Worksheet -02

Outputs Screenshots

```python
print("Dataset Preview:")
print(df.head())
print("\nDataset Information:")
print(df.info())
```

```
Dataset Preview:
   label  pixel_0  pixel_1  pixel_2  pixel_3  pixel_4  pixel_5  pixel_6  \
0      5        0        0        0        0        0        0        0
1      0        0        0        0        0        0        0        0
2      4        0        0        0        0        0        0        0
3      1        0        0        0        0        0        0        0
4      9        0        0        0        0        0        0        0

   pixel_7  pixel_8  ...  pixel_774  pixel_775  pixel_776  pixel_777  \
0        0        0  ...          0          0          0          0
1        0        0  ...          0          0          0          0
2        0        0  ...          0          0          0          0
3        0        0  ...          0          0          0          0
4        0        0  ...          0          0          0          0

   pixel_778  pixel_779  pixel_780  pixel_781  pixel_782  pixel_783
0          0          0          0          0          0          0
1          0          0          0          0          0          0
2          0          0          0          0          0          0
3          0          0          0          0          0          0
4          0          0          0          0          0          0

[5 rows x 785 columns]

Dataset Information:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 60000 entries, 0 to 59999
Columns: 785 entries, label to pixel_783
dtypes: int64(785)
memory usage: 359.3 MB
None
```

```python
X = df.iloc[:, 1:-1].values
y = df.iloc[:, -1].values

label_encoder = LabelEncoder()
y_encoded = label_encoder.fit_transform(y)

one_hot_encoder = OneHotEncoder(sparse_output=False)
y_one_hot = one_hot_encoder.fit_transform(y_encoded.reshape(-1, 1))

print("\nUnique Classes:", np.unique(y))
print("Encoded Labels:", np.unique(y_encoded))
print("One-Hot Encoded Labels:\n", y_one_hot[:5])
```

```
Unique Classes: [0]
Encoded Labels: [0]
One-Hot Encoded Labels:
 [[1.]
 [1.]
 [1.]
 [1.]
 [1.]]
```

```
[ ] X_train, X_test, y_train, y_test = train_test_split(X, y_one_hot, test_size=0.2, random_state=

    print("\nShapes:")
    print("X_train:", X_train.shape, "y_train:", y_train.shape)
    print("X_test:", X_test.shape, "y_test:", y_test.shape)
```

```
Shapes:
X_train: (48000, 783) y_train: (48000, 1)
X_test: (12000, 783) y_test: (12000, 1)
```

Softmax Function:

```
import numpy as np

def softmax(z):
    """
    Compute the softmax probabilities for a given input matrix.

    Parameters:
    z (numpy.ndarray): Logits (raw scores) of shape (m, n), where
                       - m is the number of samples.
                       - n is the number of classes.

    Returns:
    numpy.ndarray: Softmax probability matrix of shape (m, n), where
                   each row sums to 1 and represents the probability
                   distribution over classes.

    Notes:
    - The input to softmax is typically computed as: z = XW + b.
    - Uses numerical stabilization by subtracting the max value per row.
    """
    z_shifted = z - np.max(z, axis=1, keepdims=True)
    exp_z = np.exp(z_shifted)
    return exp_z / np.sum(exp_z, axis=1, keepdims=True)
```

Softmax Test Case:

This test case checks that each row in the resulting softmax probabilities sums to 1, which is the fundamental property of softmax.

```
[ ] z_test = np.array([[2.0, 1.0, 0.1], [1.0, 1.0, 1.0]])
    softmax_output = softmax(z_test)

    row_sums = np.sum(softmax_output, axis=1)

    assert np.allclose(row_sums, 1), f"Test failed: Row sums are {row_sums}"

    print("Softmax function passed the test case!")
```

```
Softmax function passed the test case!
```

Prediction Function:

```
[ ] def predict_softmax(X, W, b):
        """
        Predict the class labels for a set of samples using the trained softmax model.

        Parameters:
        X (numpy.ndarray): Feature matrix of shape (n, d), where n is the number of samples and d
        W (numpy.ndarray): Weight matrix of shape (d, c), where c is the number of classes.
        b (numpy.ndarray): Bias vector of shape (c,).

        Returns:
        numpy.ndarray: Predicted class labels of shape (n,), where each value is the index of the
        """
        z = np.dot(X, W) + b
        y_pred = softmax(z)

        predicted_classes = np.argmax(y_pred, axis=1)

        return predicted_classes
```

Test Function for Prediction Function:

The test function ensures that the predicted class labels have the same number of elements as the input samples, verifying that the model produces a valid output shape.

```
X_test = np.array([[0.2, 0.8], [0.5, 0.5], [0.9, 0.1]])
W_test = np.array([[0.4, 0.2, 0.1], [0.3, 0.7, 0.5]])
b_test = np.array([0.1, 0.2, 0.3])

y_pred_test = predict_softmax(X_test, W_test, b_test)

assert y_pred_test.shape == (3,), f"Test failed: Expected shape (3,), got {y_pred_test.shape}"

print("Predicted class labels:", y_pred_test)
```

```
Predicted class labels: [1 1 0]
```

Loss Function:

```
[ ] def loss_softmax(y_pred, y):
        """
        Compute the cross-entropy loss for a single sample.

        Parameters:
        y_pred (numpy.ndarray): Predicted probabilities of shape (c,) for a single sample,
                                where c is the number of classes.
        y (numpy.ndarray): True labels (one-hot encoded) of shape (c,), where c is the number of c

        Returns:
        float: Cross-entropy loss for the given sample.
        """

        epsilon = 1e-12
        y_pred = np.clip(y_pred, epsilon, 1.0 - epsilon)
        n = y.shape[0]
        loss = -np.sum(y * np.log(y_pred)) / n
        return loss
```

Test case for Loss Function:

This test case Compares loss for correct vs. incorrect predictions.

- Expects low loss for correct predictions.
- Expects high loss for incorrect predictions.

```
import numpy as np

y_true_correct = np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]])
y_pred_correct = np.array([[0.9, 0.05, 0.05],
                           [0.1, 0.85, 0.05],
                           [0.05, 0.1, 0.85]])

y_pred_incorrect = np.array([[0.05, 0.05, 0.9],
                             [0.1, 0.05, 0.85],
                             [0.85, 0.1, 0.05]])

loss_correct = loss_softmax(y_pred_correct, y_true_correct)
loss_incorrect = loss_softmax(y_pred_incorrect, y_true_correct)

assert loss_correct < loss_incorrect, f"Test failed: Expected loss_correct < loss_incorrect, b

print(f"Cross-Entropy Loss (Correct Predictions): {loss_correct:.4f}")
print(f"Cross-Entropy Loss (Incorrect Predictions): {loss_incorrect:.4f}")
```

```
Cross-Entropy Loss (Correct Predictions): 0.1435
Cross-Entropy Loss (Incorrect Predictions): 2.9957
```

Cost Function:

```
[ ] def cost_softmax(X, y, W, b):
        """
        Compute the average softmax regression cost (cross-entropy loss) over all samples.

        Parameters:
        X (numpy.ndarray): Feature matrix of shape (n, d), where n is the number of samples and d
        y (numpy.ndarray): True labels (one-hot encoded) of shape (n, c), where n is the number of
        W (numpy.ndarray): Weight matrix of shape (d, c).
        b (numpy.ndarray): Bias vector of shape (c,).

        Returns:
        float: Average softmax cost (cross-entropy loss) over all samples.
        """

        n = X.shape[0]
        z = np.dot(X, W) + b
        y_pred = softmax(z)
        cost = loss_softmax(y_pred, y)
        return cost
```

Test Case for Cost Function:

The test case assures that the cost for the incorrect prediction should be higher than for the correct prediction, confirming that the cost function behaves as expected.

Test Case for Cost Function:

The test case assures that the cost for the incorrect prediction should be higher than for the correct prediction, confirming that the cost function behaves as expected.

```
import numpy as np

X_correct = np.array([[1.0, 0.0], [0.0, 1.0]])
y_correct = np.array([[1, 0], [0, 1]])
W_correct = np.array([[5.0, -2.0], [-3.0, 5.0]])
b_correct = np.array([0.1, 0.1])

X_incorrect = np.array([[0.1, 0.9], [0.8, 0.2]])
y_incorrect = np.array([[1, 0], [0, 1]])
W_incorrect = np.array([[0.1, 2.0], [1.5, 0.3]])
b_incorrect = np.array([0.5, 0.6])

cost_correct = cost_softmax(X_correct, y_correct, W_correct, b_correct)

cost_incorrect = cost_softmax(X_incorrect, y_incorrect, W_incorrect, b_incorrect)

assert cost_incorrect > cost_correct, f"Test failed: Incorrect cost {cost_incorrect} is not gr

print("Cost for correct prediction:", cost_correct)
print("Cost for incorrect prediction:", cost_incorrect)

print("Test passed!")
```

```
Cost for correct prediction: 0.0006234364133349324
Cost for incorrect prediction: 0.29930861359446115
Test passed!
```

Computing Gradients:

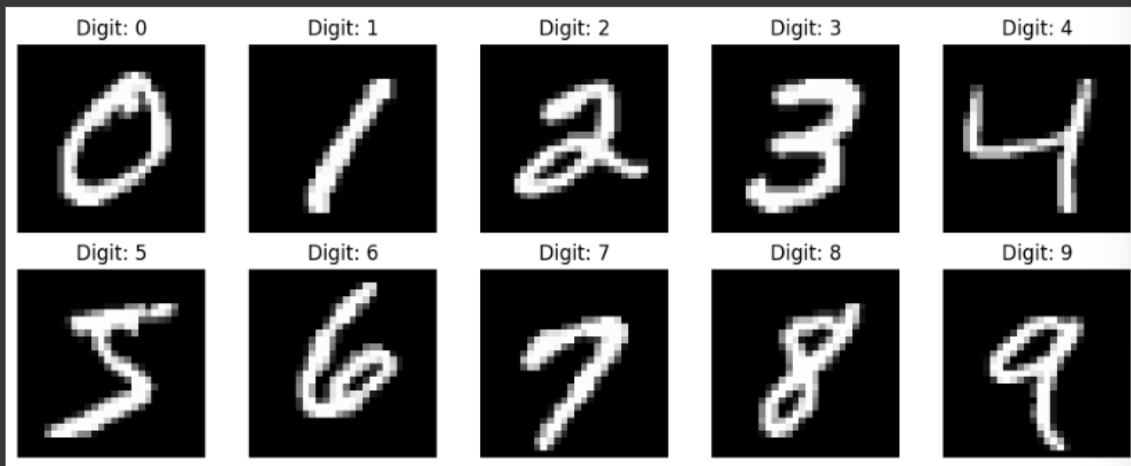
```
def compute_gradient_softmax(X, y, W, b):
    """
    Compute the gradients of the cost function with respect to weights and biases.

    Parameters:
    X (numpy.ndarray): Feature matrix of shape (n, d).
    y (numpy.ndarray): True labels (one-hot encoded) of shape (n, c).
    W (numpy.ndarray): Weight matrix of shape (d, c).
    b (numpy.ndarray): Bias vector of shape (c,).

    Returns:
    tuple: Gradients with respect to weights (d, c) and biases (c,).
    """


    n, d = X.shape
    z = np.dot(X, W) + b
    y_pred = softmax(z)

    grad_W = np.dot(X.T, (y_pred - y)) / n
    grad_b = np.sum(y_pred - y, axis=0) / n

    return grad_W, grad_b
```

The test checks if the gradients from the function are close enough to the manually computed gradients using np.allclose, which accounts for potential floating-point discrepancies.

```
[ ] import numpy as np

    X_test = np.array([[0.2, 0.8], [0.5, 0.5], [0.9, 0.1]])
    y_test = np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]])

    W_test = np.array([[0.4, 0.2, 0.1], [0.3, 0.7, 0.5]])
    b_test = np.array([0.1, 0.2, 0.3])

    grad_W, grad_b = compute_gradient_softmax(X_test, y_test, W_test, b_test)

    z_test = np.dot(X_test, W_test) + b_test
    y_pred_test = softmax(z_test)

    grad_W_manual = np.dot(X_test.T, (y_pred_test - y_test)) / X_test.shape[0]
    grad_b_manual = np.sum(y_pred_test - y_test, axis=0) / X_test.shape[0]

    assert np.allclose(grad_W, grad_W_manual), f"Test failed: Gradients w.r.t. W are not equal.\nE
    assert np.allclose(grad_b, grad_b_manual), f"Test failed: Gradients w.r.t. b are not equal.\nE

    print("Gradient w.r.t. W:", grad_W)
    print("Gradient w.r.t. b:", grad_b)

    print("Test passed!")
```

```
Gradient w.r.t. W: [[ 0.1031051   0.01805685 -0.12116196]
 [-0.13600547  0.00679023  0.12921524]]
Gradient w.r.t. b: [-0.03290036  0.02484708  0.00805328]
Test passed!
```

```python
csv_file_path = "/content/drive/MyDrive/Sem6/AI and ML Workshop/Week-2/mnist_dataset.csv"
X_train, X_test, y_train, y_test = load_and_prepare_mnist(csv_file_path)
```



A Quick debugging Step:

```python
assert len(X_train) == len(y_train), f"Error: X and y have different lengths! X={len(X_train)}
print("Move forward: Dimension of Feture Matrix X and label vector y matched.")
```

```
Move forward: Dimension of Feture Matrix X and label vector y matched.
```

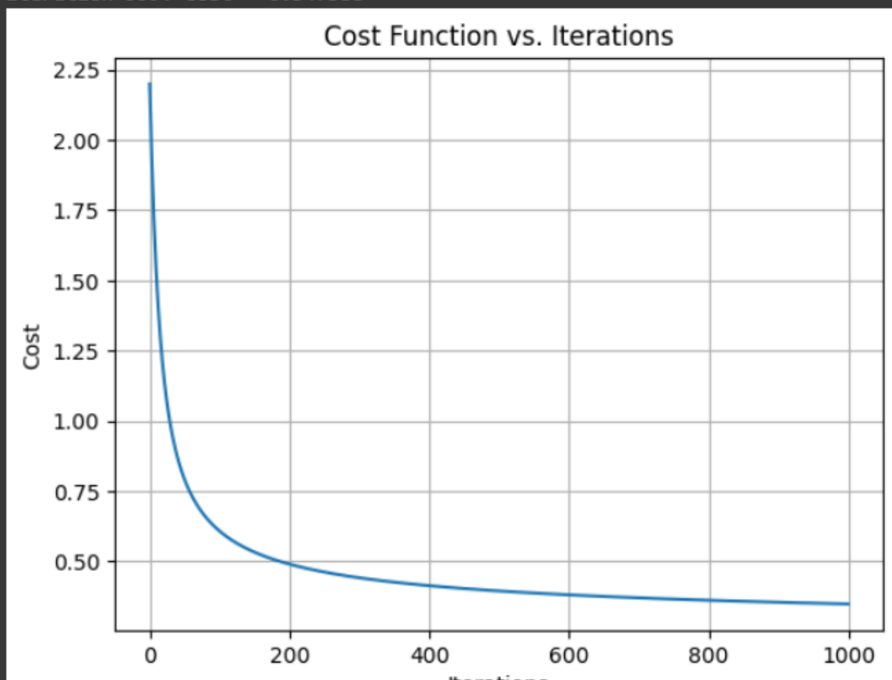
Train the Model:

```
print(f"Training data shape: {X_train.shape}")
print(f"Test data shape: {X_test.shape}")
```

```
Training data shape: (48000, 784)
Test data shape: (12000, 784)
```

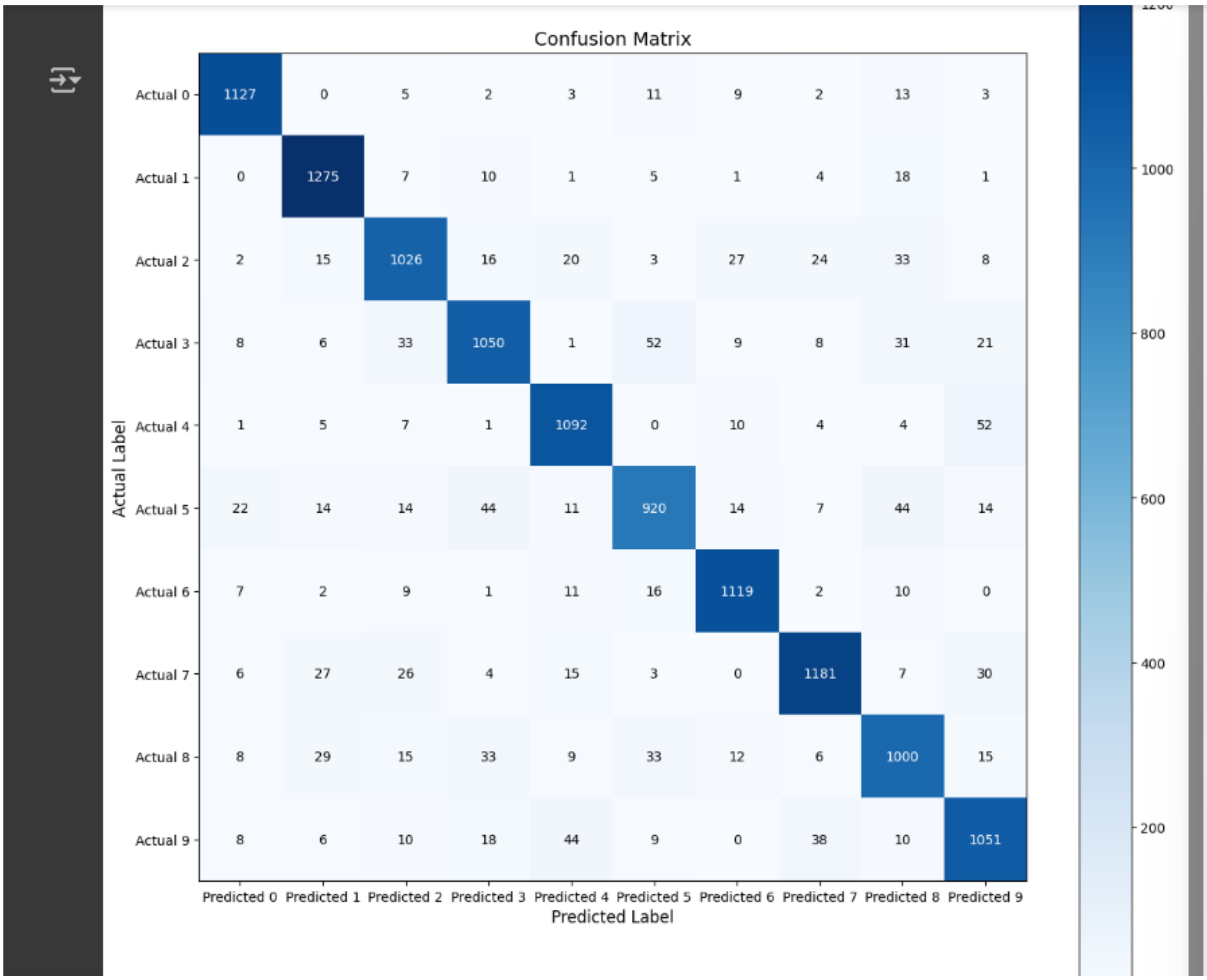
```
from sklearn.preprocessing import OneHotEncoder

if len(y_train.shape) == 1:
    encoder = OneHotEncoder(sparse_output=False)
    y_train = encoder.fit_transform(y_train.reshape(-1, 1))
    y_test = encoder.transform(y_test.reshape(-1, 1))

d = X_train.shape[1]
c = y_train.shape[1]

W = np.random.randn(d, c) * 0.01
b = np.zeros(c)

alpha = 0.1
n_iter = 1000

W_opt, b_opt, cost_history = gradient_descent_softmax(X_train, y_train, W, b, alpha, n_iter, s

plt.plot(cost_history)
plt.title('Cost Function vs. Iterations')
plt.xlabel('Iterations')
plt.ylabel('Cost')
plt.grid(True)
```

```
Iteration 0: Cost = 2.199553
Iteration 100: Cost = 0.607084
Iteration 200: Cost = 0.489327
Iteration 300: Cost = 0.440739
Iteration 400: Cost = 0.412682
Iteration 500: Cost = 0.393835
Iteration 600: Cost = 0.380030
Iteration 700: Cost = 0.369336
Iteration 800: Cost = 0.360723
Iteration 900: Cost = 0.353583
Iteration 999: Cost = 0.347588
```



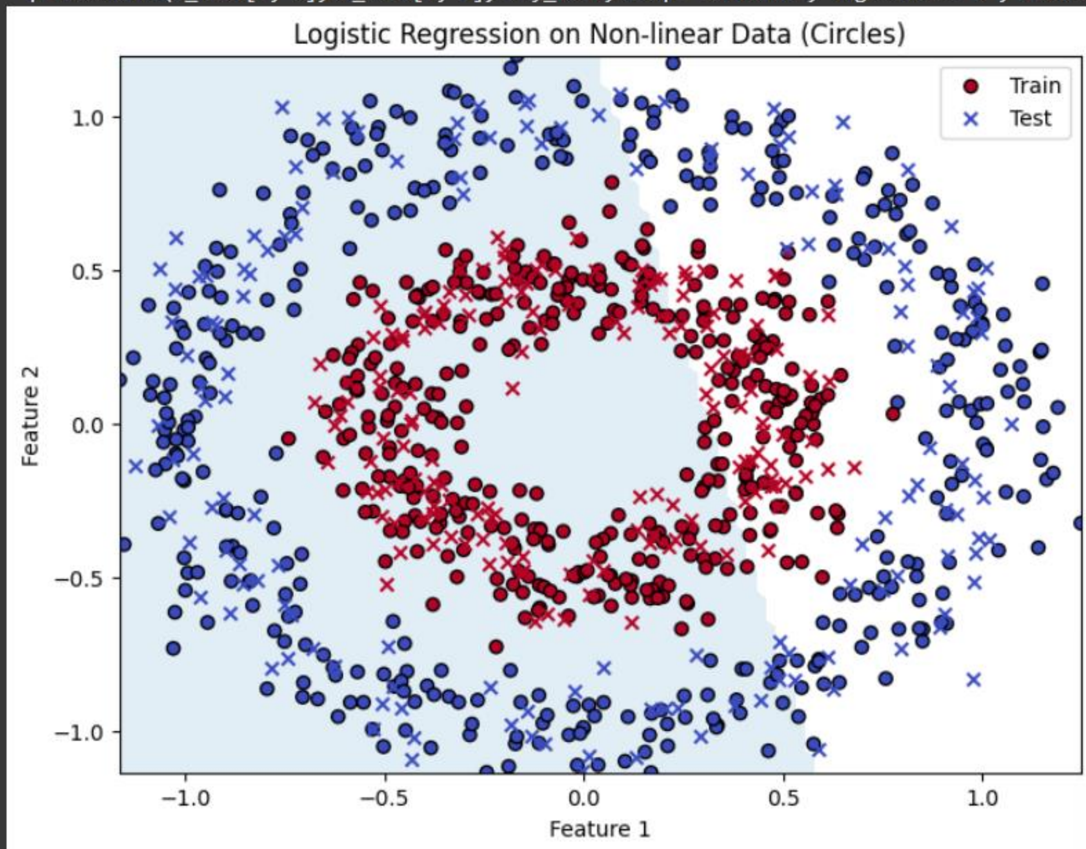
```
[ ] ax.set_xticks(range(num_classes))
    ax.set_yticks(range(num_classes))
    ax.set_xticklabels([f'Predicted {i}' for i in range(num_classes)])
    ax.set_yticklabels([f'Actual {i}' for i in range(num_classes)])

    for i in range(cm.shape[0]):
        for j in range(cm.shape[1]):
            ax.text(j, i, cm[i, j], ha='center', va='center', color='white' if cm[i, j] > np.max(

    ax.grid(False)
    plt.title('Confusion Matrix', fontsize=14)
    plt.xlabel('Predicted Label', fontsize=12)
    plt.ylabel('Actual Label', fontsize=12)

    plt.tight_layout()
    plt.colorbar(cax)
    plt.show()
```

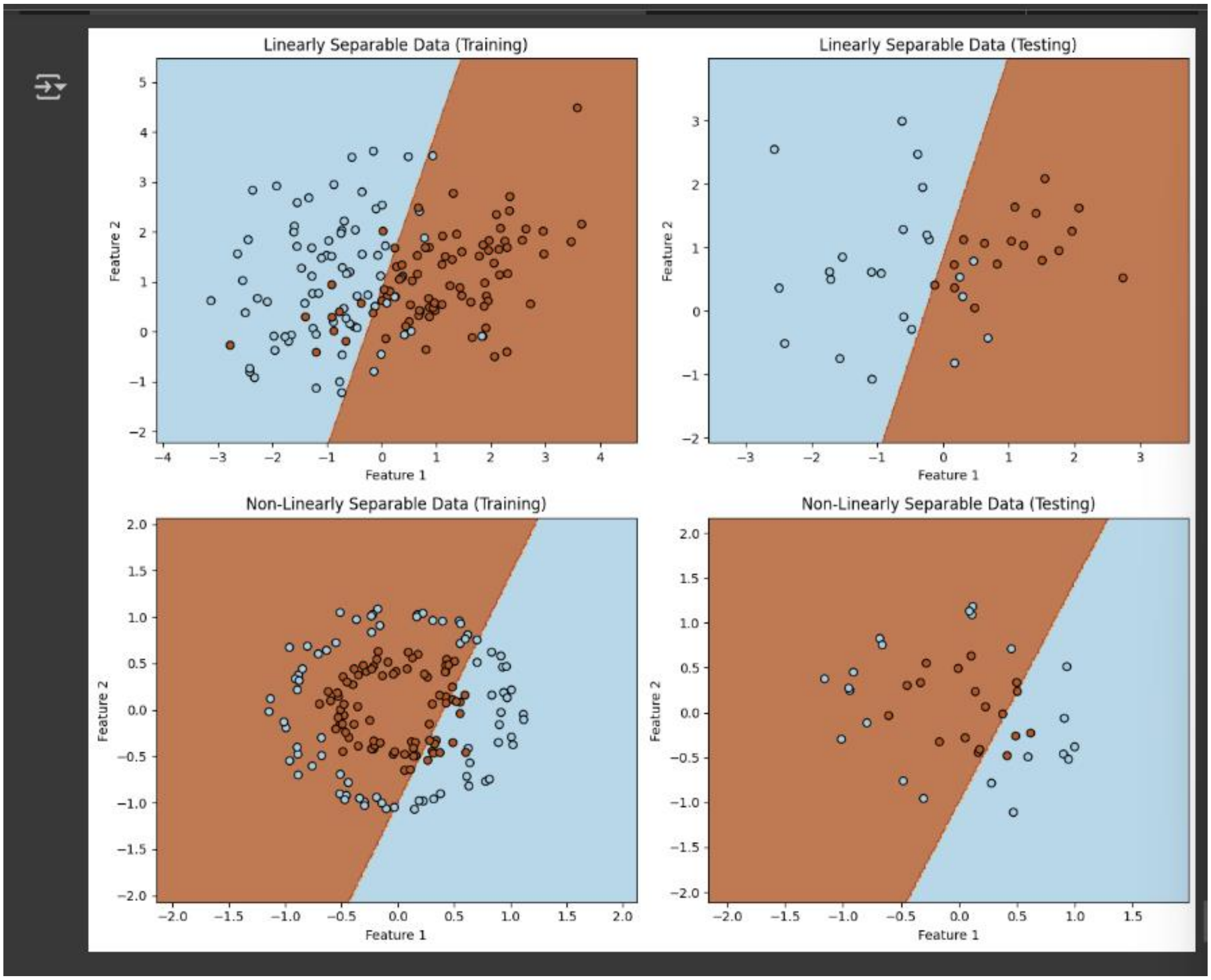
```
Confusion Matrix:
[[1127    0    5    2    3   11    9    2   13    3]
 [   0 1275    7   10    1    5    1    4   18    1]
 [   2   15 1026   16   20    3   27   24   33    8]
 [   8    6   33 1050    1   52    9    8   31   21]
 [   1    5    7    1 1092    0   10    4    4   52]
 [  22   14   14   44   11  920   14    7   44   14]
 [   7    2    9    1   11   16 1119    2   10    0]
 [   6   27   26    4   15    3    0 1181    7   30]
 [   8   29   15   33    9   33   12    6 1000   15]
 [   8    6   10   18   44    9    0   38   10 1051]]
Precision: 0.90
Recall: 0.90
F1-Score: 0.90
```



Confusion Matrix
Actual Label
Predicted Label
Actual 0
Actual 1
Actual 2
Actual 3
Actual 4
Actual 5
Actual 6
Actual 7
Actual 8
Actual 9
Predicted 0
Predicted 1
Predicted 2
Predicted 3
Predicted 4
Predicted 5
Predicted 6
Predicted 7
Predicted 8
Predicted 9
1127 0 5 2 3 11 9 2 13 3
0 1275 7 10 1 5 1 4 18 1
2 15 1026 16 20 3 27 24 33 8
8 6 33 1050 1 52 9 8 31 21
1 5 7 1 1092 0 10 4 4 52
22 14 14 44 11 920 14 7 44 14
7 2 9 1 11 16 1119 2 10 0
6 27 26 4 15 3 0 1181 7 30
8 29 15 33 9 33 12 6 1000 15
8 6 10 18 44 9 0 38 10 1051
1000
800
600
400
200

```
<ipython-input-24-6a5a1fca6a1b>:19: UserWarning: You passed a edgecolor/edgecolors ('k') for an
  plt.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap='coolwarm', edgecolors='k', marker='x'
```





Linearly Separable Data (Training)
Linearly Separable Data (Testing)
Non-Linearly Separable Data (Training)
Non-Linearly Separable Data (Testing)
Feature 1
Feature 2