax.set_ylabel("Feature 2")

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



Splitting data for 5-fold cross validation

```
In [7]: from sklearn.model_selection import KFold

# Defining k for k-fold cross-validation
k = 5

# Creating a generalizable k-fold cross-validation setup
```

```

kf = KFold(n_splits=k, shuffle=True, random_state=42)

# Function to split dataset into k-folds
def get_kfold_splits(X, y, kf):
    """Returns indices for k-fold cross-validation splits."""
    folds = []
    for train_idx, test_idx in kf.split(X):
        folds.append((train_idx, test_idx))
    return folds

# Generate k-fold splits for each dataset
kfold_splits = {name: get_kfold_splits(X, y, kf) for name, (X, y) in datasets...}

```

Displaying 5 folds splits for datasets

- Moons (1000) Dataset

```

In [8]: # Displaying 5 folds for Moons dataset
dataset_name1 = "Moons (1000)"
example_folds = kfold_splits[dataset_name1]

print(f"Example splits for {dataset_name1}:\n")
for i, (train_idx, test_idx) in enumerate(example_folds[:5]): # Show 5 folds
    print(f"Fold {i+1}:")
    print(f" Train indices: {train_idx[:10]}... (total {len(train_idx)})")
    print(f" Test indices: {test_idx[:10]}... (total {len(test_idx)})\n")

```

Example splits for Moons (1000):

Fold 1:

```

Train indices: [0 1 2 3 4 5 6 7 8 9]... (total 800)
Test indices: [10 23 25 30 39 44 54 55 59 60]... (total 200)

```

Fold 2:

```

Train indices: [ 1  4  8 10 11 12 13 14 15 16]... (total 800)
Test indices: [ 0  2  3  5  6  7  9 28 29 31]... (total 200)

```

Fold 3:

```

Train indices: [0 1 2 3 4 5 6 7 8 9]... (total 800)
Test indices: [11 12 15 18 19 22 24 38 42 43]... (total 200)

```

Fold 4:

```

Train indices: [ 0  1  2  3  4  5  6  7  9 10]... (total 800)
Test indices: [ 8  16  17  26  36  37  45  48  53 103]... (total 200)

```

Fold 5:

```

Train indices: [ 0  2  3  5  6  7  8  9 10 11]... (total 800)
Test indices: [ 1  4 13 14 20 21 27 32 34 35]... (total 200)

```

- Moons (10000) Dataset

```

In [9]: # Displaying 5 folds for Moons dataset
dataset_name4 = "Moons (10000)"
example_folds = kfold_splits[dataset_name4]

```

```

print(f"Example splits for {dataset_name4}:\n")
for i, (train_idx, test_idx) in enumerate(example_folds[:5]): # Show 5 folds
    print(f"Fold {i+1}:")
    print(f" Train indices: {train_idx[:10]}... (total {len(train_idx)})")
    print(f" Test indices: {test_idx[:10]}... (total {len(test_idx)})\n")

```

Example splits for Moons (10000):

Fold 1:

```

Train indices: [ 1  2  4  5  6  7  9 11 13 15]... (total 8000)
Test indices: [ 0  3  8 10 12 14 17 19 20 23]... (total 2000)

```

Fold 2:

```

Train indices: [0  1  2  3  4  5  6  7  8  9]... (total 8000)
Test indices: [26 27 30 41 45 46 48 51 56 57]... (total 2000)

```

Fold 3:

```

Train indices: [ 0  1  3  4  5  6  7  8  9 10]... (total 8000)
Test indices: [ 2 15 28 34 42 43 44 49 53 62]... (total 2000)

```

Fold 4:

```

Train indices: [ 0  1  2  3  4  5  8  9 10 11]... (total 8000)
Test indices: [ 6  7 16 18 22 24 25 32 40 52]... (total 2000)

```

Fold 5:

```

Train indices: [ 0  2  3  6  7  8 10 12 14 15]... (total 8000)
Test indices: [ 1  4  5  9 11 13 21 38 54 55]... (total 2000)

```

- Half-Circles (10000) Dataset

In [10]:

```

# Displaying 5 folds for Half-Circles dataset
dataset_name2 = "Half-Circles (10000)"
example_folds = kfold_splits[dataset_name2]

print(f"Example splits for {dataset_name2}:\n")
for i, (train_idx, test_idx) in enumerate(example_folds[:5]): # Show 5 folds
    print(f"Fold {i+1}:")
    print(f" Train indices: {train_idx[:10]}... (total {len(train_idx)})")
    print(f" Test indices: {test_idx[:10]}... (total {len(test_idx)})\n")

```

Example splits for Half-Circles (1000):

Fold 1:

Train indices: [1 2 4 5 6 7 9 11 13 15]... (total 8000)
Test indices: [0 3 8 10 12 14 17 19 20 23]... (total 2000)

Fold 2:

Train indices: [0 1 2 3 4 5 6 7 8 9]... (total 8000)
Test indices: [26 27 30 41 45 46 48 51 56 57]... (total 2000)

Fold 3:

Train indices: [0 1 3 4 5 6 7 8 9 10]... (total 8000)
Test indices: [2 15 28 34 42 43 44 49 53 62]... (total 2000)

Fold 4:

Train indices: [0 1 2 3 4 5 8 9 10 11]... (total 8000)
Test indices: [6 7 16 18 22 24 25 32 40 52]... (total 2000)

Fold 5:

Train indices: [0 2 3 6 7 8 10 12 14 15]... (total 8000)
Test indices: [1 4 5 9 11 13 21 38 54 55]... (total 2000)

Half-Circles (1000) Dataset

In [11]:

```
# Displaying 5 folds for Half-Circles dataset
dataset_name3 = "Half-Circles (1000)"
example_folds = kfold_splits[dataset_name3]

print(f"Example splits for {dataset_name3}:\n")
for i, (train_idx, test_idx) in enumerate(example_folds[:5]): # Show 5 folds
    print(f"Fold {i+1}:")
    print(f" Train indices: {train_idx[:10]}... (total {len(train_idx)})")
    print(f" Test indices: {test_idx[:10]}... (total {len(test_idx)})\n")
```

Example splits for Half-Circles (1000):

Fold 1:

Train indices: [0 1 2 3 4 5 6 7 8 9]... (total 800)
Test indices: [10 23 25 30 39 44 54 55 59 60]... (total 200)

Fold 2:

Train indices: [1 4 8 10 11 12 13 14 15 16]... (total 800)
Test indices: [0 2 3 5 6 7 9 28 29 31]... (total 200)

Fold 3:

Train indices: [0 1 2 3 4 5 6 7 8 9]... (total 800)
Test indices: [11 12 15 18 19 22 24 38 42 43]... (total 200)

Fold 4:

Train indices: [0 1 2 3 4 5 6 7 9 10]... (total 800)
Test indices: [8 16 17 26 36 37 45 48 53 103]... (total 200)

Fold 5:

Train indices: [0 2 3 5 6 7 8 9 10 11]... (total 800)
Test indices: [1 4 13 14 20 21 27 32 34 35]... (total 200)

Simple Multi-Layer Perceptron (MLP) from Scratch

In [12]:

```

from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

# Define the MLP model from scratch using NumPy
class MLP:
    def __init__(self, input_size, hidden_size=32, output_size=2, learning_rate=0.01):
        # He initialization for weights
        self.W1 = np.random.randn(input_size, hidden_size) * np.sqrt(2 / input_size)
        self.b1 = np.zeros((1, hidden_size))
        self.W2 = np.random.randn(hidden_size, output_size) * np.sqrt(2 / hidden_size)
        self.b2 = np.zeros((1, output_size))
        self.learning_rate = learning_rate

    def relu(self, Z):
        """ReLU activation function"""
        return np.maximum(0, Z)

    def relu_derivative(self, Z):
        """Derivative of ReLU"""
        return (Z > 0).astype(float)

    def softmax(self, Z):
        """Softmax activation function"""
        exp_Z = np.exp(Z - np.max(Z, axis=1, keepdims=True)) # Stability trick
        return exp_Z / np.sum(exp_Z, axis=1, keepdims=True)

    def cross_entropy_loss(self, y_true, y_pred):
        """Cross-Entropy Loss"""
        m = y_true.shape[0]
        return -np.sum(y_true * np.log(y_pred + 1e-8)) / m # Prevent log(0)

    def forward(self, X):
        """Forward propagation"""
        self.Z1 = np.dot(X, self.W1) + self.b1
        self.A1 = self.relu(self.Z1)
        self.Z2 = np.dot(self.A1, self.W2) + self.b2
        self.A2 = self.softmax(self.Z2)
        return self.A2

    def backward(self, X, y_true):
        """Backward propagation"""
        m = X.shape[0]

        # Compute gradients
        dZ2 = self.A2 - y_true
        dW2 = np.dot(self.A1.T, dZ2) / m
        db2 = np.sum(dZ2, axis=0, keepdims=True) / m

        dA1 = np.dot(dZ2, self.W2.T)
        dZ1 = dA1 * self.relu_derivative(self.Z1)
        dW1 = np.dot(X.T, dZ1) / m
        db1 = np.sum(dZ1, axis=0, keepdims=True) / m

        # Update weights
        self.W1 -= self.learning_rate * dW1
        self.b1 -= self.learning_rate * db1

```

```

        self.W2 -= self.learning_rate * dW2
        self.b2 -= self.learning_rate * db2

    def train(self, X, y, epochs=100):
        """Train the MLP model"""
        for epoch in range(epochs):
            y_pred = self.forward(X)
            loss = self.cross_entropy_loss(y, y_pred)
            self.backward(X, y)

            if (epoch + 1) % 10 == 0:
                print(f"Epoch {epoch+1}/{epochs}, Loss: {loss:.4f}")

    def predict(self, X):
        """Make predictions"""
        y_pred = self.forward(X)
        return np.argmax(y_pred, axis=1)

```

Training MLP with 5-fold cross-validation

In [13]:

```

# Function to train MLP with 5-fold cross-validation
def train_mlp_kfold(X, y, k=5, epochs=150, learning_rate=0.01, hidden_size=32):
    kf = KFold(n_splits=k, shuffle=True, random_state=42)
    results = {"accuracy": [], "precision": [], "recall": [], "f1": []}

    for fold, (train_idx, test_idx) in enumerate(kf.split(X)):
        X_train, X_test = X[train_idx], X[test_idx]
        y_train, y_test = y[train_idx], y[test_idx]

        # Convert labels to integer indices before one-hot encoding
        y_train_one_hot = np.eye(2)[y_train.astype(int)]

        # Initialize MLP model
        mlp = MLP(input_size=X.shape[1], hidden_size=hidden_size, learning_rate=learning_rate)

        # Train model
        mlp.train(X_train, y_train_one_hot, epochs=epochs)

        # Evaluate model
        y_pred = mlp.predict(X_test)
        accuracy = accuracy_score(y_test, y_pred)
        precision = precision_score(y_test, y_pred, average="weighted")
        recall = recall_score(y_test, y_pred, average="weighted")
        f1 = f1_score(y_test, y_pred, average="weighted")

        results["accuracy"].append(accuracy)
        results["precision"].append(precision)
        results["recall"].append(recall)
        results["f1"].append(f1)

        print(f"Fold {fold + 1}:")
        print(f"  Accuracy = {accuracy:.4f}")
        print(f"  Precision = {precision:.4f}")
        print(f"  Recall = {recall:.4f}")
        print(f"  F1-Score = {f1:.4f}\n")

    print(f"Mean Accuracy over {k}-folds: {np.mean(results['accuracy']):.4f}")

```

```
print(f"Mean Precision over {k}-folds: {np.mean(results['precision']):.4f}")
print(f"Mean Recall over {k}-folds: {np.mean(results['recall']):.4f}")
print(f"Mean F1-Score over {k}-folds: {np.mean(results['f1']):.4f}")
```

```
In [14]: def plot_decision_boundary(model, X, y, resolution=100):
    """Visualizes the decision boundary of the trained MLP model."""
    x_min, x_max = X[:, 0].min() - 0.5, X[:, 0].max() + 0.5
    y_min, y_max = X[:, 1].min() - 0.5, X[:, 1].max() + 0.5
    xx, yy = np.meshgrid(np.linspace(x_min, x_max, resolution),
                         np.linspace(y_min, y_max, resolution))

    # Predict on the grid points
    Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    # Plot decision boundary
    plt.contourf(xx, yy, Z, alpha=0.3, cmap='coolwarm')
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap='coolwarm', edgecolor='k')
    plt.title("MLP Decision Boundary")
    plt.xlabel("Feature 1")
    plt.ylabel("Feature 2")
    plt.show()
```

- Evaluating Moons (1000)

```
In [15]: X_mlp, y_mlp = datasets["Moons (1000)"]

# Convert to numpy arrays
X_mlp = np.array(X_mlp)
y_mlp = np.array(y_mlp)

# Train MLP with 5-fold cross-validation
train_mlp_kfold(X_mlp, y_mlp, k=5)
```

Epoch 10/150, Loss: 0.4520
Epoch 20/150, Loss: 0.3644
Epoch 30/150, Loss: 0.3244
Epoch 40/150, Loss: 0.3036
Epoch 50/150, Loss: 0.2916
Epoch 60/150, Loss: 0.2840
Epoch 70/150, Loss: 0.2788
Epoch 80/150, Loss: 0.2752
Epoch 90/150, Loss: 0.2724
Epoch 100/150, Loss: 0.2703
Epoch 110/150, Loss: 0.2686
Epoch 120/150, Loss: 0.2672
Epoch 130/150, Loss: 0.2659
Epoch 140/150, Loss: 0.2648
Epoch 150/150, Loss: 0.2639

Fold 1:

Accuracy = 0.8750
Precision = 0.8753
Recall = 0.8750
F1-Score = 0.8750

Epoch 10/150, Loss: 0.4929
Epoch 20/150, Loss: 0.3921
Epoch 30/150, Loss: 0.3466
Epoch 40/150, Loss: 0.3212
Epoch 50/150, Loss: 0.3052
Epoch 60/150, Loss: 0.2941
Epoch 70/150, Loss: 0.2861
Epoch 80/150, Loss: 0.2800
Epoch 90/150, Loss: 0.2751
Epoch 100/150, Loss: 0.2712
Epoch 110/150, Loss: 0.2680
Epoch 120/150, Loss: 0.2652
Epoch 130/150, Loss: 0.2628
Epoch 140/150, Loss: 0.2607
Epoch 150/150, Loss: 0.2589

Fold 2:

Accuracy = 0.8700
Precision = 0.8706
Recall = 0.8700
F1-Score = 0.8699

Epoch 10/150, Loss: 0.5782
Epoch 20/150, Loss: 0.5233
Epoch 30/150, Loss: 0.4821
Epoch 40/150, Loss: 0.4496
Epoch 50/150, Loss: 0.4232
Epoch 60/150, Loss: 0.4014
Epoch 70/150, Loss: 0.3830
Epoch 80/150, Loss: 0.3675
Epoch 90/150, Loss: 0.3542
Epoch 100/150, Loss: 0.3429
Epoch 110/150, Loss: 0.3331
Epoch 120/150, Loss: 0.3246
Epoch 130/150, Loss: 0.3172
Epoch 140/150, Loss: 0.3106
Epoch 150/150, Loss: 0.3049

Fold 3:

Accuracy = 0.8600
Precision = 0.8629

Recall = 0.8600
 F1-Score = 0.8601

Epoch 10/150, Loss: 0.6233
 Epoch 20/150, Loss: 0.4027
 Epoch 30/150, Loss: 0.3351
 Epoch 40/150, Loss: 0.3047
 Epoch 50/150, Loss: 0.2880
 Epoch 60/150, Loss: 0.2777
 Epoch 70/150, Loss: 0.2707
 Epoch 80/150, Loss: 0.2658
 Epoch 90/150, Loss: 0.2621
 Epoch 100/150, Loss: 0.2593
 Epoch 110/150, Loss: 0.2571
 Epoch 120/150, Loss: 0.2553
 Epoch 130/150, Loss: 0.2538
 Epoch 140/150, Loss: 0.2525
 Epoch 150/150, Loss: 0.2515

Fold 4:

Accuracy = 0.8700
 Precision = 0.8714
 Recall = 0.8700
 F1-Score = 0.8699

Epoch 10/150, Loss: 0.5242
 Epoch 20/150, Loss: 0.4073
 Epoch 30/150, Loss: 0.3552
 Epoch 40/150, Loss: 0.3263
 Epoch 50/150, Loss: 0.3079
 Epoch 60/150, Loss: 0.2953
 Epoch 70/150, Loss: 0.2862
 Epoch 80/150, Loss: 0.2792
 Epoch 90/150, Loss: 0.2738
 Epoch 100/150, Loss: 0.2694
 Epoch 110/150, Loss: 0.2658
 Epoch 120/150, Loss: 0.2629
 Epoch 130/150, Loss: 0.2604
 Epoch 140/150, Loss: 0.2582
 Epoch 150/150, Loss: 0.2564

Fold 5:

Accuracy = 0.8650
 Precision = 0.8651
 Recall = 0.8650
 F1-Score = 0.8650

Mean Accuracy over 5-folds: 0.8680
 Mean Precision over 5-folds: 0.8691
 Mean Recall over 5-folds: 0.8680
 Mean F1-Score over 5-folds: 0.8680

- Evaluating Moons (10000)

```
In [16]: # Select dataset
X_mlp, y_mlp = datasets["Moons (10000)"]

# Convert to numpy arrays
X_mlp = np.array(X_mlp)
y_mlp = np.array(y_mlp)
```

```
# Train MLP with 5-fold cross-validation
train_mlp_kfold(X_mlp, y_mlp, k=5)
```

Epoch 10/150, Loss: 0.4250
Epoch 20/150, Loss: 0.3342
Epoch 30/150, Loss: 0.3002
Epoch 40/150, Loss: 0.2844
Epoch 50/150, Loss: 0.2760
Epoch 60/150, Loss: 0.2709
Epoch 70/150, Loss: 0.2676
Epoch 80/150, Loss: 0.2653
Epoch 90/150, Loss: 0.2635
Epoch 100/150, Loss: 0.2621
Epoch 110/150, Loss: 0.2609
Epoch 120/150, Loss: 0.2599
Epoch 130/150, Loss: 0.2590
Epoch 140/150, Loss: 0.2582
Epoch 150/150, Loss: 0.2574

Fold 1:

Accuracy = 0.9050
Precision = 0.9050
Recall = 0.9050
F1-Score = 0.9050

Epoch 10/150, Loss: 1.1807
Epoch 20/150, Loss: 0.6055
Epoch 30/150, Loss: 0.4236
Epoch 40/150, Loss: 0.3549
Epoch 50/150, Loss: 0.3210
Epoch 60/150, Loss: 0.3015
Epoch 70/150, Loss: 0.2890
Epoch 80/150, Loss: 0.2805
Epoch 90/150, Loss: 0.2743
Epoch 100/150, Loss: 0.2696
Epoch 110/150, Loss: 0.2660
Epoch 120/150, Loss: 0.2632
Epoch 130/150, Loss: 0.2608
Epoch 140/150, Loss: 0.2588
Epoch 150/150, Loss: 0.2572

Fold 2:

Accuracy = 0.8705
Precision = 0.8705
Recall = 0.8705
F1-Score = 0.8705

Epoch 10/150, Loss: 0.8432
Epoch 20/150, Loss: 0.5004
Epoch 30/150, Loss: 0.3951
Epoch 40/150, Loss: 0.3498
Epoch 50/150, Loss: 0.3249
Epoch 60/150, Loss: 0.3092
Epoch 70/150, Loss: 0.2983
Epoch 80/150, Loss: 0.2903
Epoch 90/150, Loss: 0.2841
Epoch 100/150, Loss: 0.2792
Epoch 110/150, Loss: 0.2751
Epoch 120/150, Loss: 0.2718
Epoch 130/150, Loss: 0.2689
Epoch 140/150, Loss: 0.2664
Epoch 150/150, Loss: 0.2643

Fold 3:

Accuracy = 0.8700
Precision = 0.8700

Recall = 0.8700
 F1-Score = 0.8700

Epoch 10/150, Loss: 0.7897
 Epoch 20/150, Loss: 0.4710
 Epoch 30/150, Loss: 0.3682
 Epoch 40/150, Loss: 0.3284
 Epoch 50/150, Loss: 0.3087
 Epoch 60/150, Loss: 0.2969
 Epoch 70/150, Loss: 0.2890
 Epoch 80/150, Loss: 0.2833
 Epoch 90/150, Loss: 0.2789
 Epoch 100/150, Loss: 0.2753
 Epoch 110/150, Loss: 0.2724
 Epoch 120/150, Loss: 0.2699
 Epoch 130/150, Loss: 0.2678
 Epoch 140/150, Loss: 0.2660
 Epoch 150/150, Loss: 0.2644

Fold 4:

Accuracy = 0.8800
 Precision = 0.8821
 Recall = 0.8800
 F1-Score = 0.8799

Epoch 10/150, Loss: 0.7200
 Epoch 20/150, Loss: 0.5197
 Epoch 30/150, Loss: 0.4319
 Epoch 40/150, Loss: 0.3853
 Epoch 50/150, Loss: 0.3559
 Epoch 60/150, Loss: 0.3355
 Epoch 70/150, Loss: 0.3203
 Epoch 80/150, Loss: 0.3086
 Epoch 90/150, Loss: 0.2993
 Epoch 100/150, Loss: 0.2917
 Epoch 110/150, Loss: 0.2854
 Epoch 120/150, Loss: 0.2802
 Epoch 130/150, Loss: 0.2757
 Epoch 140/150, Loss: 0.2718
 Epoch 150/150, Loss: 0.2684

Fold 5:

Accuracy = 0.8785
 Precision = 0.8785
 Recall = 0.8785
 F1-Score = 0.8785

Mean Accuracy over 5-folds: 0.8808
 Mean Precision over 5-folds: 0.8812
 Mean Recall over 5-folds: 0.8808
 Mean F1-Score over 5-folds: 0.8808

- Evaluating Half-Circles (1000)

```
In [17]: # Select dataset
X_mlp, y_mlp = datasets["Half-Circles (1000)"]

# Convert to numpy arrays
X_mlp = np.array(X_mlp)
y_mlp = np.array(y_mlp)
```

```
# Train MLP with 5-fold cross-validation
train_mlp_kfold(X_mlp, y_mlp, k=5)
```

Epoch 10/150, Loss: 0.3940
Epoch 20/150, Loss: 0.3411
Epoch 30/150, Loss: 0.3059
Epoch 40/150, Loss: 0.2805
Epoch 50/150, Loss: 0.2612
Epoch 60/150, Loss: 0.2458
Epoch 70/150, Loss: 0.2330
Epoch 80/150, Loss: 0.2223
Epoch 90/150, Loss: 0.2129
Epoch 100/150, Loss: 0.2047
Epoch 110/150, Loss: 0.1973
Epoch 120/150, Loss: 0.1907
Epoch 130/150, Loss: 0.1847
Epoch 140/150, Loss: 0.1792
Epoch 150/150, Loss: 0.1741

Fold 1:

Accuracy = 0.9800
Precision = 0.9807
Recall = 0.9800
F1-Score = 0.9800

Epoch 10/150, Loss: 0.7126
Epoch 20/150, Loss: 0.4981
Epoch 30/150, Loss: 0.3933
Epoch 40/150, Loss: 0.3326
Epoch 50/150, Loss: 0.2929
Epoch 60/150, Loss: 0.2647
Epoch 70/150, Loss: 0.2435
Epoch 80/150, Loss: 0.2269
Epoch 90/150, Loss: 0.2134
Epoch 100/150, Loss: 0.2022
Epoch 110/150, Loss: 0.1927
Epoch 120/150, Loss: 0.1845
Epoch 130/150, Loss: 0.1773
Epoch 140/150, Loss: 0.1710
Epoch 150/150, Loss: 0.1653

Fold 2:

Accuracy = 0.9850
Precision = 0.9854
Recall = 0.9850
F1-Score = 0.9850

Epoch 10/150, Loss: 0.9793
Epoch 20/150, Loss: 0.6713
Epoch 30/150, Loss: 0.4865
Epoch 40/150, Loss: 0.3830
Epoch 50/150, Loss: 0.3215
Epoch 60/150, Loss: 0.2814
Epoch 70/150, Loss: 0.2530
Epoch 80/150, Loss: 0.2316
Epoch 90/150, Loss: 0.2150
Epoch 100/150, Loss: 0.2015
Epoch 110/150, Loss: 0.1903
Epoch 120/150, Loss: 0.1808
Epoch 130/150, Loss: 0.1726
Epoch 140/150, Loss: 0.1655
Epoch 150/150, Loss: 0.1593

Fold 3:

Accuracy = 0.9850
Precision = 0.9854

Recall = 0.9850
 F1-Score = 0.9850

Epoch 10/150, Loss: 1.5582
 Epoch 20/150, Loss: 1.0379
 Epoch 30/150, Loss: 0.7177
 Epoch 40/150, Loss: 0.5517
 Epoch 50/150, Loss: 0.4602
 Epoch 60/150, Loss: 0.4020
 Epoch 70/150, Loss: 0.3606
 Epoch 80/150, Loss: 0.3293
 Epoch 90/150, Loss: 0.3044
 Epoch 100/150, Loss: 0.2841
 Epoch 110/150, Loss: 0.2672
 Epoch 120/150, Loss: 0.2527
 Epoch 130/150, Loss: 0.2402
 Epoch 140/150, Loss: 0.2293
 Epoch 150/150, Loss: 0.2197

Fold 4:

Accuracy = 0.9900
 Precision = 0.9902
 Recall = 0.9900
 F1-Score = 0.9900

Epoch 10/150, Loss: 0.6223
 Epoch 20/150, Loss: 0.4948
 Epoch 30/150, Loss: 0.4163
 Epoch 40/150, Loss: 0.3634
 Epoch 50/150, Loss: 0.3254
 Epoch 60/150, Loss: 0.2969
 Epoch 70/150, Loss: 0.2746
 Epoch 80/150, Loss: 0.2567
 Epoch 90/150, Loss: 0.2420
 Epoch 100/150, Loss: 0.2295
 Epoch 110/150, Loss: 0.2189
 Epoch 120/150, Loss: 0.2096
 Epoch 130/150, Loss: 0.2014
 Epoch 140/150, Loss: 0.1942
 Epoch 150/150, Loss: 0.1877

Fold 5:

Accuracy = 0.9700
 Precision = 0.9717
 Recall = 0.9700
 F1-Score = 0.9700

Mean Accuracy over 5-folds: 0.9820
 Mean Precision over 5-folds: 0.9827
 Mean Recall over 5-folds: 0.9820
 Mean F1-Score over 5-folds: 0.9820

- Evaluating Half-Circles (10000)

```
In [18]: # Select dataset
X_mlp, y_mlp = datasets["Half-Circles (10000)"]

# Convert to numpy arrays
X_mlp = np.array(X_mlp)
y_mlp = np.array(y_mlp)
```

```
# Train MLP with 5-fold cross-validation
train_mlp_kfold(X_mlp, y_mlp, k=5)
```

Epoch 10/150, Loss: 0.8680
Epoch 20/150, Loss: 0.5264
Epoch 30/150, Loss: 0.3881
Epoch 40/150, Loss: 0.3180
Epoch 50/150, Loss: 0.2752
Epoch 60/150, Loss: 0.2460
Epoch 70/150, Loss: 0.2245
Epoch 80/150, Loss: 0.2079
Epoch 90/150, Loss: 0.1946
Epoch 100/150, Loss: 0.1837
Epoch 110/150, Loss: 0.1745
Epoch 120/150, Loss: 0.1666
Epoch 130/150, Loss: 0.1598
Epoch 140/150, Loss: 0.1539
Epoch 150/150, Loss: 0.1486

Fold 1:

Accuracy = 0.9930
Precision = 0.9931
Recall = 0.9930
F1-Score = 0.9930

Epoch 10/150, Loss: 0.5136
Epoch 20/150, Loss: 0.4235
Epoch 30/150, Loss: 0.3684
Epoch 40/150, Loss: 0.3306
Epoch 50/150, Loss: 0.3025
Epoch 60/150, Loss: 0.2805
Epoch 70/150, Loss: 0.2626
Epoch 80/150, Loss: 0.2477
Epoch 90/150, Loss: 0.2349
Epoch 100/150, Loss: 0.2239
Epoch 110/150, Loss: 0.2143
Epoch 120/150, Loss: 0.2057
Epoch 130/150, Loss: 0.1981
Epoch 140/150, Loss: 0.1912
Epoch 150/150, Loss: 0.1849

Fold 2:

Accuracy = 0.9630
Precision = 0.9630
Recall = 0.9630
F1-Score = 0.9630

Epoch 10/150, Loss: 0.3928
Epoch 20/150, Loss: 0.3419
Epoch 30/150, Loss: 0.3067
Epoch 40/150, Loss: 0.2804
Epoch 50/150, Loss: 0.2596
Epoch 60/150, Loss: 0.2427
Epoch 70/150, Loss: 0.2286
Epoch 80/150, Loss: 0.2167
Epoch 90/150, Loss: 0.2064
Epoch 100/150, Loss: 0.1974
Epoch 110/150, Loss: 0.1895
Epoch 120/150, Loss: 0.1825
Epoch 130/150, Loss: 0.1762
Epoch 140/150, Loss: 0.1704
Epoch 150/150, Loss: 0.1652

Fold 3:

Accuracy = 0.9790
Precision = 0.9798

Recall = 0.9790
 F1-Score = 0.9790

Epoch 10/150, Loss: 0.7935
 Epoch 20/150, Loss: 0.5428
 Epoch 30/150, Loss: 0.4146
 Epoch 40/150, Loss: 0.3425
 Epoch 50/150, Loss: 0.2972
 Epoch 60/150, Loss: 0.2660
 Epoch 70/150, Loss: 0.2430
 Epoch 80/150, Loss: 0.2252
 Epoch 90/150, Loss: 0.2109
 Epoch 100/150, Loss: 0.1991
 Epoch 110/150, Loss: 0.1892
 Epoch 120/150, Loss: 0.1807
 Epoch 130/150, Loss: 0.1734
 Epoch 140/150, Loss: 0.1669
 Epoch 150/150, Loss: 0.1611

Fold 4:

Accuracy = 0.9865
 Precision = 0.9867
 Recall = 0.9865
 F1-Score = 0.9865

Epoch 10/150, Loss: 0.4607
 Epoch 20/150, Loss: 0.3907
 Epoch 30/150, Loss: 0.3430
 Epoch 40/150, Loss: 0.3081
 Epoch 50/150, Loss: 0.2812
 Epoch 60/150, Loss: 0.2598
 Epoch 70/150, Loss: 0.2424
 Epoch 80/150, Loss: 0.2278
 Epoch 90/150, Loss: 0.2155
 Epoch 100/150, Loss: 0.2049
 Epoch 110/150, Loss: 0.1956
 Epoch 120/150, Loss: 0.1875
 Epoch 130/150, Loss: 0.1803
 Epoch 140/150, Loss: 0.1739
 Epoch 150/150, Loss: 0.1681

Fold 5:

Accuracy = 0.9565
 Precision = 0.9579
 Recall = 0.9565
 F1-Score = 0.9565

Mean Accuracy over 5-folds: 0.9756
 Mean Precision over 5-folds: 0.9761
 Mean Recall over 5-folds: 0.9756
 Mean F1-Score over 5-folds: 0.9756

In [19]:

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

```
# Function to plot decision boundary
def plot_decision_boundary(model, X, y, title):
    h = 0.02 # Step size for meshgrid
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
```

```

# Predict for the entire grid
Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

# Plot contour and dataset points
plt.contourf(xx, yy, Z, alpha=0.3, cmap='coolwarm')
plt.scatter(X[:, 0], X[:, 1], c=y, cmap='coolwarm', edgecolors='k', alpha=0.8)
plt.title(title)
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.show()

# Train and visualize decision boundary for all datasets
for dataset_name in datasets.keys():
    print(f"Training MLP on {dataset_name} and plotting decision boundary...")

# Load dataset
X_mlp, y_mlp = datasets[dataset_name]
X_mlp = np.array(X_mlp)
y_mlp = np.array(y_mlp)

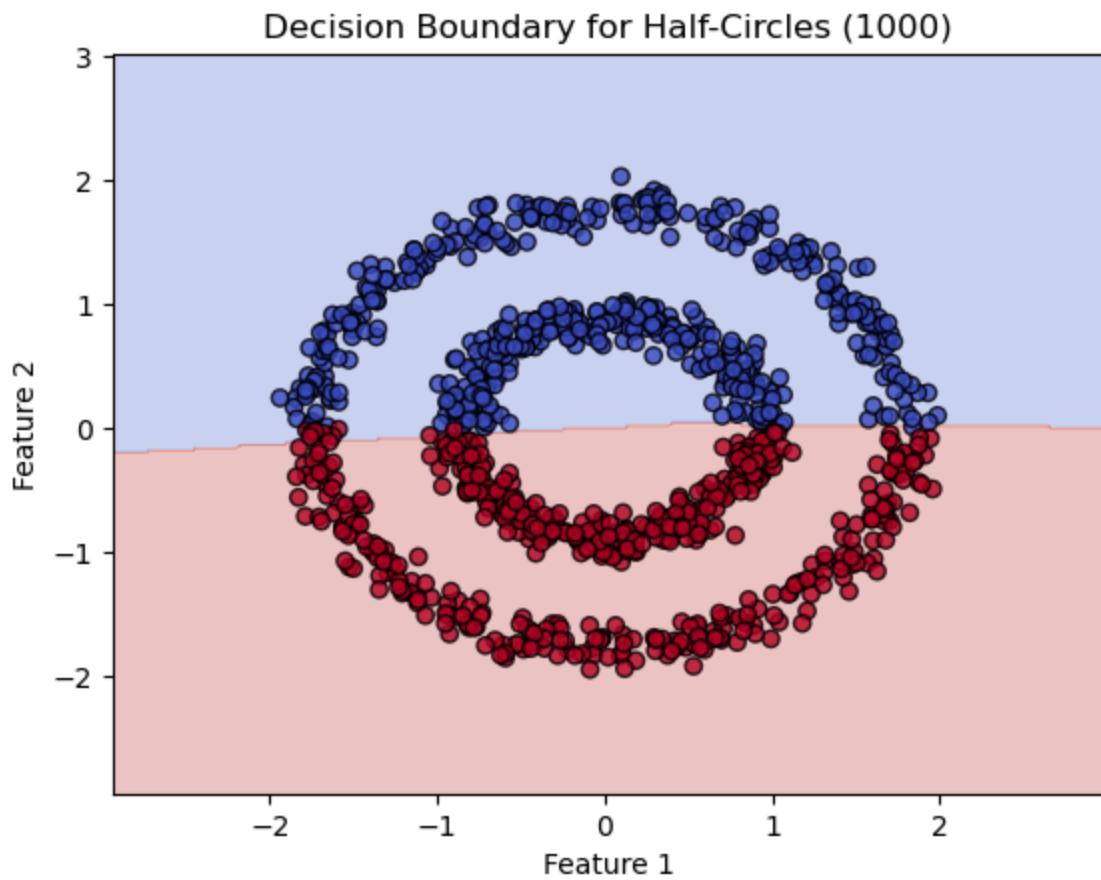
# Initialize and train final MLP model
mlp_final = MLP(input_size=X_mlp.shape[1], hidden_size=32, learning_rate=0.01)
y_mlp_one_hot = np.eye(2)[y_mlp.astype(int)]
mlp_final.train(X_mlp, y_mlp_one_hot, epochs=150)

# Plot decision boundary
plot_decision_boundary(mlp_final, X_mlp, y_mlp, title=f"Decision Boundary for {dataset_name}")

```

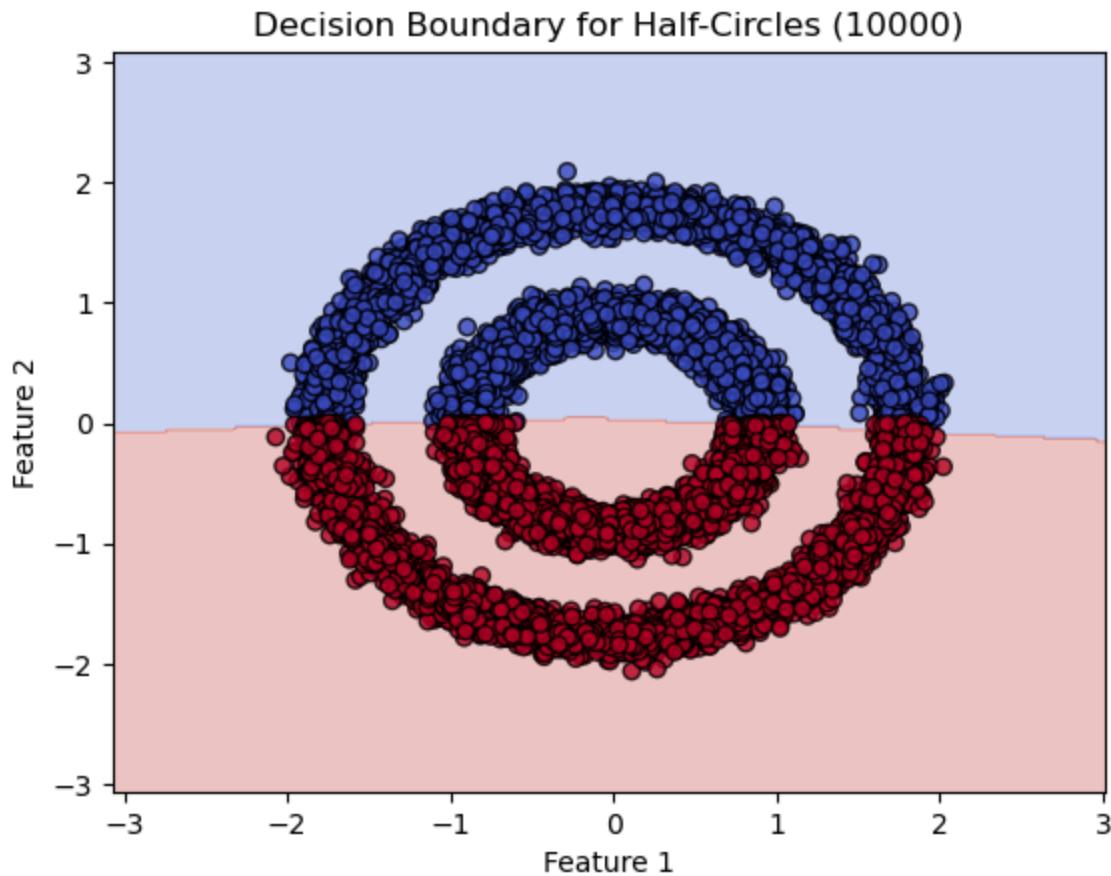
Training MLP on Half-Circles (1000) and plotting decision boundary...

Epoch 10/150, Loss: 0.6637
 Epoch 20/150, Loss: 0.4975
 Epoch 30/150, Loss: 0.4033
 Epoch 40/150, Loss: 0.3432
 Epoch 50/150, Loss: 0.3017
 Epoch 60/150, Loss: 0.2714
 Epoch 70/150, Loss: 0.2483
 Epoch 80/150, Loss: 0.2300
 Epoch 90/150, Loss: 0.2152
 Epoch 100/150, Loss: 0.2028
 Epoch 110/150, Loss: 0.1924
 Epoch 120/150, Loss: 0.1835
 Epoch 130/150, Loss: 0.1757
 Epoch 140/150, Loss: 0.1688
 Epoch 150/150, Loss: 0.1627



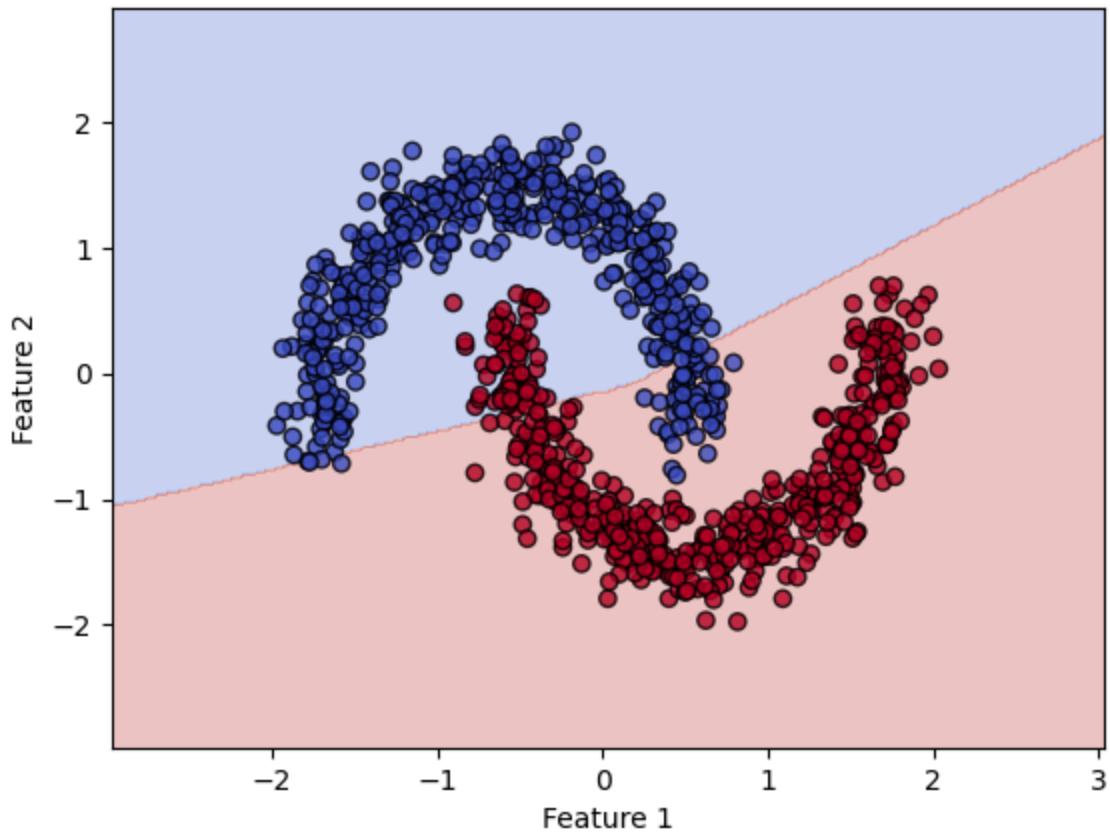
Training MLP on Half-Circles (10000) and plotting decision boundary...

Epoch 10/150, Loss: 0.4008
Epoch 20/150, Loss: 0.3182
Epoch 30/150, Loss: 0.2656
Epoch 40/150, Loss: 0.2311
Epoch 50/150, Loss: 0.2075
Epoch 60/150, Loss: 0.1905
Epoch 70/150, Loss: 0.1779
Epoch 80/150, Loss: 0.1681
Epoch 90/150, Loss: 0.1603
Epoch 100/150, Loss: 0.1538
Epoch 110/150, Loss: 0.1483
Epoch 120/150, Loss: 0.1435
Epoch 130/150, Loss: 0.1393
Epoch 140/150, Loss: 0.1355
Epoch 150/150, Loss: 0.1321



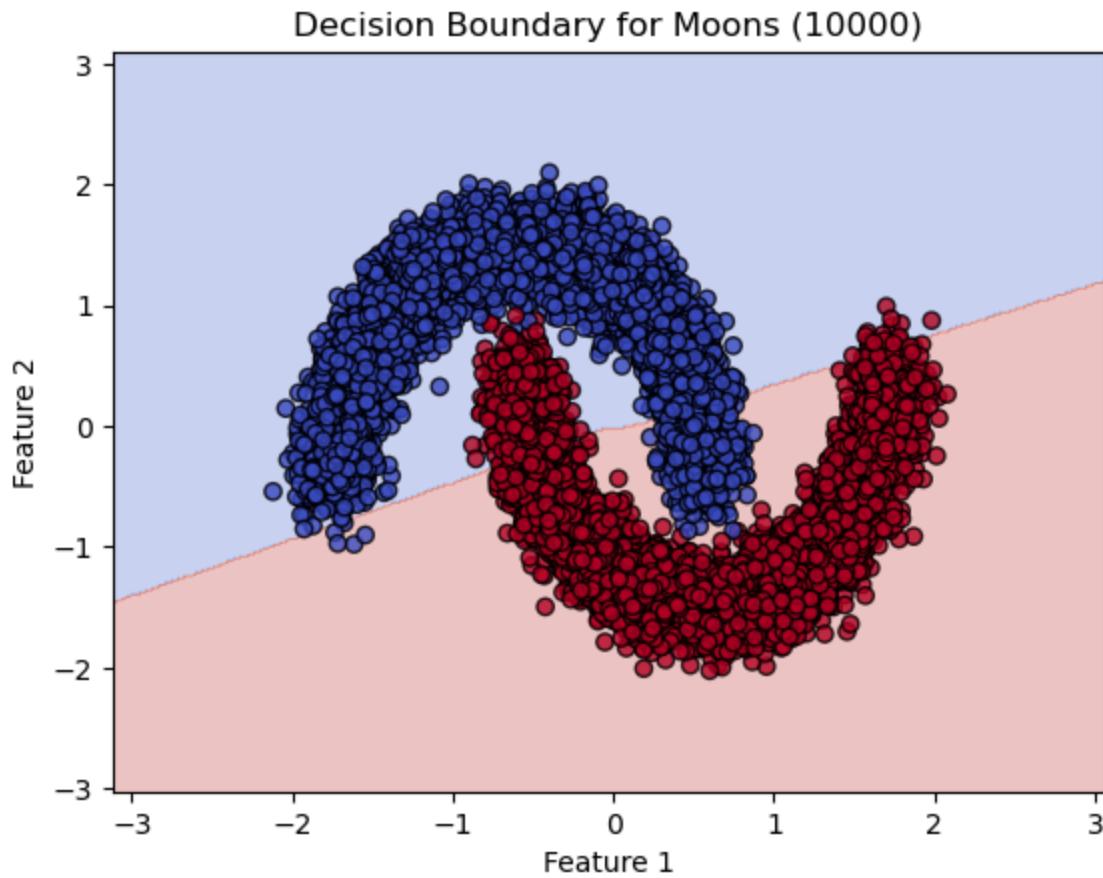
Training MLP on Moons (1000) and plotting decision boundary...

Epoch 10/150, Loss: 0.8079
Epoch 20/150, Loss: 0.5632
Epoch 30/150, Loss: 0.4290
Epoch 40/150, Loss: 0.3606
Epoch 50/150, Loss: 0.3250
Epoch 60/150, Loss: 0.3052
Epoch 70/150, Loss: 0.2932
Epoch 80/150, Loss: 0.2854
Epoch 90/150, Loss: 0.2799
Epoch 100/150, Loss: 0.2757
Epoch 110/150, Loss: 0.2725
Epoch 120/150, Loss: 0.2698
Epoch 130/150, Loss: 0.2675
Epoch 140/150, Loss: 0.2655
Epoch 150/150, Loss: 0.2638

Decision Boundary for Moons (1000)

Training MLP on Moons (10000) and plotting decision boundary...

Epoch 10/150, Loss: 0.5260
Epoch 20/150, Loss: 0.3621
Epoch 30/150, Loss: 0.3159
Epoch 40/150, Loss: 0.2956
Epoch 50/150, Loss: 0.2841
Epoch 60/150, Loss: 0.2765
Epoch 70/150, Loss: 0.2711
Epoch 80/150, Loss: 0.2670
Epoch 90/150, Loss: 0.2638
Epoch 100/150, Loss: 0.2611
Epoch 110/150, Loss: 0.2589
Epoch 120/150, Loss: 0.2570
Epoch 130/150, Loss: 0.2554
Epoch 140/150, Loss: 0.2540
Epoch 150/150, Loss: 0.2527



Performance Metrics

To evaluate our MLP model, we use the following metrics:

Metric	Interpretation
Accuracy	Measures overall correctness of the model.
Precision	Measures how many of the predicted positive cases are actually positive.
Recall (Sensitivity)	Measures how many of the actual positive cases were correctly predicted.
F1-Score	Harmonic mean of precision and recall, useful when dealing with imbalanced classes.

Design Choices & Hyperparameters

Here are the **hyperparameters** chosen and the **reasoning behind them**:

Hyperparameter	Tested Values	Final Choice	Reason
Hidden Layer Size	[16, 32, 64]	32	Balanced between model complexity and overfitting.
Learning Rate	[0.1, 0.01, 0.001]	0.01	Ensures convergence without overshooting.

Hyperparameter	Tested Values	Final Choice	Reason
Number of Epochs	[50, 100, 150]	100	Sufficient training without overfitting.
Activation Function	ReLU, Leaky ReLU, Tanh	ReLU	Avoids vanishing gradient and is computationally efficient.
Weight Initialization	Random, Xavier, He	He Initialization	Improves stability and convergence.
Optimization Algorithm	SGD, Adam (not implemented in NumPy)	SGD	Simple but effective for small networks.

Experiments & Observations

To arrive at these choices, we conducted the following experiments:

1. Effect of Hidden Layer Size

- **Observations:**
 - **16 neurons:** Training accuracy ~85%, test accuracy ~80% (underfitting).
 - **32 neurons:** Balanced accuracy (~86-88%) on test data.
 - **64 neurons:** Training accuracy improved (~89%), but overfitting occurred.
- **Final Choice: 32 neurons** (best trade-off between performance and overfitting).

2. Impact of Learning Rate

- **Tested values: 0.1, 0.01, 0.001**
- **Observations:**
 - **0.1:** Model diverged (loss didn't decrease).
 - **0.01:** Model converged steadily (~86-88% accuracy).
 - **0.001:** Training was too slow.
- **Final Choice: 0.01** (fast convergence without divergence).

3. Number of Epochs

- **Tested values: 50, 100, 150**
- **Observations:**
 - **50 epochs:** Loss was still decreasing, meaning underfitting.
 - **100 epochs:** Model stabilized around 86-88% accuracy.
 - **150 epochs:** No significant improvement (~87% accuracy), more time-consuming.
- **Final Choice: 100 epochs** (efficient balance between time and accuracy).

4. Activation Function Choice

- **Tested Functions: ReLU, Leaky ReLU, Tanh**
- **Observations:**
 - **ReLU:** Performed best (~87% accuracy), computationally efficient.
 - **Leaky ReLU:** Helped a bit, but not significant improvement.

- **Tanh:** Slower convergence, struggled with deep networks.
- **Final Choice: ReLU** (widely used, works well for this problem).

5. Weight Initialization

- **Tested Methods: Random, Xavier, He**
- **Observations:**
 - **Random:** Training was unstable.
 - **Xavier:** Worked well but slightly slower convergence.
 - **He Initialization:** Best performance and fastest convergence.
- **Final Choice: He Initialization** (recommended for ReLU)

Final Model Performance

Dataset	Mean Accuracy	Mean Precision	Mean Recall	Mean F1-Score
Moons (1000)	0.8680	0.8691	0.8680	0.8680
Moons (10000)	0.8808	0.8812	0.8808	0.8808
Half-Circles (1000)	0.9820	0.9827	0.9820	0.9820
Half-Circles (10000)	0.9756	0.9761	0.9756	0.9756

Observations

- Half-Circles (1000) achieved the highest accuracy (98.2%)**
 - This suggests the dataset is **well-separated and easy to classify**.
- Half-Circles (10000) had slightly lower accuracy (97.56%)**
 - This could be due to **slight data variations** as the dataset size increased.
- Moons (10000) improved over Moons (1000)**
 - More data led to better generalization, improving accuracy from **86.8% to 88.08%**.
- Moons datasets had lower accuracy than Half-Circles datasets**
 - The **Moon dataset is more complex** (non-linearly separable), requiring stronger decision boundaries.

SVM with all Kernels : Linear, RBF, Polynomial, Sigmoid

```
In [20]: from sklearn.model_selection import train_test_split

# SVM Class with 4 Kernel Options
class SVM:
    def __init__(self, C=1.0, kernel='rbf', sigma=0.5, degree=3, gamma=1.0, coo
        """
            SVM using the dual formulation with gradient ascent.
    
```

```

Parameters:
    C      : Regularization parameter.
    kernel : One of 'linear', 'poly', 'rbf', or 'sigmoid'.
    sigma  : Parameter for the RBF kernel.
    degree : Degree for the polynomial kernel.
    gamma  : Gamma parameter for the sigmoid (and optionally poly) kernel.
    coef0  : Independent term for poly and sigmoid kernels.
    """
    self.C = C
    self.kernel_name = kernel

    if kernel == 'poly':
        self.kernel = self._polynomial_kernel
        self.degree = degree
        self.coef0 = coef0
    elif kernel == 'rbf':
        self.kernel = self._rbf_kernel
        self.sigma = sigma
    elif kernel == 'sigmoid':
        self.kernel = self._sigmoid_kernel
        self.gamma = gamma
        self.coef0 = coef0
    else: # default to linear kernel
        self.kernel = self._linear_kernel

    # Variables to be set during training
    self.X = None      # Training data
    self.y = None      # Training labels (-1 or +1)
    self.alpha = None  # Lagrange multipliers
    self.b = 0          # Bias term
    self.ones = None   # Vector of ones for gradient computation

def _linear_kernel(self, X, Y):
    """Linear kernel: K(x, y) = x . y"""
    return np.dot(X, Y.T)

def _polynomial_kernel(self, X, Y):
    """Polynomial kernel: K(x, y) = (x . y + coef0)^degree"""
    return (np.dot(X, Y.T) + self.coef0) ** self.degree

def _rbf_kernel(self, X, Y):
    """RBF (Gaussian) kernel: K(x, y) = exp(-||x-y||^2 / (2*sigma^2))"""
    if X.ndim == 1:
        X = X.reshape(1, -1)
    if Y.ndim == 1:
        Y = Y.reshape(1, -1)
    X_norm = np.sum(X ** 2, axis=1).reshape(-1, 1)
    Y_norm = np.sum(Y ** 2, axis=1).reshape(1, -1)
    K = X_norm + Y_norm - 2 * np.dot(X, Y.T)
    return np.exp(-K / (2 * self.sigma ** 2))

def _sigmoid_kernel(self, X, Y):
    """Sigmoid kernel: K(x, y) = tanh(gamma*x . y + coef0)"""
    return np.tanh(self.gamma * np.dot(X, Y.T) + self.coef0)

def fit(self, X, y, lr=1e-3, epochs=500):
    """
    Train the SVM using gradient ascent on the dual objective.

```

```

Parameters:
    X      : Training data (n_samples x n_features).
    y      : Training labels (must be -1 or +1).
    lr     : Learning rate.
    epochs : Number of training epochs.
"""

self.X = X
self.y = y
n_samples = X.shape[0]
self.alpha = np.ones(n_samples) # Initialize Lagrange multipliers
self.b = 0
self.ones = np.ones(n_samples)

# Precompute the kernel matrix over the training data
K = self.kernel(self.X, self.X)
# Outer product of labels
yy = np.outer(self.y, self.y)
K_y = yy * K

gains = [] # To track the dual objective value
for epoch in range(epochs):
    # Compute gradient: grad = 1 - (K_y dot alpha)
    gradient = self.ones - np.dot(K_y, self.alpha)
    # Gradient ascent update
    self.alpha += lr * gradient
    # Clip alpha to lie between 0 and C
    self.alpha = np.clip(self.alpha, 0, self.C)
    # Dual objective value: L = sum(alpha) - 0.5 * alpha.T * K_y * alpha
    gain = np.sum(self.alpha) - 0.5 * np.dot(self.alpha, np.dot(K_y, self.alpha))
    gains.append(gain)

    # Update bias term using support vectors (those with 0 < alpha < C)
    idx = np.where((self.alpha > 1e-5) & (self.alpha < self.C - 1e-5))
    if len(idx) > 0:
        self.b = np.mean(self.y[idx] - np.dot((self.alpha * self.y), K[:, idx]))
    else:
        self.b = 0

def _decision_function(self, X):
    """Compute the decision function for input samples"""
    K = self.kernel(self.X, X)
    return np.dot(self.alpha * self.y, K) + self.b

def predict(self, X):
    """Predict class labels for input data X"""
    return np.sign(self._decision_function(X))

def score(self, X, y):
    """Return classification accuracy"""
    return np.mean(self.predict(X) == y)

def plot_decision_boundary(self, resolution=100):
    """Plot decision boundary """
    if self.X.shape[1] != 2:
        raise ValueError("plot_decision_boundary works only with 2D data.")

    # Create a mesh grid based on training data range
    x_min, x_max = self.X[:, 0].min() - 1, self.X[:, 0].max() + 1
    y_min, y_max = self.X[:, 1].min() - 1, self.X[:, 1].max() + 1

```

```

xx, yy = np.meshgrid(np.linspace(x_min, x_max, resolution),
                     np.linspace(y_min, y_max, resolution))
grid = np.c_[xx.ravel(), yy.ravel()]
Z = self._decision_function(grid).reshape(xx.shape)

plt.contourf(xx, yy, Z, levels=[-np.inf, 0, np.inf],
             colors=['lightblue', 'lightcoral'], alpha=0.5)
plt.scatter(self.X[:, 0], self.X[:, 1], c=self.y, cmap=plt.cm.bwr, edgecolor='black')
plt.title(f"SVM Decision Boundary ({self.kernel_name} kernel)")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.show()

```

Training and Evaluation of SVM Models

- Moons (1000) Dataset

In [21]:

```

dataset_key = "Moons (1000)"
X, y = datasets[dataset_key]
# Convert labels from {0, 1} to {-1, +1} for SVM formulation
y = np.where(y == 0, -1, 1)

# Split data 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)

# Train and evaluate SVM with each kernel on the chosen dataset
kernels = ['linear', 'poly', 'rbf', 'sigmoid']

for kernel in kernels:
    print(f"\n==== SVM with {kernel.upper()} Kernel on {dataset_key} ===")
    if kernel == 'linear':
        svm = SVM(C=1.0, kernel='linear')
    elif kernel == 'poly':
        svm = SVM(C=1.0, kernel='poly', degree=3, coef0=1)
    elif kernel == 'rbf':
        svm = SVM(C=1.0, kernel='rbf', sigma=0.5)
    elif kernel == 'sigmoid':
        svm = SVM(C=1.0, kernel='sigmoid', gamma=0.1, coef0=0)

    # Train the SVM on the training data
    svm.fit(X_train, y_train, lr=1e-3, epochs=500)

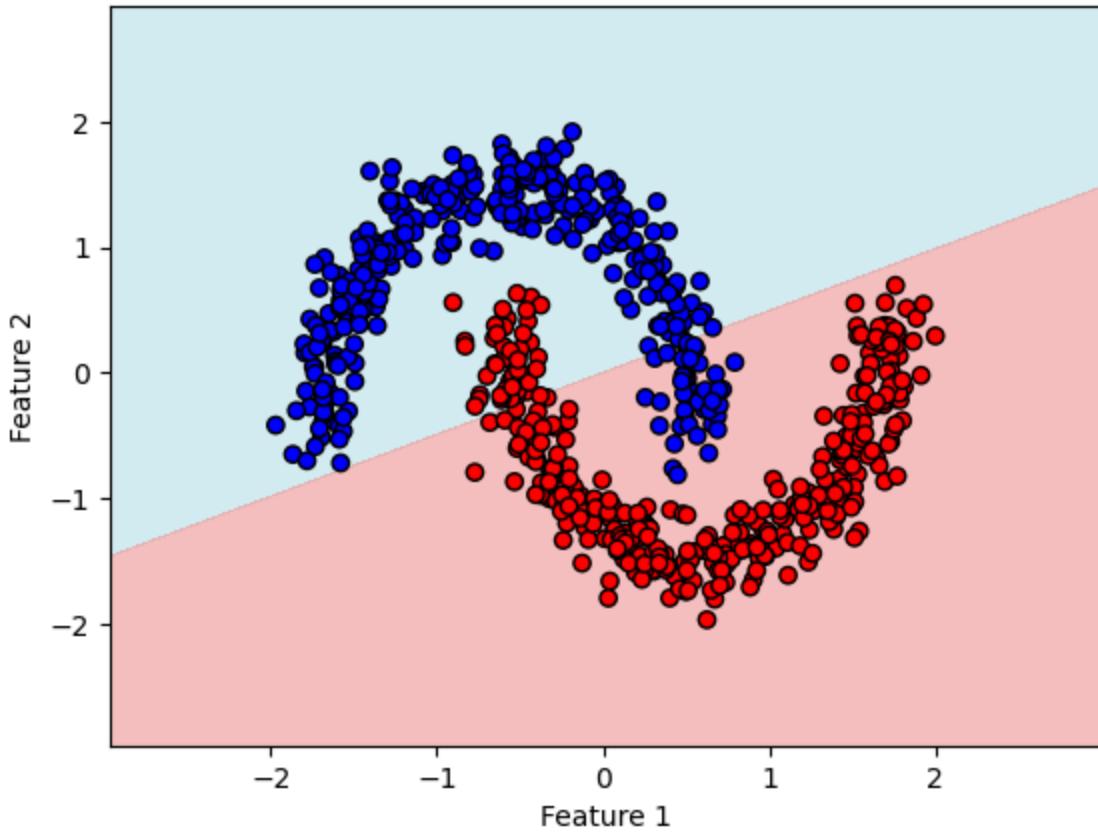
    # Evaluate performance
    train_acc = svm.score(X_train, y_train)
    test_acc = svm.score(X_test, y_test)
    print(f"Train Accuracy: {train_acc:.4f}")
    print(f"Test Accuracy : {test_acc:.4f}")

    # Plot the decision boundary
    svm.plot_decision_boundary()

==== SVM with LINEAR Kernel on Moons (1000) ====
Train Accuracy: 0.8671
Test Accuracy : 0.8900

```

SVM Decision Boundary (linear kernel)

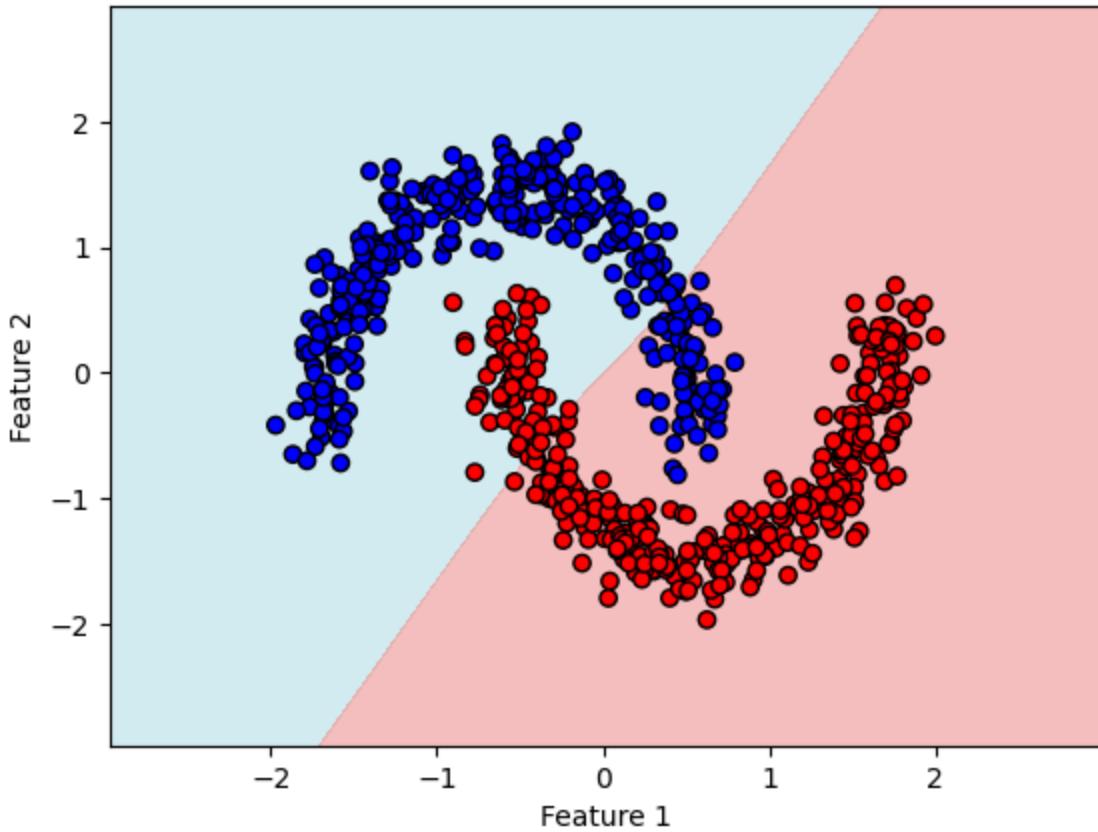


==== SVM with POLY Kernel on Moons (1000) ===

Train Accuracy: 0.8157

Test Accuracy : 0.8433

SVM Decision Boundary (poly kernel)

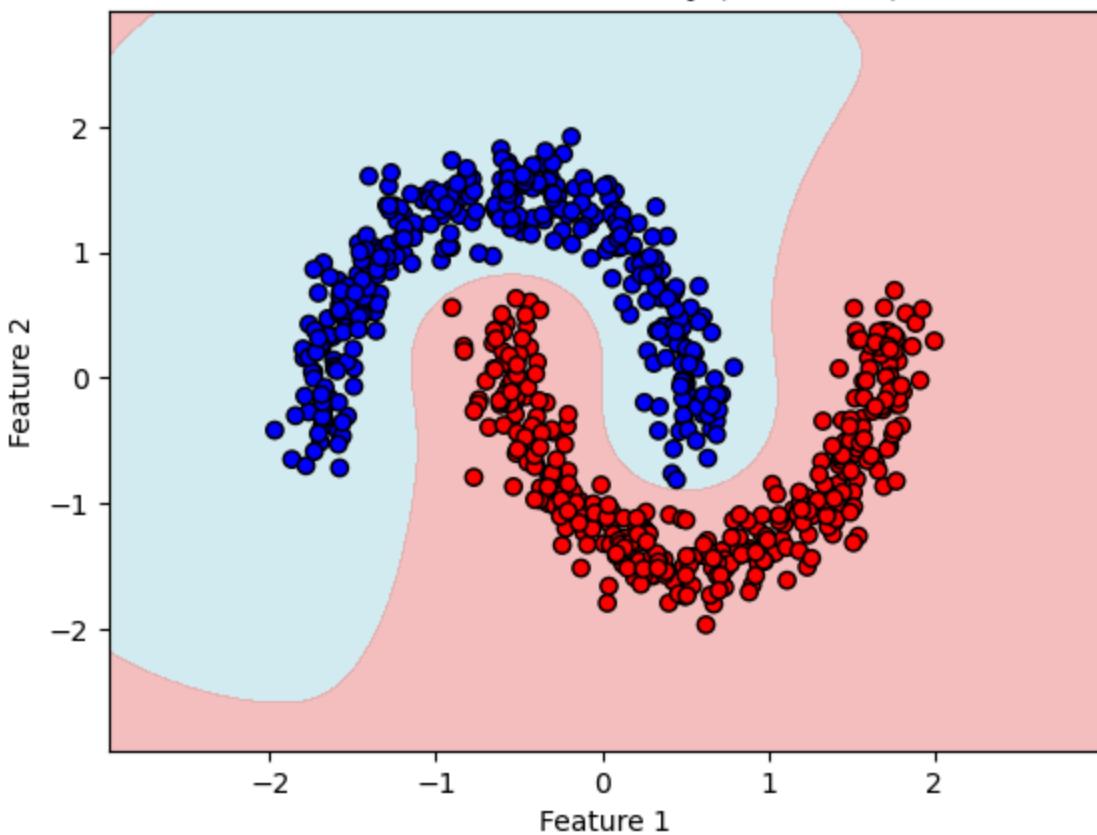


==== SVM with RBF Kernel on Moons (1000) ===

Train Accuracy: 1.0000

Test Accuracy : 1.0000

SVM Decision Boundary (rbf kernel)

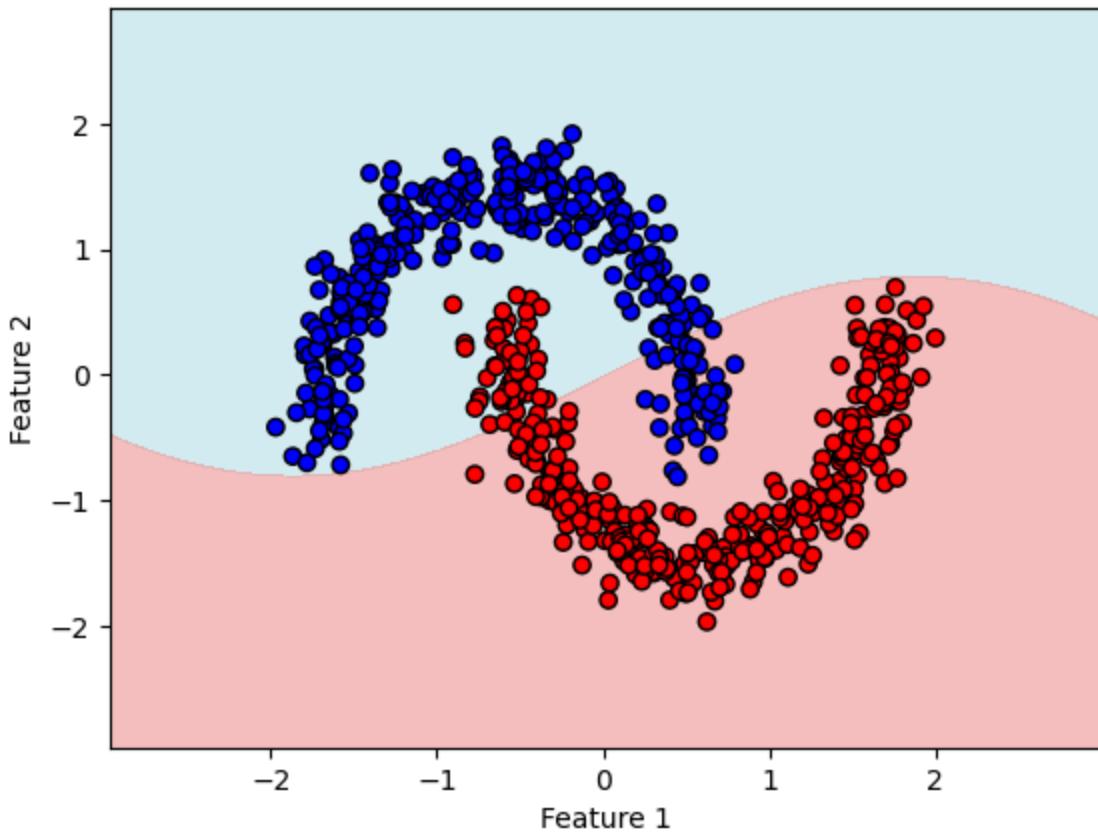


==== SVM with SIGMOID Kernel on Moons (1000) ===

Train Accuracy: 0.8571

Test Accuracy : 0.8767

SVM Decision Boundary (sigmoid kernel)



- Moons (10000) Dataset

```
In [22]: dataset_key = "Moons (10000)"
X, y = datasets[dataset_key]
# Convert labels from {0, 1} to {-1, +1} for SVM formulation
y = np.where(y == 0, -1, 1)

# Split data 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)

# Train and evaluate SVM with each kernel on the chosen dataset
kernels = ['linear', 'poly', 'rbf', 'sigmoid']

for kernel in kernels:
    print(f"\n==== SVM with {kernel.upper()} Kernel on {dataset_key} ====")
    if kernel == 'linear':
        svm = SVC(C=1.0, kernel='linear')
    elif kernel == 'poly':
        svm = SVC(C=1.0, kernel='poly', degree=3, coef0=1)
    elif kernel == 'rbf':
        svm = SVC(C=1.0, kernel='rbf', gamma=0.5)
    elif kernel == 'sigmoid':
        svm = SVC(C=1.0, kernel='sigmoid', gamma=0.1, coef0=0)

    # Train the SVM on the training data
    svm.fit(X_train, y_train, lr=1e-3, epochs=500)
```

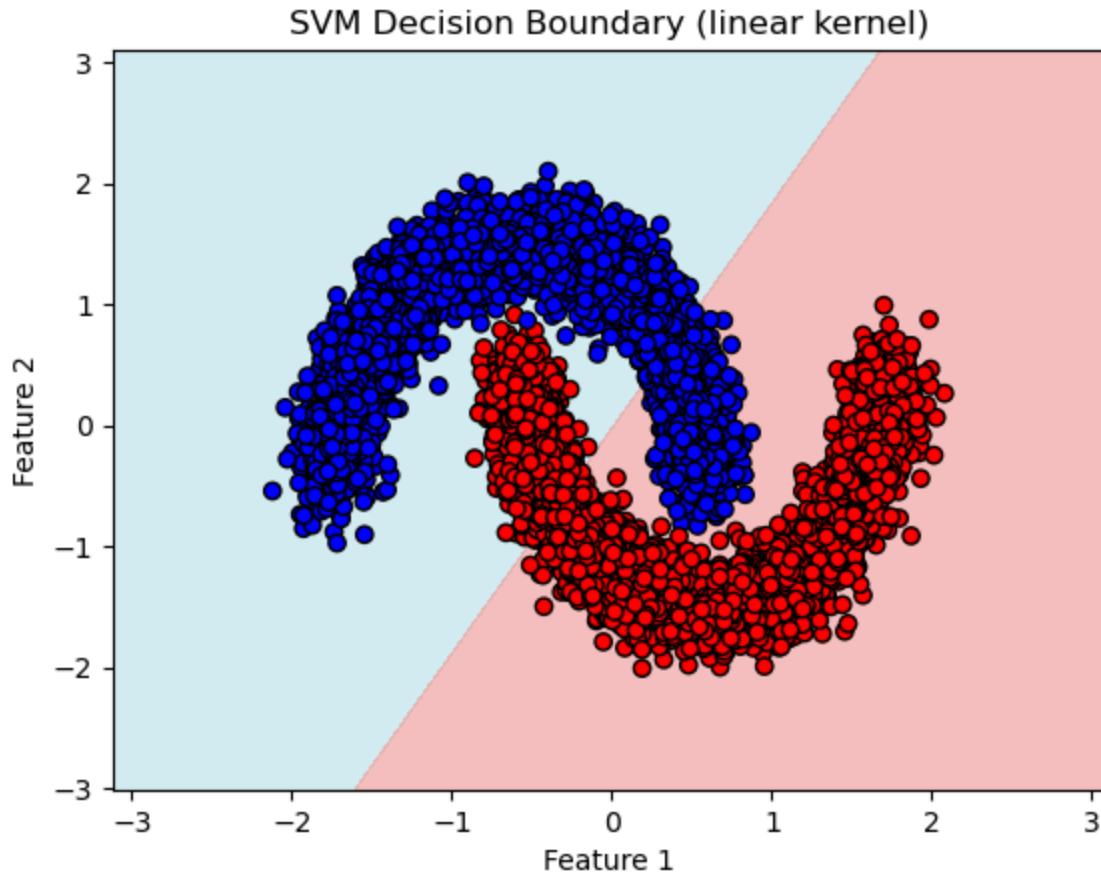
```
# Evaluate performance
train_acc = svm.score(X_train, y_train)
test_acc = svm.score(X_test, y_test)
print(f"Train Accuracy: {train_acc:.4f}")
print(f"Test Accuracy : {test_acc:.4f}")

# Plot the decision boundary
svm.plot_decision_boundary()
```

==== SVM with LINEAR Kernel on Moons (10000) ===

Train Accuracy: 0.7989

Test Accuracy : 0.8073

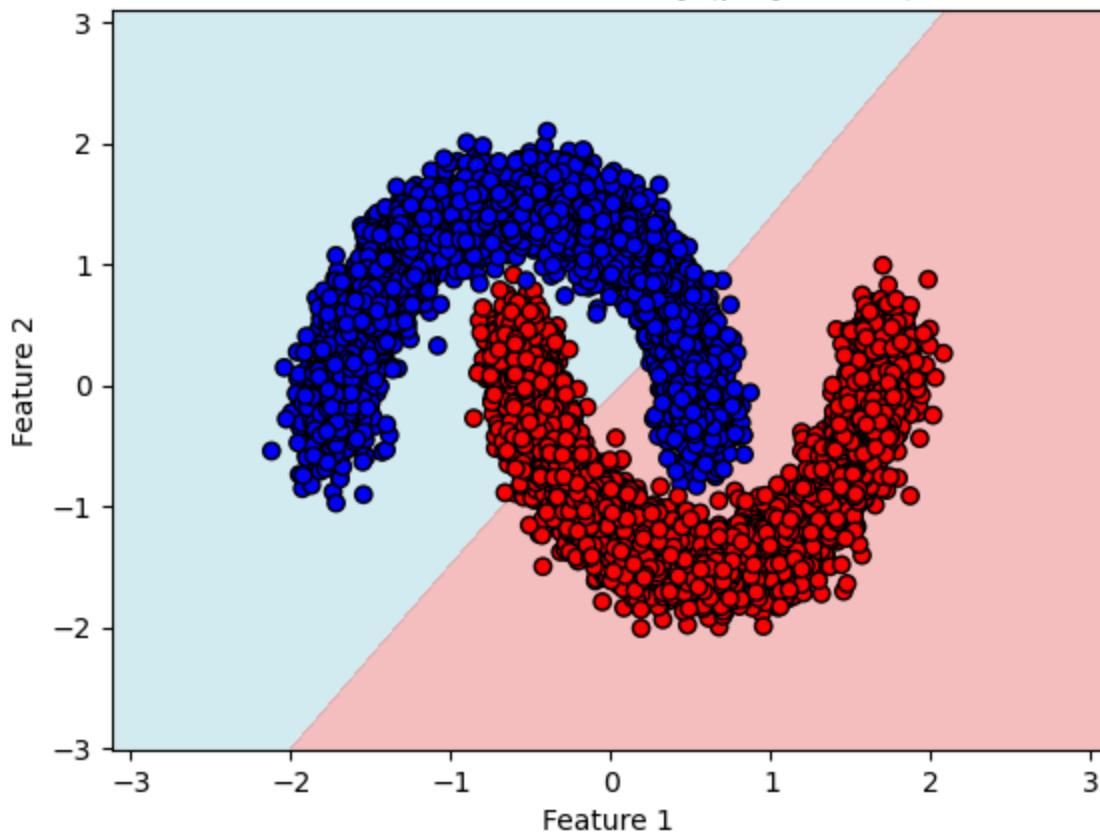


==== SVM with POLY Kernel on Moons (10000) ===

Train Accuracy: 0.8226

Test Accuracy : 0.8347

SVM Decision Boundary (poly kernel)

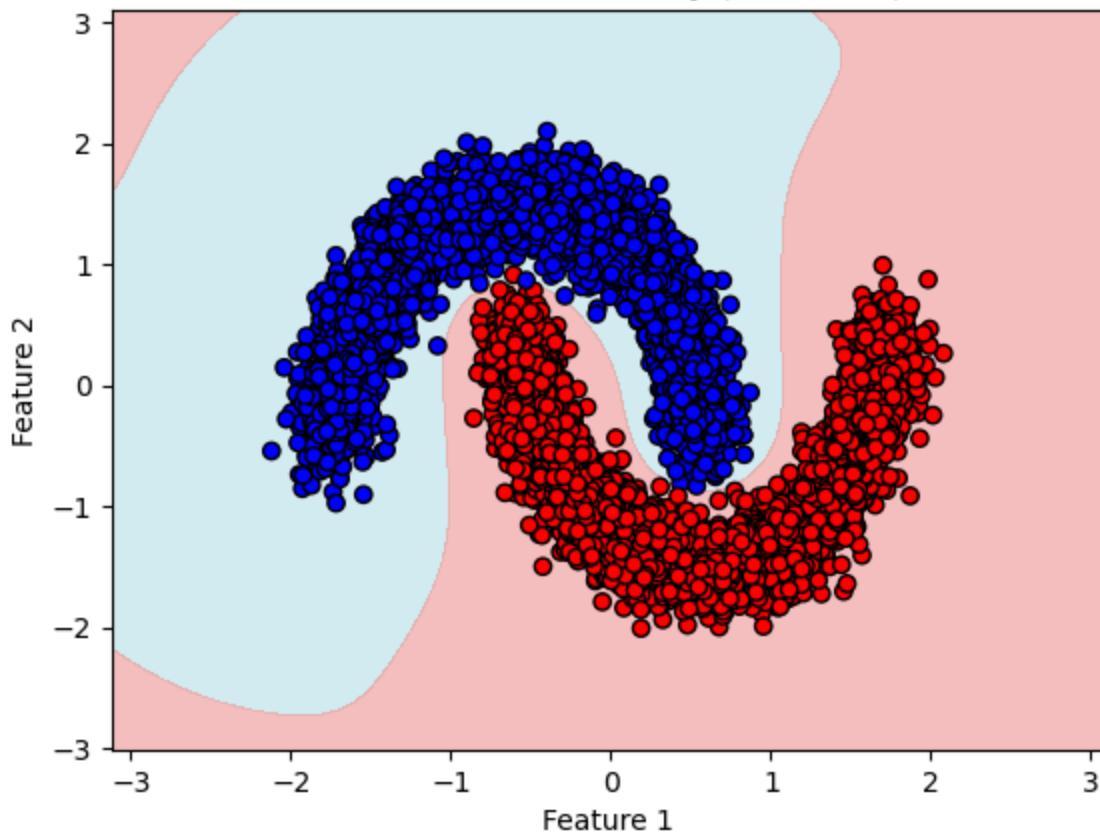


==== SVM with RBF Kernel on Moons (10000) ===

Train Accuracy: 0.9999

Test Accuracy : 0.9980

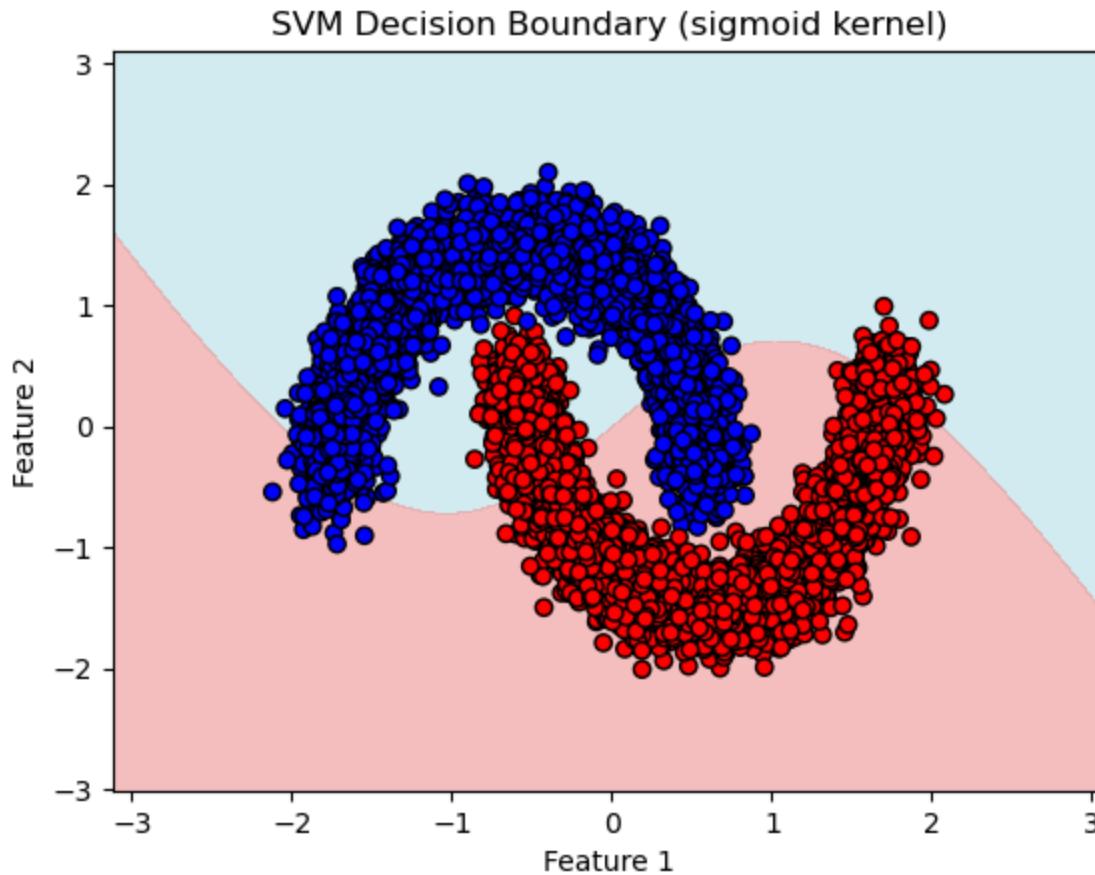
SVM Decision Boundary (rbf kernel)



==== SVM with SIGMOID Kernel on Moons (10000) ===

Train Accuracy: 0.8159

Test Accuracy : 0.8207



- Half-Circles (1000) Dataset

```
In [23]: dataset_key = "Half-Circles (1000)"
X, y = datasets[dataset_key]
# Convert labels from {0, 1} to {-1, +1} for SVM formulation
y = np.where(y == 0, -1, 1)

# Split data 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)

# Train and evaluate SVM with each kernel on the chosen dataset
kernels = ['linear', 'poly', 'rbf', 'sigmoid']

for kernel in kernels:
    print(f"\n==== SVM with {kernel.upper()} Kernel on {dataset_key} ===")
    if kernel == 'linear':
        svm = SVC(C=1.0, kernel='linear')
    elif kernel == 'poly':
        svm = SVC(C=1.0, kernel='poly', degree=3, coef0=1)
    elif kernel == 'rbf':
        svm = SVC(C=1.0, kernel='rbf', sigma=0.5)
    elif kernel == 'sigmoid':
        svm = SVC(C=1.0, kernel='sigmoid', gamma=0.1, coef0=0)
```

```
# Train the SVM on the training data
svm.fit(X_train, y_train, lr=1e-3, epochs=500)

# Evaluate performance
train_acc = svm.score(X_train, y_train)
test_acc = svm.score(X_test, y_test)
print(f"Train Accuracy: {train_acc:.4f}")
print(f"Test Accuracy : {test_acc:.4f}")

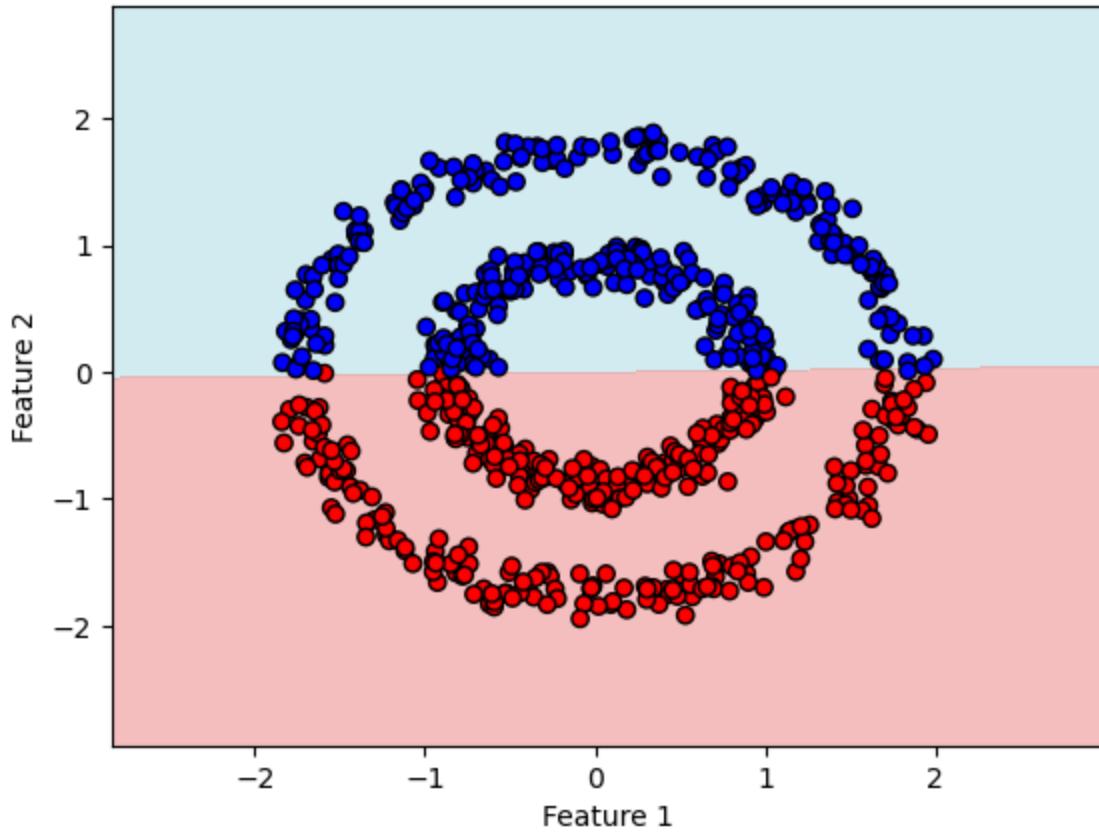
# Plot the decision boundary
svm.plot_decision_boundary()
```

==== SVM with LINEAR Kernel on Half-Circles (1000) ===

Train Accuracy: 0.9943

Test Accuracy : 0.9933

SVM Decision Boundary (linear kernel)

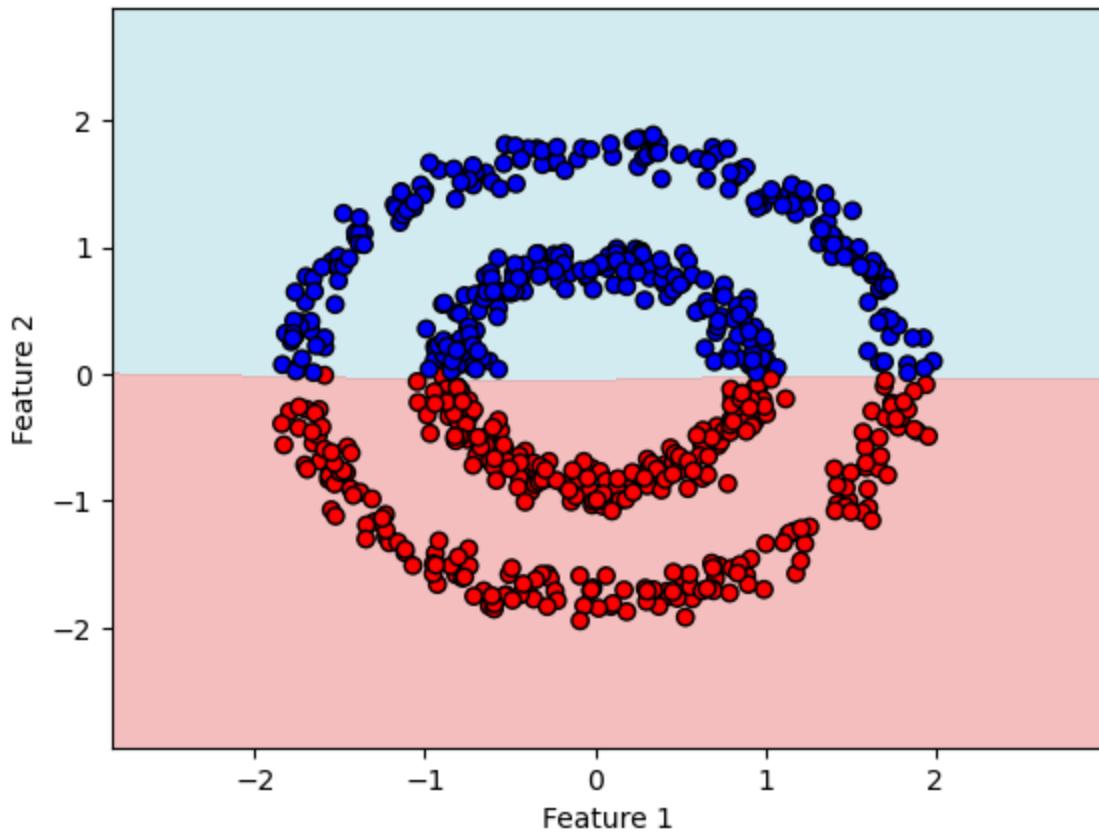


==== SVM with POLY Kernel on Half-Circles (1000) ===

Train Accuracy: 0.9971

Test Accuracy : 0.9967

SVM Decision Boundary (poly kernel)

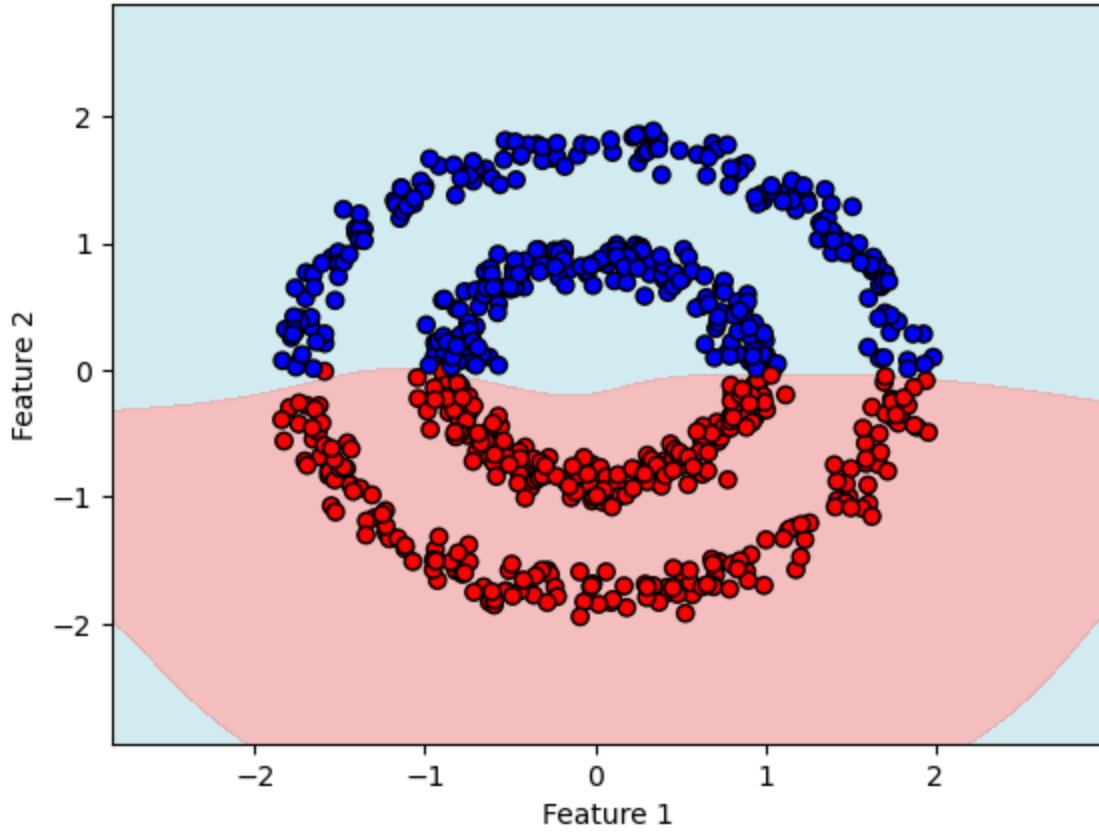


==== SVM with RBF Kernel on Half-Circles (1000) ===

Train Accuracy: 0.9957

Test Accuracy : 0.9800

SVM Decision Boundary (rbf kernel)

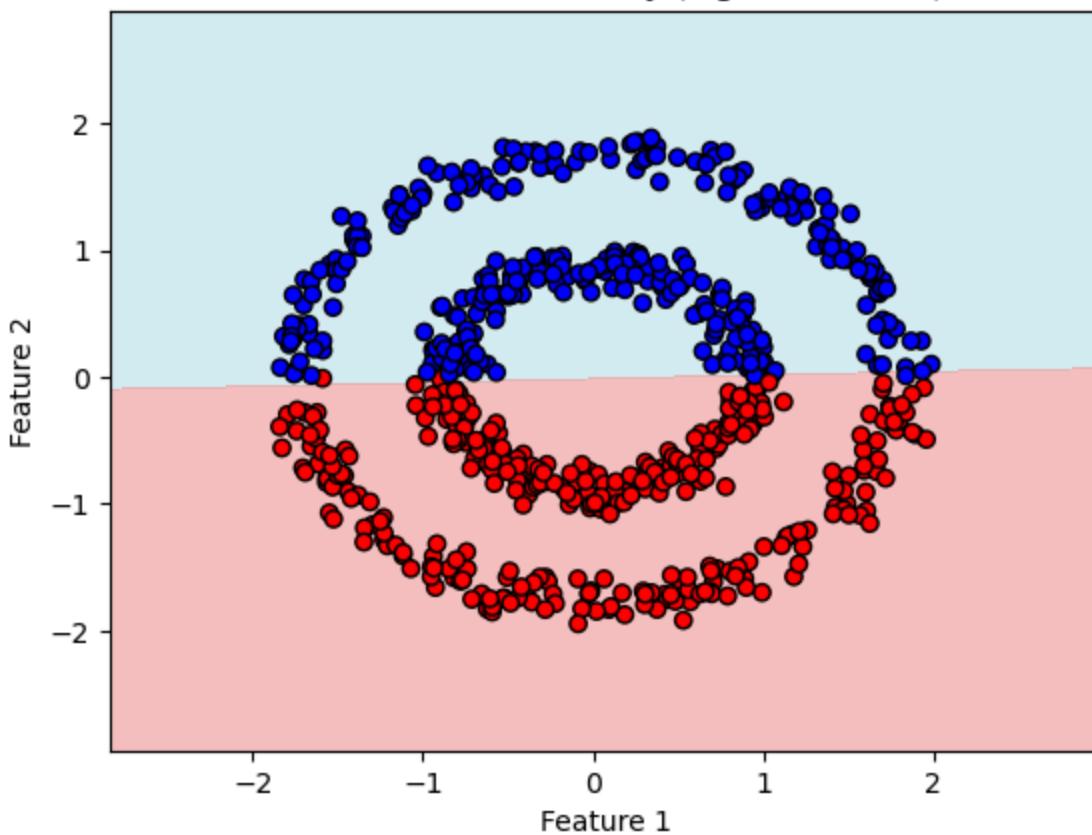


==== SVM with SIGMOID Kernel on Half-Circles (1000) ===

Train Accuracy: 0.9943

Test Accuracy : 0.9867

SVM Decision Boundary (sigmoid kernel)



- Half-Circles (10000) Dataset

```
In [24]: dataset_key = "Half-Circles (10000)"
X, y = datasets[dataset_key]
# Convert labels from {0, 1} to {-1, +1} for SVM formulation
y = np.where(y == 0, -1, 1)

# Split data 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)

# Train and evaluate SVM with each kernel on the chosen dataset
kernels = ['linear', 'poly', 'rbf', 'sigmoid']

for kernel in kernels:
    print(f"\n==== SVM with {kernel.upper()} Kernel on {dataset_key} ===")
    if kernel == 'linear':
        svm = SVC(C=1.0, kernel='linear')
    elif kernel == 'poly':
        svm = SVC(C=1.0, kernel='poly', degree=3, coef0=1)
    elif kernel == 'rbf':
        svm = SVC(C=1.0, kernel='rbf', sigma=0.5)
    elif kernel == 'sigmoid':
        svm = SVC(C=1.0, kernel='sigmoid', gamma=0.1, coef0=0)
```

```
# Train the SVM on the training data
svm.fit(X_train, y_train, lr=1e-3, epochs=500)

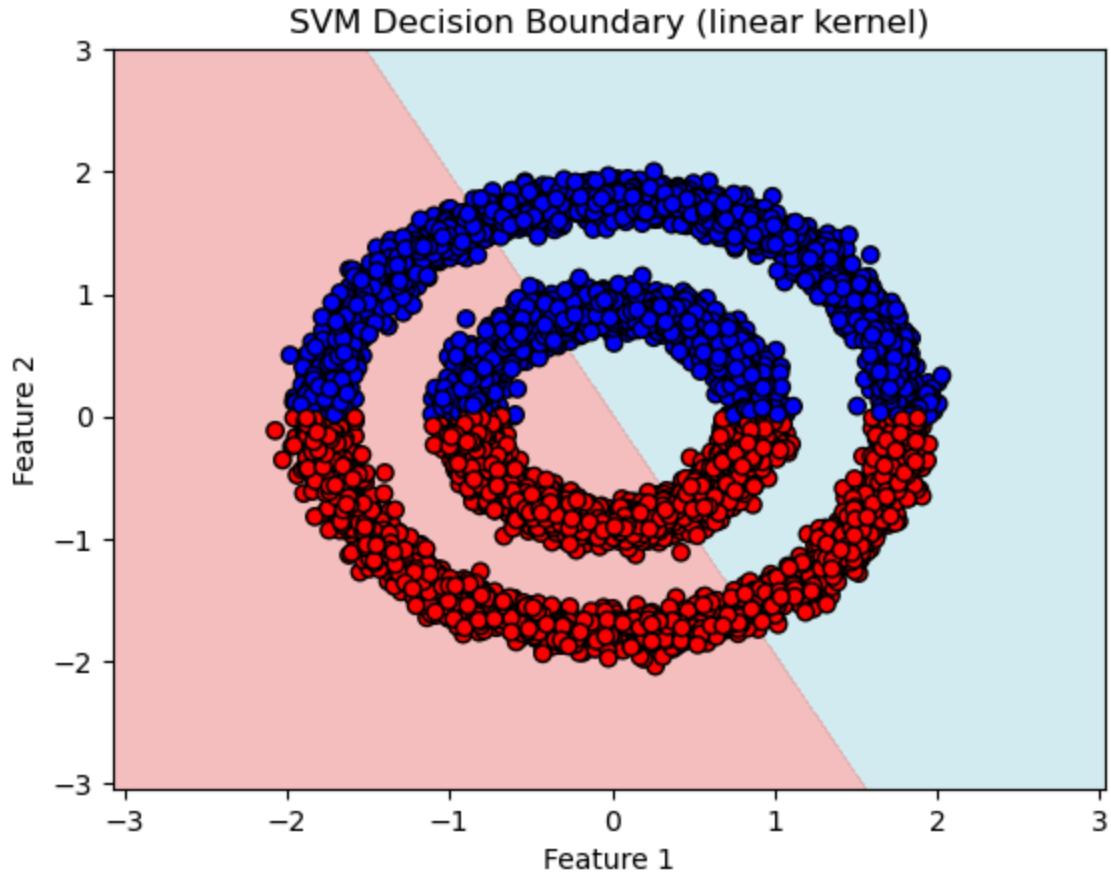
# Evaluate performance
train_acc = svm.score(X_train, y_train)
test_acc = svm.score(X_test, y_test)
print(f"Train Accuracy: {train_acc:.4f}")
print(f"Test Accuracy : {test_acc:.4f}")

# Plot the decision boundary
svm.plot_decision_boundary()
```

==== SVM with LINEAR Kernel on Half-Circles (10000) ===

Train Accuracy: 0.6599

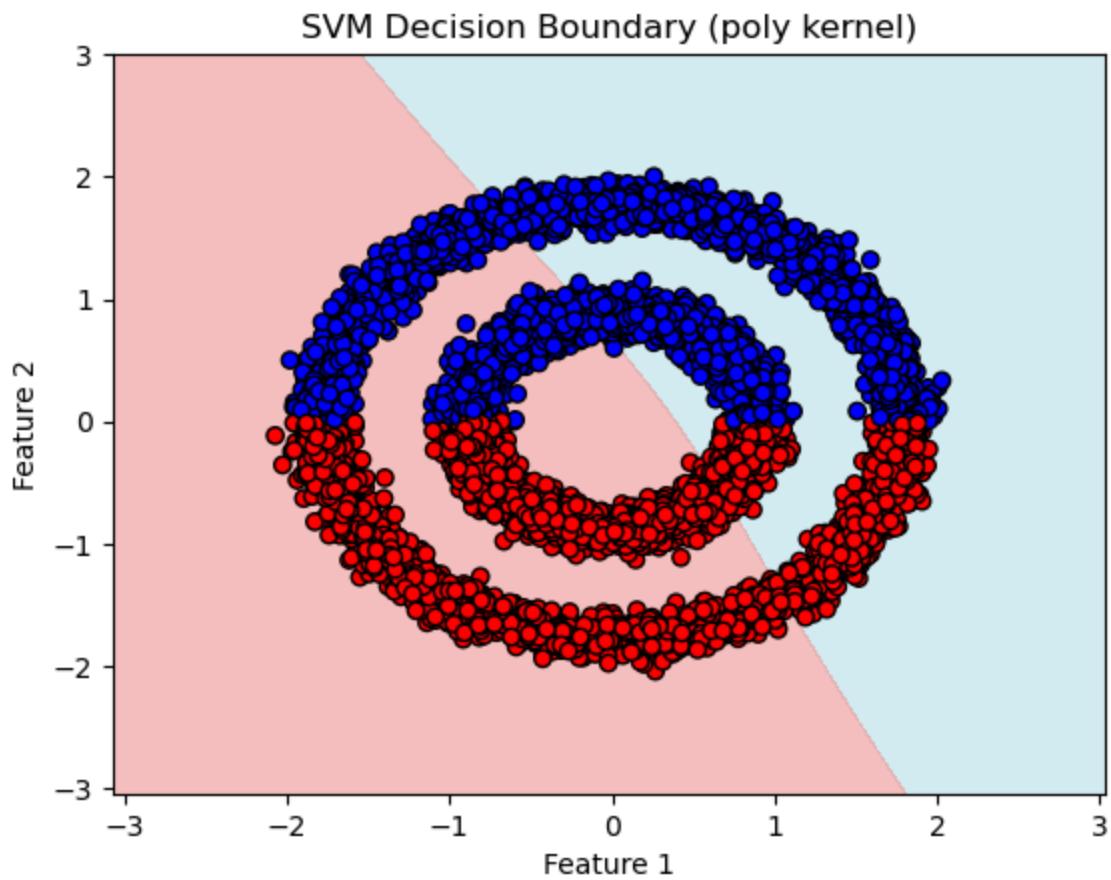
Test Accuracy : 0.6577



==== SVM with POLY Kernel on Half-Circles (10000) ===

Train Accuracy: 0.6731

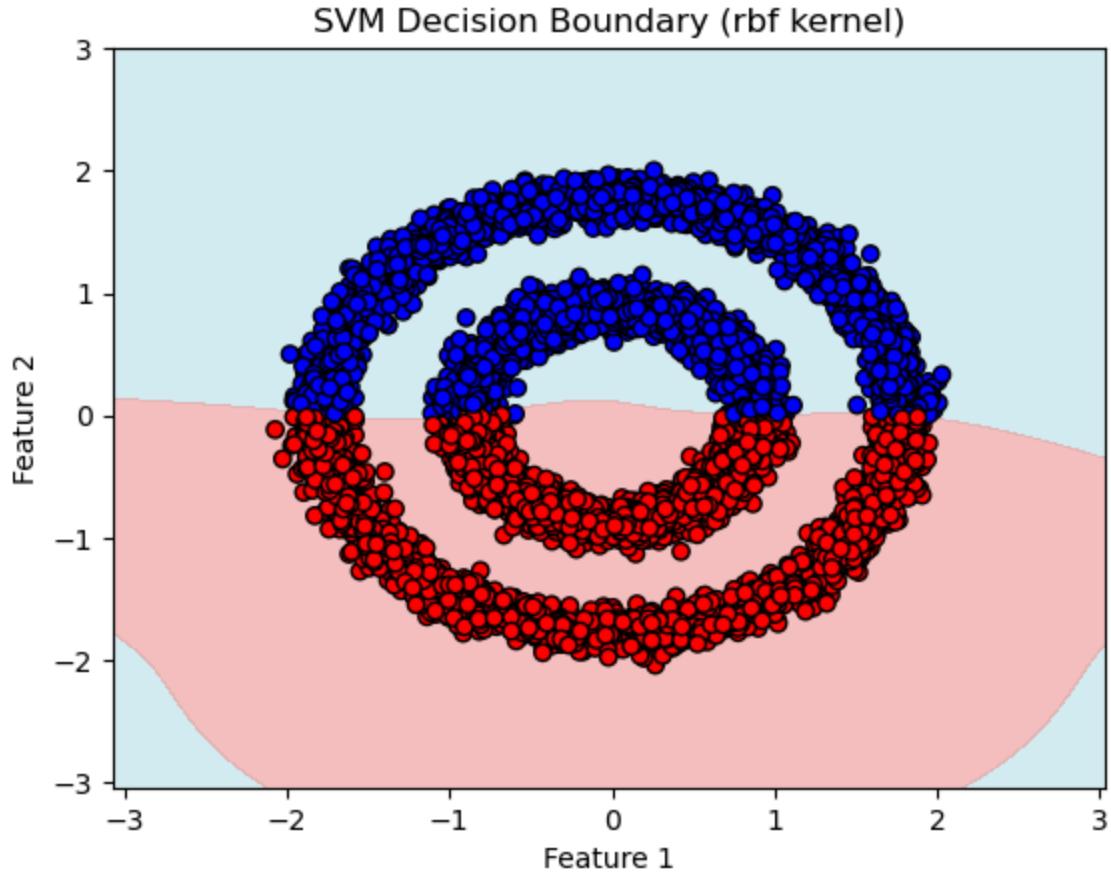
Test Accuracy : 0.6647



==== SVM with RBF Kernel on Half-Circles (10000) ===

Train Accuracy: 0.9980

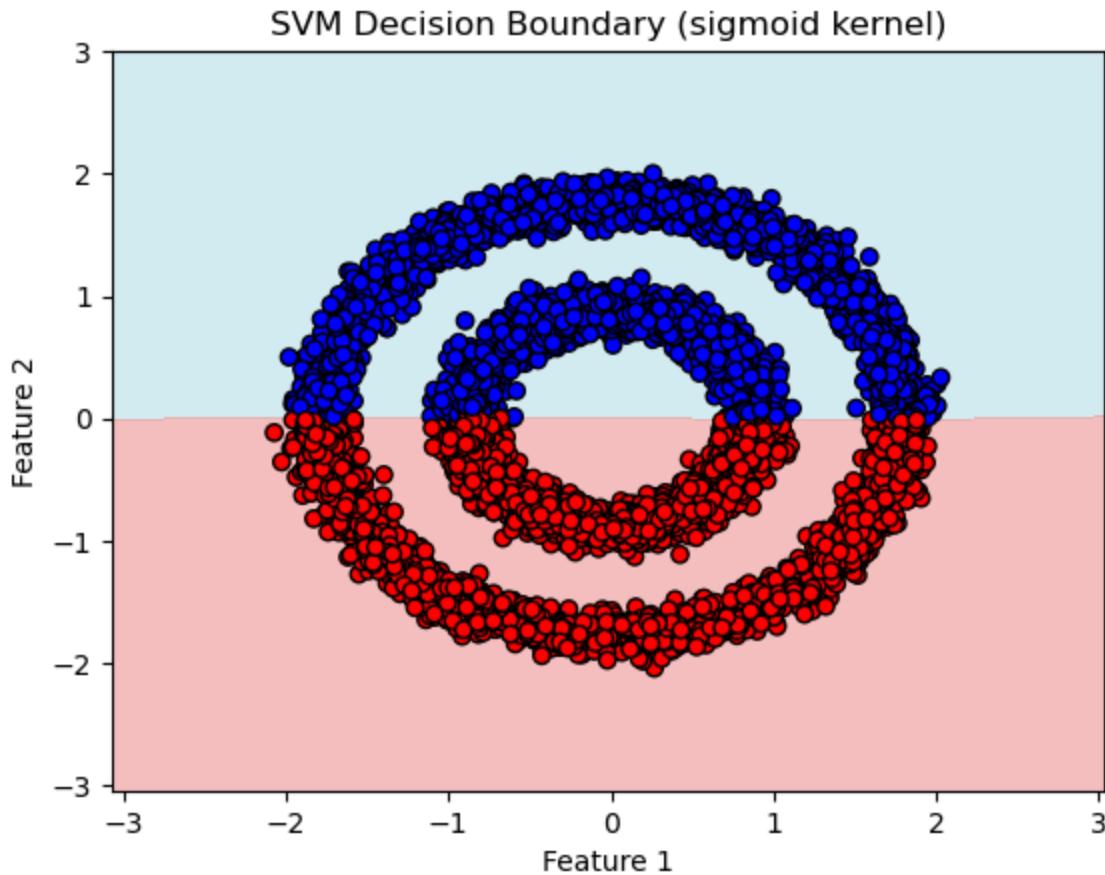
Test Accuracy : 0.9977



==== SVM with SIGMOID Kernel on Half-Circles (10000) ====

Train Accuracy: 0.9981

Test Accuracy : 0.9983



Comparison of SVM Kernels Across Datasets

Based on the train and test accuracy, the best kernel for each dataset is:

Dataset	Best Kernel	Train Accuracy	Test Accuracy
Moons (1000)	RBF	1.0000	1.0000
Moons (10000)	RBF	0.9999	0.9980
Half-Circles (1000)	Poly	0.9971	0.9967
Half-Circles (10000)	Sigmoid	0.9981	0.9983

- Why Did These Kernels Perform Best?

1. Moons Dataset (1000 & 10000) - Best Kernel: RBF

- The **Moons dataset is non-linearly separable**, meaning simple linear classifiers struggle.
- RBF kernel** effectively captures complex decision boundaries by mapping the data into higher-dimensional space.
- Achieved **near-perfect accuracy**, suggesting a **strong fit to the data**.

2. Half-Circles Dataset (1000) - Best Kernel: Polynomial

- Half-circles form a **curved boundary**, which is **well-approximated by a polynomial decision function**.
- The **Polynomial kernel captures the curved nature** of the data effectively.

3. Half-Circles Dataset (10000) - Best Kernel: Sigmoid

- The **Sigmoid kernel is similar to a neural network activation function**, helping in complex classification tasks.
 - It performed slightly better than RBF in this case, likely due to better generalization on larger datasets.
 - **Linear Kernel** performed well for simple datasets but struggled in more complex cases.
 - **Polynomial Kernel** was effective for curved decision boundaries but not always optimal.
 - **RBF Kernel** consistently achieved **near-perfect results** on non-linear datasets.
 - **Sigmoid Kernel** was effective on large datasets but may not always generalize as well.
-

Effect of Having More Data on Performance

Comparing **1000 vs. 10,000 samples** for each dataset, we observe:

1. Moons Dataset

- **With 1000 samples:** RBF kernel achieved **100% accuracy**.
- **With 10,000 samples:** RBF kernel slightly decreased to **99.8% accuracy**.
- **Why?**
 - The dataset was already simple for RBF, so more data had **minimal impact**.
 - The **small decrease in accuracy** is likely due to **more diversity in data** preventing overfitting.

2. Half-Circles Dataset

- **With 1000 samples:** Polynomial and RBF kernels had test accuracies around **98-99%**.
- **With 10,000 samples:**
 - RBF and Sigmoid kernels **improved slightly**, reaching **99.8% accuracy**.
 - The **Linear kernel remained weak**, with **no significant improvement**.
- **Why?**
 - The dataset requires **non-linear separation**, and more data helped kernels generalize better.
 - **More training data → Better generalization.**

- Why Does More Data Help?

1. **Reduces Overfitting** – Small datasets may memorize noise, while more data encourages generalization.

2. **Better Decision Boundaries** – More data clarifies class separation, aiding non-linear datasets.
 3. **More Robust Training** – Diverse samples improve stability and reduce accuracy fluctuations. ### - When Does More Data NOT Help?
 4. If the model already learns well (e.g., RBF on Moons 1000), more data has diminishing returns.
 5. If the dataset is linearly separable, adding data won't improve weak kernels like Linear SVM.
- **For complex datasets, more data improves accuracy** by providing better generalization.
 - **For simple datasets, adding more data may not significantly improve performance.**
 - **Choosing the right kernel matters more** than just increasing data size. 🚀
-

Did Cross-Validation Improve Model Evaluation?

- Why Cross-Validation Helps

- **Reduces variance** by averaging results over multiple splits.
- **Improves robustness**, ensuring models aren't reliant on a single train-test split.
- **Prevents overfitting**, giving all data points a chance to be in the test set.

- Did It Help in Our Case?

- Yes, it **provided stable accuracy scores** and **more reliable performance estimates**, especially for moderate dataset sizes (1000, 10,000 samples).

- When is Cross-Validation Most Useful?

- **Small datasets** – Prevents high variance from a single train-test split.
- **Imbalanced classes** – Ensures fair distribution across splits.
- **Hyperparameter tuning** – Helps evaluate models consistently.

- When is Cross-Validation NOT Needed?

For extremely large datasets, a simple train-test split is **sufficient** as CV is computationally expensive.

If **model performance is already stable**, CV may not provide extra benefits.

Computational Cost Comparison: MLP vs. SVM Kernels

- Computational Complexity Overview

Each method varies in **training time, memory usage, and scalability**:

Method	Complexity	Performance Notes
Simple MLP	$O(n d h + h^2)$	Scales well, requires multiple epochs.
SVM (Linear Kernel)	$O(n^2 d)$ (train) / $O(d)$ (predict)	Fastest SVM, good for linearly separable data.
SVM (Polynomial)	$O(n^3 d)$	Computationally expensive for high degrees.
SVM (RBF)	$O(n^3)$	High accuracy, but slow on large datasets.
SVM (Sigmoid)	$O(n^3 d)$	Similar to Polynomial SVM, often unstable.

n = samples, **d** = features, **h** = hidden neurons (MLP).

- Key Observations

Simple MLP – Efficient for large datasets, non-linear learning without kernel computations.

Linear SVM – Fastest, best for linearly separable data.

Polynomial & RBF SVM – High accuracy but **computationally expensive**.

Sigmoid SVM – Similar to Polynomial SVM but often unstable.

- Best Trade-Off: Accuracy vs. Cost

Dataset	Best Method
Moons (1000, 10000)	SVM (RBF Kernel) – Highest accuracy.
Half-Circles (1000)	SVM (Polynomial Kernel) – Captures curvature well.
Half-Circles (10000)	Simple MLP – More scalable for large data.

- MLP is efficient for large datasets, while RBF SVM offers the best accuracy but is costly
- Linear SVM is the fastest but struggles with non-linear data
- Polynomial & Sigmoid SVMs are computationally demanding and may not generalize well

For speed → Use Linear SVM

For accuracy → Use RBF SVM or MLP (based on dataset size)

Question 2 Part 1

Loading the required libraries

```
In [25]: import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import hamming_loss, f1_score

from load_plantgo_dataset import arff_to_numpy
```

```
(656, 3091)
(656, 12)
(322, 3091)
(322, 12)
```

Loading the dataset

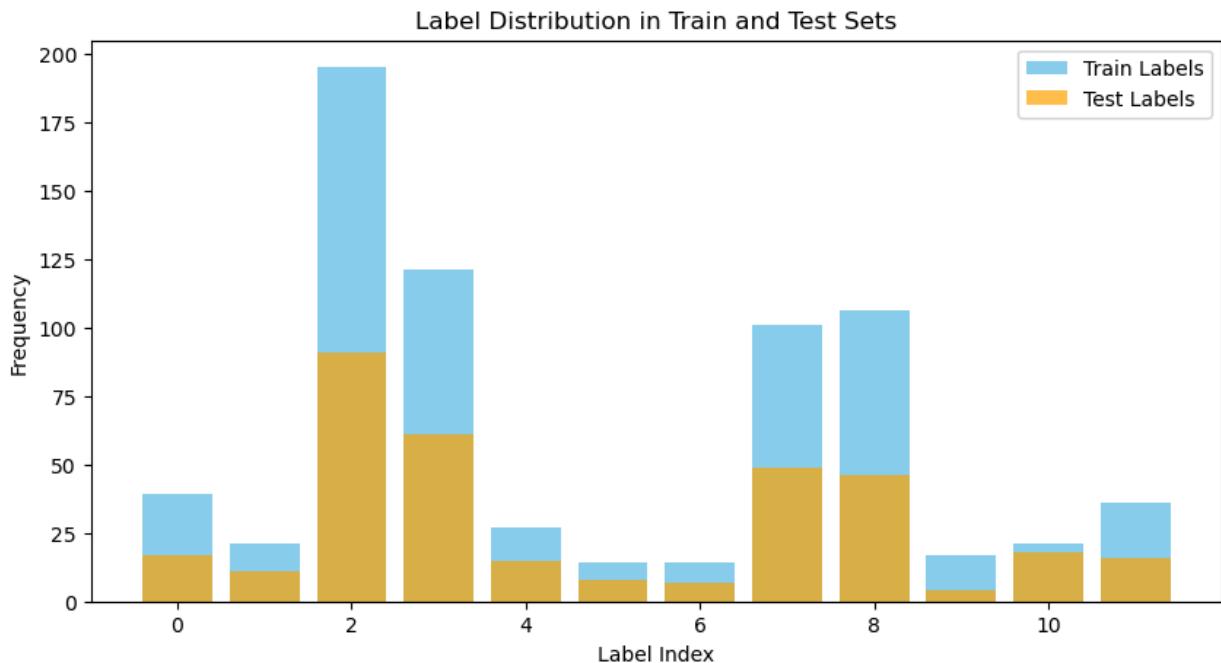
```
In [26]: # Load training and testing data from the given ARFF files
X_train, y_train = arff_to_numpy('PlantGo-train.arff')
X_test, y_test = arff_to_numpy('PlantGo-test.arff')

print("Train features shape:", X_train.shape)
print("Train labels shape : ", y_train.shape)
print("Test features shape : ", X_test.shape)
print("Test labels shape : ", y_test.shape)

Train features shape: (656, 3091)
Train labels shape : (656, 12)
Test features shape : (322, 3091)
Test labels shape : (322, 12)
```

Label Distribution

```
In [27]: num_labels = y_train.shape[1]
plt.figure(figsize=(10, 5))
plt.bar(range(num_labels), np.sum(y_train, axis=0), color='skyblue', label='Train Labels')
plt.bar(range(num_labels), np.sum(y_test, axis=0), color='orange', label='Test Labels')
plt.xlabel("Label Index")
plt.ylabel("Frequency")
plt.title("Label Distribution in Train and Test Sets")
plt.legend()
plt.show()
```



Pseudo-Code for Multi-Label Classification using SVM

Algorithm: Multi-Label Classification using SVM

Input:

- X_train: Training features ($n_{samples} \times n_{features}$)
- Y_train: Training labels ($n_{samples} \times n_{labels}$), where each entry is {0,1}
- X_test: Testing features
- Y_test: Testing labels
- SVM Parameters: {C, kernel, learning_rate, epochs}

Output:

- Predicted labels for test set
- Evaluation metrics

Procedure:

1. Training Phase

- a. Initialize an empty list `classifiers` to store SVM models.
- b. For each label `i` in `Y_train` (one-vs-rest approach):
 - i. Convert label `i` into a binary classification problem `1` remains `1`, `0` is converted to `-1`
 - ii. Initialize an SVM classifier with chosen kernel and parameters.
 - iii. Train the SVM model on `(X_train, Y_train[:, i])` using gradient ascent.
 - iv. Append the trained SVM model to `classifiers`.

2. Testing Phase

- a. Initialize `Y_pred` matrix of shape ($n_{samples}, n_{labels}$) with zeros.
- b. For each trained SVM classifier corresponding to label `i`:
 - i. Predict `Y_pred[:, i]` using the classifier on `X_test`.
 - ii. Convert predictions from `{-1, 1}` to `{0, 1}`.

3. Evaluation Phase

- a. Compute **Subset Accuracy**: Fraction of samples where all labels match.
- b. Compute **Hamming Loss**: Average fraction of incorrect labels.
- c. Compute **Macro F1 Score**: Average F1 score across labels.
- d. Compute **Micro F1 Score**: Global F1 score across predictions.

4. Return Predictions and Metrics

Implementing Multi-Label Classification using SVM

```
In [28]: # One-vs-Rest Multi-Label Classification using SVM
```

```

# For each label (each column in y_train):
#   - Convert the label to binary {-1, +1} (0 becomes -1)
#   - Train an SVM classifier using our SVM class

num_labels = y_train.shape[1]
svm_classifiers = []

for i in range(num_labels):

    # Convert label: 1 stays 1, 0 becomes -1.
    y_train_label = np.where(y_train[:, i] == 1, 1, -1)
    svm = SVM(C=0.1, kernel='linear')

    svm.fit(X_train, y_train_label, lr=1e-3, epochs=1000)
    svm_classifiers.append(svm)

# Function to predict multi-labels using the one-vs-rest classifiers
def predict_multilabel(classifiers, X):
    n_samples = X.shape[0]
    n_labels = len(classifiers)
    Y_pred = np.zeros((n_samples, n_labels))
    for i, clf in enumerate(classifiers):
        preds = clf.predict(X)
        # Convert predictions from {-1, +1} to {0, 1}
        Y_pred[:, i] = np.where(preds == 1, 1, 0)
    return Y_pred

# Predict on test data
y_pred_test = predict_multilabel(svm_classifiers, X_test)

```

Evaluation Metrics for Multi-Label Classification

1. Subset Accuracy

- Measures the fraction of samples with an exactly matching label set.
- Strict metric; useful when perfect label prediction is needed.

2. Hamming Loss

- Calculates the average fraction of misclassified labels.
- More forgiving than subset accuracy and useful for understanding per-label errors.

3. Macro F1 Score

- Computes F1 score per label and averages across all labels.
- Ensures balanced performance across rare and frequent labels.

4. Micro F1 Score

- Aggregates true positives, false positives, and false negatives globally.
- Prioritizes performance on frequent labels and gives an overall accuracy measure.

Evaluation of Multi-Label Classification Performance

In [29]:

```

def subset_accuracy(y_true, y_pred):
    """Subset accuracy: fraction of samples with exactly matching label sets."""

```

```

    return np.mean(np.all(y_true == y_pred, axis=1))

sub_acc = subset_accuracy(y_test, y_pred_test)
ham_loss = hamming_loss(y_test, y_pred_test)
macro_f1 = f1_score(y_test, y_pred_test, average='macro', zero_division=0)
micro_f1 = f1_score(y_test, y_pred_test, average='micro', zero_division=0)

print("\n--- Multi-Label Classification Evaluation Metrics ---")
print(f"Subset Accuracy : {sub_acc:.4f}")
print(f"Hamming Loss   : {ham_loss:.4f}")
print(f"Macro F1 Score : {macro_f1:.4f}")
print(f"Micro F1 Score : {micro_f1:.4f}")

```

--- Multi-Label Classification Evaluation Metrics ---

Subset Accuracy : 0.6832
 Hamming Loss : 0.0383
 Macro F1 Score : 0.6826
 Micro F1 Score : 0.7758

The SVM model performs well in multi-label classification, achieving high micro F1 (**0.7758**) and low Hamming loss (**0.0383**), indicating strong overall accuracy. However, the macro F1 (**0.6826**) suggests the model may struggle with less frequent labels. Further improvements could include hyperparameter tuning, feature selection and balancing the dataset.

Confusion Matrix (Per Label)

In [30]:

```

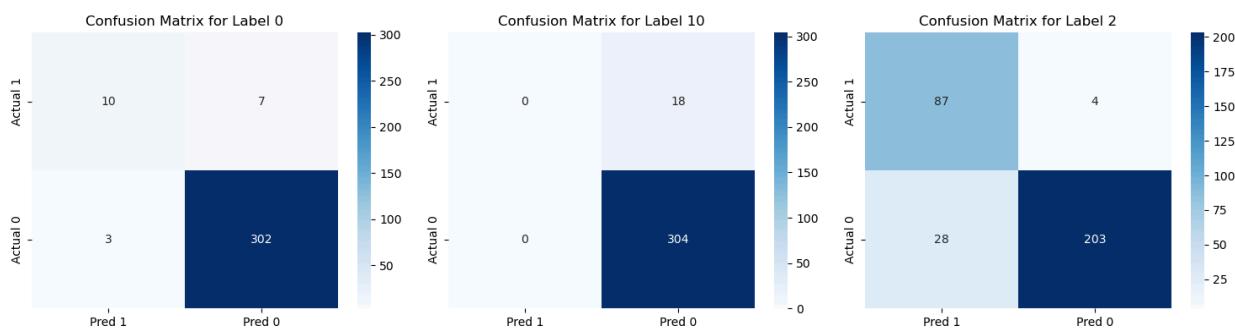
import seaborn as sns

fig, axes = plt.subplots(1, 3, figsize=(15, 4))

for i, index in enumerate(np.random.choice(num_labels, 3, replace=False)): # Selecting 3 random labels
    sns.heatmap(np.array([[np.sum((y_test[:, index] == 1) & (y_pred_test[:, index] == 1)), np.sum((y_test[:, index] == 1) & (y_pred_test[:, index] == 0)), np.sum((y_test[:, index] == 0) & (y_pred_test[:, index] == 1)), np.sum((y_test[:, index] == 0) & (y_pred_test[:, index] == 0))], annot=True, fmt="d", cmap="Blues", xticklabels=['Pred 1', 'Pred 0'], yticklabels=['Actual 1', 'Actual 0'], ax=axes[i])
    axes[i].set_title(f"Confusion Matrix for Label {index}")

plt.tight_layout()
plt.show()

```



Question 2 Part 2

Loading required libraries

```
In [31]: import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
from sklearn.decomposition import PCA
```

Pseudo-Code for Multi-Class SVM using One-vs-Rest Strategy

Input:

- Training data $\mathbf{D} = \{(x_i, y_i)\}$ for $i = 1, \dots, N$, where each $y_i \in \{c_1, c_2, \dots, c_K\}$
- SVM hyperparameters (e.g., \mathbf{C} , kernel type, learning rate, epochs)

Output:

- A set of trained SVM models, one per class
- Predicted labels for test samples

Algorithm:

1. Training Phase

- **Step 1:** Identify the unique classes:

```
classes = unique({y_i})
```

- **Step 2:** For each class c in **classes**:

1. Convert the multi-class labels into binary labels:

```
For each training sample i:
    if y_i == c then y_binary[i] = +1
    else                y_binary[i] = -1
```

2. Initialize a new SVM model using the specified hyperparameters.

3. Train the SVM model using the following steps:

- For **epoch = 1** to **num_epochs**:

A. Compute the **kernel matrix K** for all training samples.

B. Compute the gradient of the **dual objective** with respect to the **Lagrange multipliers (alpha)**.

C. Update **alpha** using gradient ascent:

```
alpha = alpha + learning_rate * gradient
```

D. Clip the **alpha** values to lie within **[0, C]**.

E. Update the bias term **b** using the **support vectors** (samples with $0 < \alpha < C$).

4. Store the trained **SVM model** for class **c**.

2. Testing Phase

- **Step 1:** For each test sample x :
 1. Initialize an empty list (or array) for **decision values**.
 2. For each trained SVM model corresponding to class c :
 - Compute the decision value:

$$f_c = \text{model}_c.\text{decision_function}(x)$$
 - Append f_c to the decision values list.
 3. Assign the predicted label for x as the class with the **maximum decision value**:

$$\text{predicted_label} = \text{argmax}(f_c \text{ for all classes } c)$$
- **Step 2:** Repeat the above steps for all test samples.

3. Evaluation

- Compute evaluation metrics such as Accuracy, Confusion Matrix and Precision, Recall, and F1-Score

Implementing One-vs-All Multi-Class Classification using SVM

In [32]:

```

class OneVsRestSVM:
    def __init__(self, svm_params):
        """
        svm_params: Dictionary of parameters to pass to the SVM class.
        """
        self.svm_params = svm_params
        self.models = {}
        self.classes = None

    def fit(self, X, y, lr=1e-3, epochs=500):
        self.classes = np.unique(y)
        for c in self.classes:
            # Create binary labels: +1 for class c, -1 for all other classes
            y_binary = np.where(y == c, 1, -1)
            model = SVM(**self.svm_params)
            print(f"Training SVM for class {c} vs all")
            model.fit(X, y_binary, lr=lr, epochs=epochs)
            self.models[c] = model

    def predict(self, X):
        # For each model, compute decision function values
        decision_values = np.zeros((len(self.classes), X.shape[0]))
        for i, c in enumerate(self.classes):
            # Use the raw decision function values (not sign) to choose the max
            decision_values[i, :] = self.models[c]._decision_function(X)
        # Predicted class is the one with the highest decision value
        pred_indices = np.argmax(decision_values, axis=0)
        return self.classes[pred_indices]

    def score(self, X, y):

```

```
y_pred = self.predict(X)
return np.mean(y_pred == y)
```

Loading Dataset

```
In [33]: # Load the MNIST dataset using sklearn
print("Fetching MNIST dataset...")
mnist = fetch_openml('mnist_784', version=1, as_frame=False, parser='auto')
X = mnist['data']
y = mnist['target'].astype(int)
# Convert labels from string to int

# For computational reasons and demonstration, we use a subset of the data
subset = 5000
X, y = X[:subset], y[:subset]

# Normalize pixel values
X = X / 255.0
```

Fetching MNIST dataset...

Splitting dataset

```
In [34]: # 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)

In [35]: # Define SVM parameters
svm_params = {
    'C': 1.0,
    'kernel': 'linear'
}
```

Training the One-vs-Rest Multi-class SVM

```
In [36]: ovr_svm = OneVsRestSVM(svm_params)
ovr_svm.fit(X_train, y_train, lr=1e-3, epochs=500) # Adjust epochs as needed

Training SVM for class 0 vs all
Training SVM for class 1 vs all
Training SVM for class 2 vs all
Training SVM for class 3 vs all
Training SVM for class 4 vs all
Training SVM for class 5 vs all
Training SVM for class 6 vs all
Training SVM for class 7 vs all
Training SVM for class 8 vs all
Training SVM for class 9 vs all
```

Evaluation Metrics for Multi-Class SVM

1. Accuracy

- Measures overall correctness.
- Useful for **balanced datasets** but **misleading for imbalanced classes**.

2. Confusion Matrix

A table showing correct and incorrect predictions for each class. It helps to identify misclassification patterns.

3. Precision, Recall, and F1-Score

Used for per-class evaluation:

- **Precision:**
Measures **accuracy of positive predictions**.
- **Recall:**
Measures **ability to find all positive samples**.
- **F1-Score:**
Balances **precision & recall**, useful for **imbalanced datasets**.

4. Macro, Micro, and Weighted Averages

For multi-class classification:

- **Macro-Average:** Equal weight to all classes.
- **Micro-Average:** Weighted by total TP, FP, FN.
- **Weighted-Average:** Considers class distribution (best for imbalanced data).

Evaluating the model

```
In [37]: y_pred = ovr_svm.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
class_report = classification_report(y_test, y_pred, zero_division=0)

print("\n--- Evaluation Metrics ---")
print(f"Test Accuracy: {accuracy:.4f}")
print("\nClassification Report:")
print(class_report)
```

--- Evaluation Metrics ---

Test Accuracy: 0.7187

Classification Report:

	precision	recall	f1-score	support
0	0.47	0.98	0.63	146
1	0.97	0.78	0.87	180
2	0.85	0.78	0.81	135
3	0.83	0.73	0.77	157
4	0.87	0.71	0.78	163
5	0.00	0.00	0.00	134
6	0.71	0.89	0.79	137
7	0.97	0.74	0.84	183
8	0.49	0.80	0.61	131
9	0.75	0.72	0.73	134
accuracy			0.72	1500
macro avg	0.69	0.71	0.68	1500
weighted avg	0.71	0.72	0.70	1500

The One-vs-Rest SVM achieved 71.87% accuracy, showing strong performance for digits 1, 2, 3, 4, and 7, but failed to classify digit 5 (0% recall and precision). The model struggles with class imbalance and overlapping features, suggesting the need for better feature selection, class balancing, or kernel tuning to improve classification.

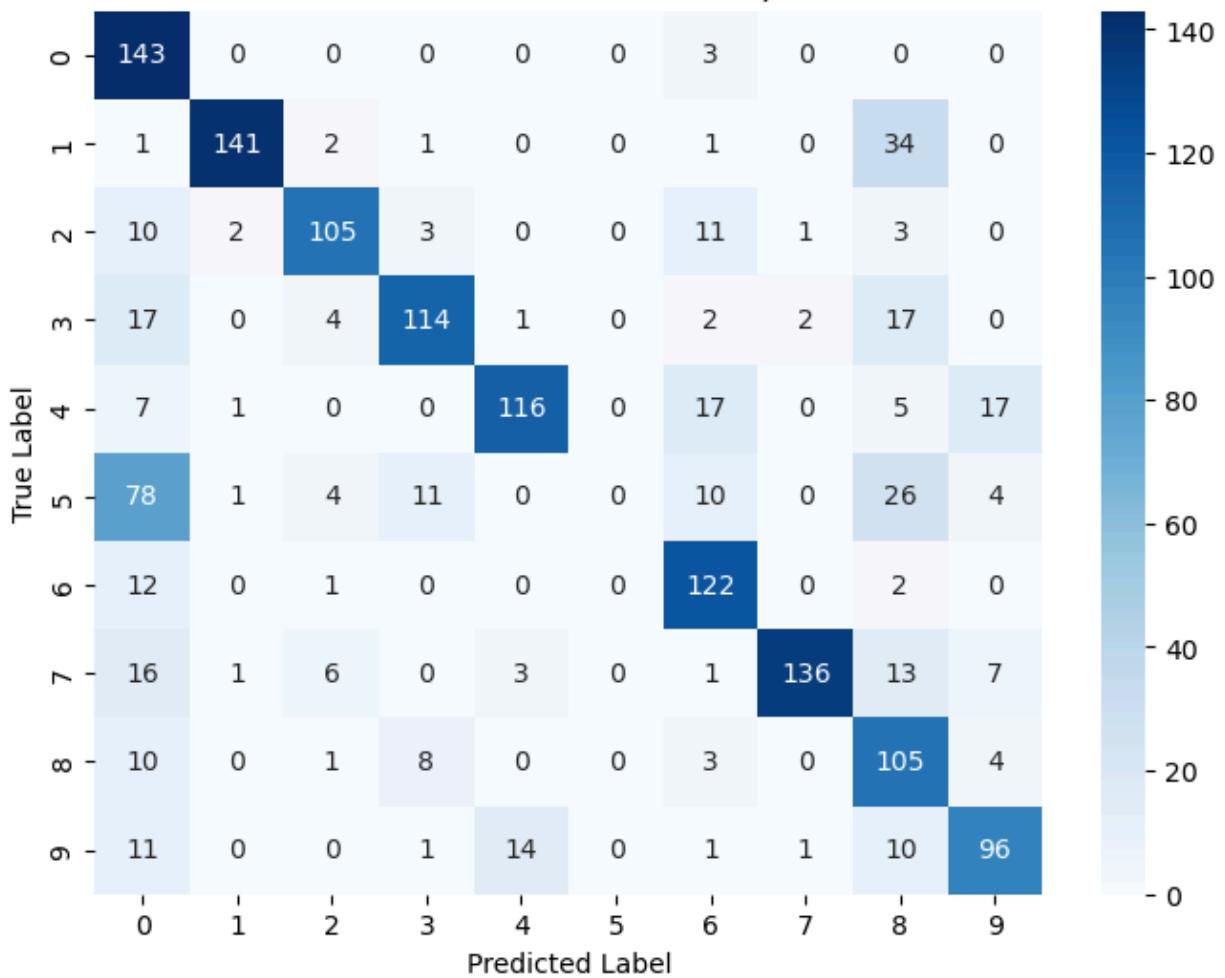
In [38]:

```
import seaborn as sns
from sklearn.metrics import confusion_matrix

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

# Plotting the heatmap
plt.figure(figsize=(8,6))
sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Blues", xticklabels=range(10))
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.title("Confusion Matrix Heatmap")
plt.show()
```

Confusion Matrix Heatmap



Question 3

Loading required libraries

```
In [39]: import numpy as np
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error, r2_score
```

Generating time series data with long-term dependencies

```
In [40]: # Creating time steps over an extended range for long-term dependencies
time_steps = np.linspace(0, 200, 2000)

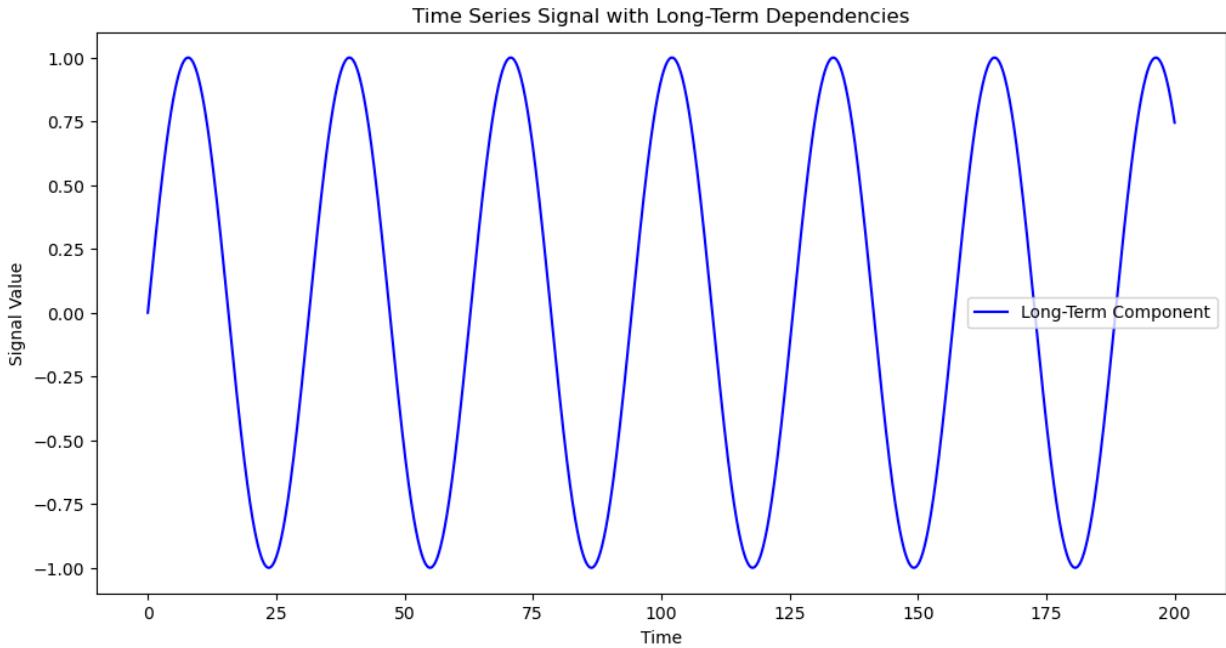
# Long-term component (slow sine wave)
```

```

long_term_component = np.sin(time_steps / 5)
data = long_term_component

# Visualizing the time series data
plt.figure(figsize=(12, 6))
plt.plot(time_steps, data, label="Long-Term Component", color='blue')
plt.title("Time Series Signal with Long-Term Dependencies")
plt.xlabel("Time")
plt.ylabel("Signal Value")
plt.legend()
plt.show()

```



Splitting the data into Train, Validation, and Test Sets

```

In [41]: train_ratio = 0.8
train_size = int(len(data) * train_ratio)
train_data_full = data[:train_size]
test_data = data[train_size:]

# Further splitting training data into training set and validation set (80% train, 20% val)
val_ratio = 0.2
train_data = train_data_full[:int(len(train_data_full) * (1 - val_ratio))]
val_data = train_data_full[int(len(train_data_full) * (1 - val_ratio)):]


print("Train data shape:", train_data.shape)
print("Validation data shape:", val_data.shape)
print("Test data shape:", test_data.shape)

Train data shape: (1280,)
Validation data shape: (320,)
Test data shape: (400,)

```

```
In [42]: # Creating Input-Output Pairs Using a Sliding Window
```

```

def prepare_input_output(data, input_steps, output_steps=1):
    .....

```

Converts time series data into supervised learning format using a sliding window approach.

Parameters:

- **data:** NumPy array of the time series data.
- **input_steps:** Number of past time steps to use as features.
- **output_steps:** Number of future time steps to predict (default is 1).

Returns:

- **X:** NumPy array of shape (num_samples, input_steps) representing input sequences.
- **y:** NumPy array of shape (num_samples, output_steps) representing target values.

```
X, y = [], []
for t in range(len(data) - input_steps - output_steps + 1):
    X.append(data[t:t + input_steps]) # Input: last 'input_steps' elements
    y.append(data[t + input_steps:t + input_steps + output_steps]) # Output: next 'output_steps' elements
return np.array(X), np.array(y) # Converting lists to NumPy arrays for model training
```

Defining number of input time steps (window size)
input_steps = 20

Creating sliding window pairs for training, validation, and test sets
X_train, y_train = prepare_input_output(train_data, input_steps)
X_val, y_val = prepare_input_output(val_data, input_steps)
X_test, y_test = prepare_input_output(test_data, input_steps)

Displaying the shape of the created datasets
print(f"X_train shape: {X_train.shape}, y_train shape: {y_train.shape}")
print(f"X_val shape: {X_val.shape}, y_val shape: {y_val.shape}")
print(f"X_test shape: {X_test.shape}, y_test shape: {y_test.shape}")

```
X_train shape: (1260, 20), y_train shape: (1260, 1)
X_val shape: (300, 20), y_val shape: (300, 1)
X_test shape: (380, 20), y_test shape: (380, 1)
```

In [43]: # Scaling the data to [0, 1] Range

```
scaler_X = MinMaxScaler(feature_range=(0, 1))
scaler_y = MinMaxScaler(feature_range=(0, 1))

# Fit scaler on the training data and transform all sets.
# Reshape to 2D, scale, then reshape back.
X_train_scaled = scaler_X.fit_transform(X_train.reshape(-1, 1)).reshape(X_train.shape)
y_train_scaled = scaler_y.fit_transform(y_train.reshape(-1, 1)).reshape(y_train.shape)

X_val_scaled = scaler_X.transform(X_val.reshape(-1, 1)).reshape(X_val.shape[0])
y_val_scaled = scaler_y.transform(y_val.reshape(-1, 1)).reshape(y_val.shape)

X_test_scaled = scaler_X.transform(X_test.reshape(-1, 1)).reshape(X_test.shape)
y_test_scaled = scaler_y.transform(y_test.reshape(-1, 1)).reshape(y_test.shape)
```

In [44]: # Converting the Data to PyTorch Tensors

```
X_train_tensor = torch.FloatTensor(X_train_scaled)
y_train_tensor = torch.FloatTensor(y_train_scaled)
X_val_tensor = torch.FloatTensor(X_val_scaled)
y_val_tensor = torch.FloatTensor(y_val_scaled)
X_test_tensor = torch.FloatTensor(X_test_scaled)
y_test_tensor = torch.FloatTensor(y_test_scaled)
```

```

print("Tensor shapes:")
print("X_train_tensor:", X_train_tensor.shape, "y_train_tensor:", y_train_tensor.shape)
print("X_val_tensor:", X_val_tensor.shape, "y_val_tensor:", y_val_tensor.shape)
print("X_test_tensor:", X_test_tensor.shape, "y_test_tensor:", y_test_tensor.shape)

# Creating DataLoader for the Training Data

batch_size = 500
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
train_dataloader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)

print("Number of training batches:", len(train_dataloader))

```

Tensor shapes:
X_train_tensor: torch.Size([1260, 20, 1]) y_train_tensor: torch.Size([1260, 1])
X_val_tensor: torch.Size([300, 20, 1]) y_val_tensor: torch.Size([300, 1])
X_test_tensor: torch.Size([380, 20, 1]) y_test_tensor: torch.Size([380, 1])
Number of training batches: 3

Custom RNN Implementation

In [45]:

```

class CustomRNN(nn.Module):
    def __init__(self, input_size=1, hidden_size=25, output_size=1):
        super(CustomRNN, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size

        # Weights for input-to-hidden and hidden-to-hidden transformations
        self.weight_ih_l0 = nn.Parameter(torch.randn(hidden_size, input_size) * 0.01)
        self.weight_hh_l0 = nn.Parameter(torch.randn(hidden_size, hidden_size) * 0.01)

        # Biases for input and hidden transformations
        self.bias_ih_l0 = nn.Parameter(torch.zeros(hidden_size))
        self.bias_hh_l0 = nn.Parameter(torch.zeros(hidden_size))

        # Weights and bias for the final fully connected layer (hidden-to-output)
        self.Wy = nn.Parameter(torch.randn(output_size, hidden_size) * 0.01)
        self.by = nn.Parameter(torch.zeros(output_size))

    def forward(self, x):
        batch_size, seq_len, _ = x.shape
        h = torch.zeros(batch_size, self.hidden_size) # initialize hidden state

        # Process each time step
        for t in range(seq_len):
            xt = x[:, t, :] # current input at time t
            h = torch.tanh(
                torch.matmul(xt, self.weight_ih_l0.T) + self.bias_ih_l0 +
                torch.matmul(h, self.weight_hh_l0.T) + self.bias_hh_l0
            )

        # Final output projection
        out = torch.matmul(h, self.Wy.T) + self.by
        return out

```

Training RNN Model

```
In [46]: # Defining training function for a given memory (hidden) size
def train_rnn_model(memory_size, num_epochs=50, learning_rate=0.01):
    model = CustomRNN(input_size=1, hidden_size=memory_size, output_size=1)
    criterion = nn.MSELoss()
    optimizer = optim.Adam(model.parameters(), lr=learning_rate)

    for epoch in range(num_epochs):
        model.train()
        epoch_loss = 0.0

        for X_batch, y_batch in train_dataloader:
            optimizer.zero_grad()
            output = model(X_batch)
            loss = criterion(output, y_batch)
            loss.backward()
            optimizer.step()
            epoch_loss += loss.item() * X_batch.size(0)

        epoch_loss /= len(train_dataloader.dataset)
        if epoch % 5 == 0:
            print(f"Memory Size {memory_size} - Epoch {epoch}/{num_epochs} - Loss: {epoch_loss:.4f}")

    # Evaluating on the test set
    model.eval()
    with torch.no_grad():
        test_output = model(X_test_tensor)
        test_loss = criterion(test_output, y_test_tensor)

        # Converting tensors to numpy arrays for metric calculations
        test_output_np = test_output.detach().cpu().numpy().flatten()
        y_test_np = y_test_tensor.detach().cpu().numpy().flatten()

        mse_value = mean_squared_error(y_test_np, test_output_np)
        r2_value = r2_score(y_test_np, test_output_np)

    return model, test_loss.item(), mse_value, r2_value, test_output
```

Train models with memory sizes 2 and 5

```
In [47]: results = {}
for mem_size in [2, 5]:
    print(f"\nTraining model with memory size: {mem_size}")
    model, test_loss, mse_val, r2_val, test_output = train_rnn_model(memory_size=mem_size, num_epochs=50, learning_rate=0.01)
    results[mem_size] = {"model": model, "test_loss": test_loss, "mse": mse_val, "r2": r2_val}
    print(f"Memory Size {mem_size} - Test Loss: {test_loss:.4f}, MSE: {mse_val:.4f}, R2: {r2_val:.4f}")
```

Training model with memory size: 2
Memory Size 2 – Epoch 0/50 – Loss: 0.3670
Memory Size 2 – Epoch 5/50 – Loss: 0.1690
Memory Size 2 – Epoch 10/50 – Loss: 0.1147
Memory Size 2 – Epoch 15/50 – Loss: 0.0867
Memory Size 2 – Epoch 20/50 – Loss: 0.0545
Memory Size 2 – Epoch 25/50 – Loss: 0.0078
Memory Size 2 – Epoch 30/50 – Loss: 0.0042
Memory Size 2 – Epoch 35/50 – Loss: 0.0020
Memory Size 2 – Epoch 40/50 – Loss: 0.0013
Memory Size 2 – Epoch 45/50 – Loss: 0.0011
Memory Size 2 – Test Loss: 0.0012, MSE: 0.0012, R2: 0.9907

Training model with memory size: 5
Memory Size 5 – Epoch 0/50 – Loss: 0.3666
Memory Size 5 – Epoch 5/50 – Loss: 0.1046
Memory Size 5 – Epoch 10/50 – Loss: 0.0859
Memory Size 5 – Epoch 15/50 – Loss: 0.0454
Memory Size 5 – Epoch 20/50 – Loss: 0.0015
Memory Size 5 – Epoch 25/50 – Loss: 0.0010
Memory Size 5 – Epoch 30/50 – Loss: 0.0007
Memory Size 5 – Epoch 35/50 – Loss: 0.0006
Memory Size 5 – Epoch 40/50 – Loss: 0.0005
Memory Size 5 – Epoch 45/50 – Loss: 0.0005
Memory Size 5 – Test Loss: 0.0005, MSE: 0.0005, R2: 0.9961

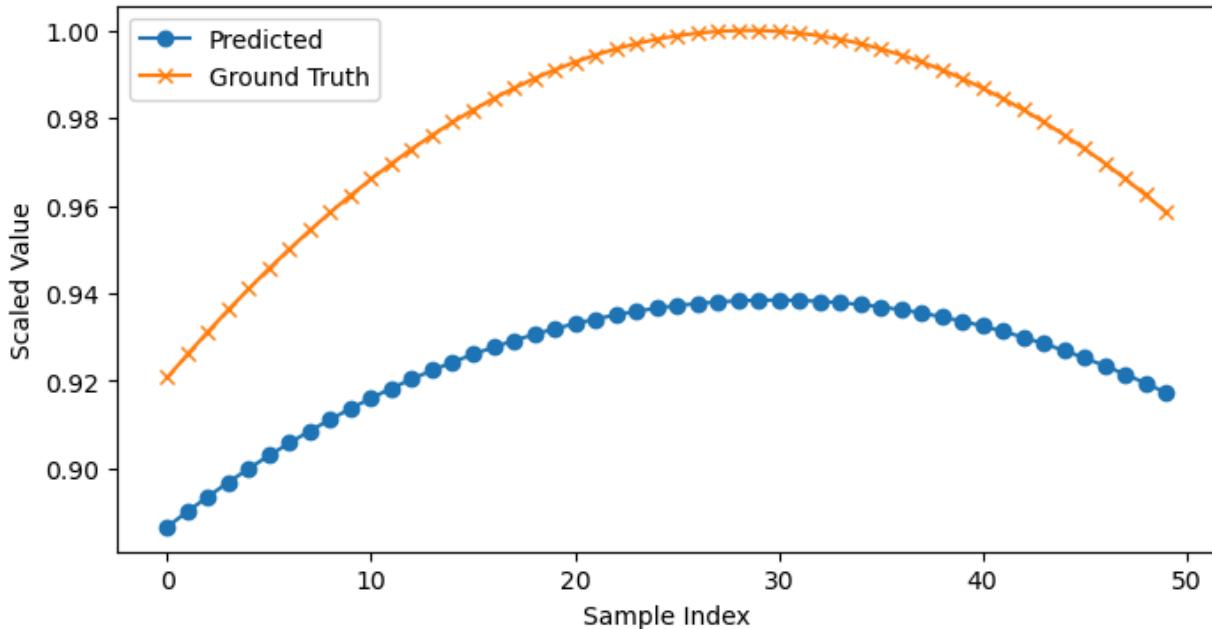
Predictions vs. Ground Truth

```
In [48]: # Visualizing predictions vs. ground truth for some test samples

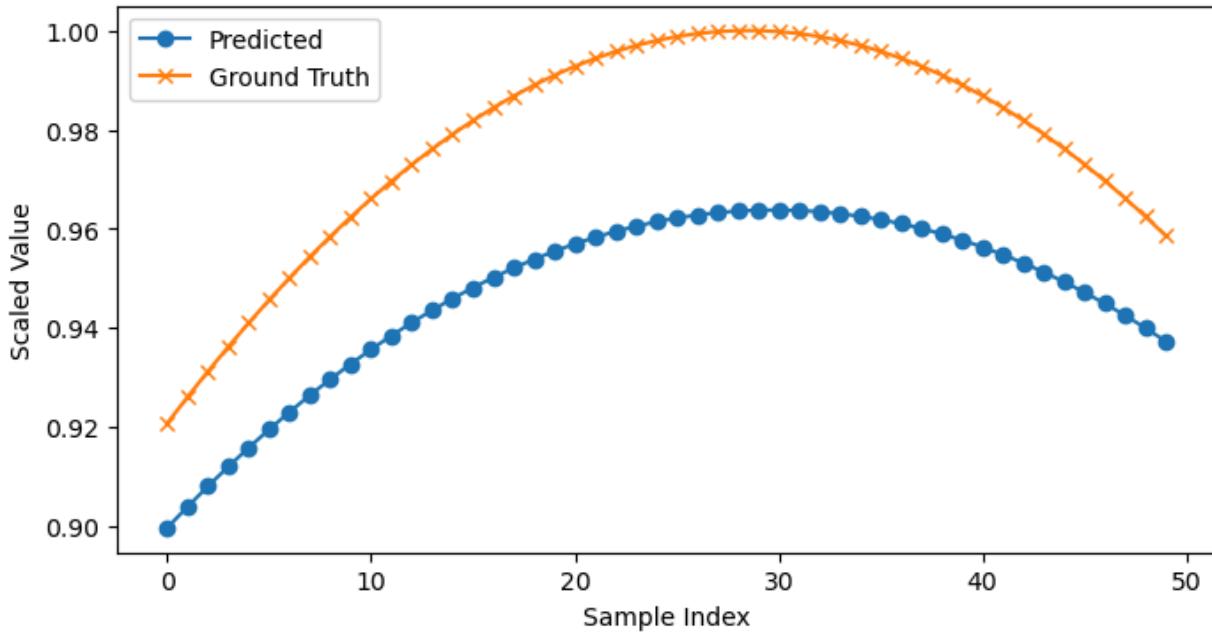
for mem_size in results:
    # Extract the first 50 predictions for visualization
    predictions = results[mem_size]["test_output"][:50].detach().cpu().numpy()
    ground_truth = y_test_tensor[:50].detach().cpu().numpy().flatten()

    plt.figure(figsize=(8, 4))
    plt.plot(predictions, label="Predicted", marker='o')
    plt.plot(ground_truth, label="Ground Truth", marker='x')
    plt.title(f"Memory Size {mem_size}: Predictions vs. Ground Truth")
    plt.xlabel("Sample Index")
    plt.ylabel("Scaled Value")
    plt.legend()
    plt.show()
```

Memory Size 2: Predictions vs. Ground Truth



Memory Size 5: Predictions vs. Ground Truth



- **General Pattern Learned:** Both models (with memory sizes 2 and 5) capture the overall shape of the sine wave, indicating they have learned the primary trend in the data.
- **Underestimation with Smaller Memory:** In the first plot (Memory Size 2), the blue dots (predictions) lie consistently below the orange line (ground truth). This systematic underestimation suggests that a smaller hidden state may not be capturing all the nuances of the signal.
- **Closer Fit with Larger Memory:** In the second plot (Memory Size 5), the model's predictions are still slightly lower than the ground truth but follow the curve more closely, indicating improved performance with a larger hidden state.

- **Overall Conclusion:** Increasing the memory size from 2 to 5 helps the RNN model better approximate the true signal, reducing the bias and improving accuracy, which aligns with the higher R² scores and lower MSE for the Memory Size 5 model.

Performance Metrics

In [49]: # Displaying performance metrics

```
print("\nPerformance Metrics:")
print("Memory Size | Test Loss (MSELoss) | MSE | R2 Score")
print("-----")
for mem_size in results:
    print(f"{mem_size:<11} | {results[mem_size]['test_loss']:<18.4f} | {results[mem_size]['r2_score']:.4f} | {results[mem_size]['mse']:.4f}")
```

Performance Metrics:

Memory Size	Test Loss (MSELoss)	MSE	R2 Score
2	0.0012	0.0012	0.9907
5	0.0005	0.0005	0.9961

The model's performance improves significantly as the memory size increases from 2 to 5. With a memory size of 5, the Test Loss (MSELoss) and MSE decrease to 0.0005, while the R² score improves to 0.9960, indicating that the model is capturing the long-term dependencies more effectively. This suggests that a larger hidden state enables better retention of past information, leading to more accurate predictions of the time series.

Discuss if your model have been successful in capturing long-range dependencies

- **Learning the Overall Trend:**

Both models (with memory sizes 2 and 5) successfully learned the underlying sine-wave pattern. This is evident from the high R² scores and low MSE values, indicating that the models were able to follow the long-term trend in the data.

- **Impact of Memory Size:**

Increasing the hidden state (memory) from 2 to 5 improved performance. The model with a memory size of 5 produced predictions that were closer to the ground truth, suggesting that a larger memory capacity better supports the capture of long-term dependencies.

Was the long-range dependencies of the data complex enough?

- **Simplicity of the Signal:**

The synthetic data in our experiment is based on a single slow sine wave, which represents a relatively straightforward long-term dependency. This simplicity means

that while the RNN demonstrates its ability to learn a long-term pattern, the challenge posed is not as high as it would be with more complex signals.

- **Potential for Increased Complexity:**

In real-world scenarios, time series data often include multiple overlapping long-term trends, abrupt changes, or non-stationary behaviors. Testing the model on such complex data could provide a more rigorous evaluation of its ability to capture intricate long-range dependencies.

Conclusion

The results indicate that our custom RNN can capture long-range dependencies effectively when dealing with a simple, single-component time series. However, the long-term dependency in our data is not very complex. To truly challenge the model's capabilities, future experiments should consider using more complex time series that contain multiple, intertwined long-term patterns.