ECE 174 Mini Project 1 Report

Armon Barakchi

November 17th, 2024

Table of Contents

Problem 1: Binary Classifiers 3
One-Versus-One Binary Classifier 3
One-Versus-All Binary Classifier 4
Multi-Class Classifiers 4
.... 9
Problem 1 Takeaways 11

Problem 2: Randomized Feature Based Least Square Classifiers 12
Intuition 12
Implementation for Feature Mapping 12
One Versus One with Randomized Features 14
One Versus All with Randomized Features 23
Problem 2 Takeaways 33

Appendix A 35

Appendix B 37

Appendix C 39

Please Note that the code provided in this report is not the final formatted code, but rather the code used to get each result. The finalized commented code is in the coding portion of the submission.

Problem 1: Binary Classifiers

One-Versus-One Binary Classifier

Intuition:

The least squares problem for the one-versus-one binary classifier is framed as:

$$\min_{\beta,\alpha} \sum_{i=1}^{N} \left(y_i - \beta^T x_i - \alpha\right)^2$$

Where the variables are defined as follows:

- $y_i \in \{-1,1\}$ is the correct output for our classifier for a given x_i
- $x_i \in R^{784}$ is an input "feature vector" where each entry represents the pixel intensity from a 28x28 MNIST hand-written digit image. These values have been normalized to reside between 0 and 1

Define:

$$y = [y_1, y_2, \ldots, y_k]^T$$
$$X = [x_1^T, x_2^T, \ldots, x_k^T]^T, \; x_i \in R^{784}$$
$$\theta = [\beta, \alpha]^T$$
$$\hat{X} = [\, X, 1]$$

Where k is the number of data points (x_i, y_i) in our training dataset, which is 60000 for the MNIST dataset. Now the least squares problem can be redefined as:

$$\min_{\theta} \|y - \hat{X}\theta\|_2^2$$

The theta in the equation above can be used for one binary classifier with solution given by:

$$\hat{X}^T X\theta = \hat{X}^T y \Rightarrow \theta = (\hat{X}^T X)^{-1}\hat{X}^T y$$

As shown in class, this equation always has a solution. Using this equation, and given a y and an X, we can construct an $\hat{X}$ and solve for θ. This would give us β (the weights for the binary classifier) and α (the intercept term). Once θ is found, the one-versus-one binary classifier is given by:

$$\hat{f}_{i,j}(x) = sign(\beta_{i,j}^T x + \alpha) = \begin{Bmatrix} 1 \; if \; label = i \\ -1 \; if \; label = j \end{Bmatrix}$$

The sign function used is from the NumPy library. Additionally, there will $\frac{k(k-1)}{2} = 45$ unique classifiers, but 90 different combinations of comparisons when order is considered. To account for this, we will only output a classifier when $i < j$.

One-Versus-All Binary Classifier

Intuition:

We extend the binary classifier to K classes, denoted by **num_classes** in the code, where the classifier is a function $\hat{f}: R^n \to \{1, \dots, K\}$ such that:

$$\hat{f}(x) = \max_{k=1,\dots,K} g_k(x)$$

Where $g_k(x) = \beta_k^T x + \alpha_k$ is the least squares model for the binary classifier with label k against all other labels, where a 1 is outputted if the label is k and -1 otherwise. The implementation of each binary classifier is analogous to the One-Versus-One, but the training data is altered such that all data points with label k are given 1 and -1 otherwise. The testing data is filtered similarly.

One-versus-all classifier training error and confusion matrix:

Multi-Class Classifiers

These classifiers will use the various binary classifiers (one-versus-one and one-versus-all) as building blocks to identify each of the 10 digits in the MNIST data apart from the others.

One-versus-all multi-class classifier:

This classifier runs the one-versus-all binary classifier for each of the 10 digits and returns the predicted label that had the highest confidence i.e. $\max g_k(x)$

Implementation and Results:

Please note that this implementation is the high-level script, and it calls helper functions from helpersOneVsAll.py. The contents of that file can be found in Appendix A of this document.

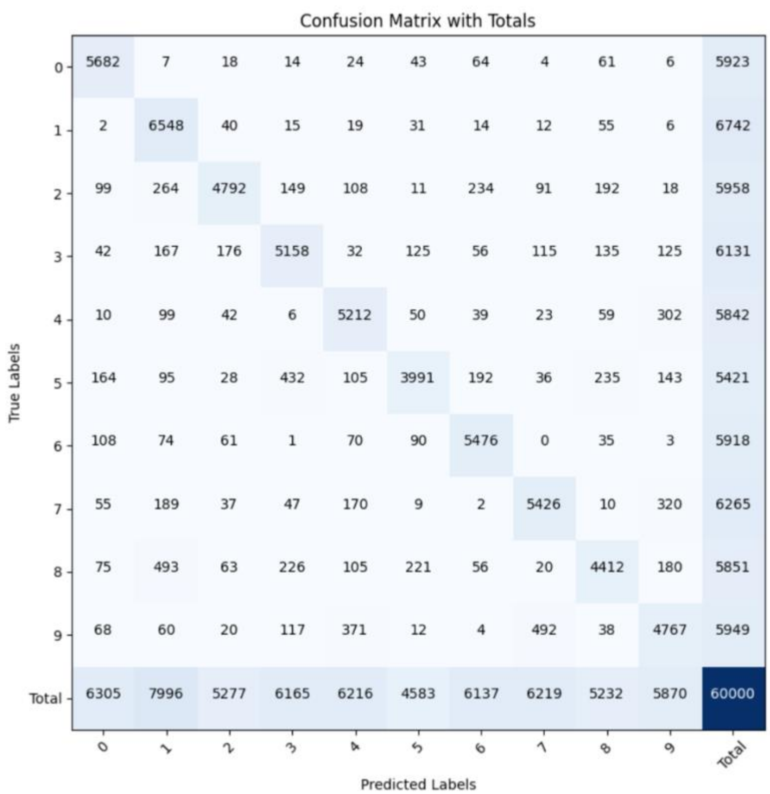
```
import helpersOneVsAll as helpers
import numpy as np
from scipy.io import loadmat
import genHelpers


#import data
data = loadmat('mnist.mat')
images = data['trainX']  # 60000 images -> each an array of 784 pixels
test_images = data['testX']
training_labels = data['trainY']  # 60000 labels
test_labels = data['testY']

# Convert to float32 and normalize; initialize variaibles
images = images.astype(np.float32) / 255.0
test_images = test_images.astype(np.float32) / 255.0
num_classes = 10  # For MNIST, this would be 10 for digits 0-9
betas = []
alphas = []

#training classifiers
for k in range(num_classes):
    # Label data for class k vs all others y(i) in {-1,1}
    binary_labels = helpers.labelBinaryData(k, training_labels)

    # Train a binary classifier for this class
    beta, alpha = helpers.train_binary_classifier(images, binary_labels)

    # Store the classifier's parameters
    betas.append(beta)
    alphas.append(alpha)

#test paramaters on training data
predicted = helpers.predict_one_vs_all_full(images, betas, alphas)
error = genHelpers.classwise_error_rate(predicted, training_labels) #error by
class
total_error = genHelpers.error_rate(predicted, training_labels) #total error
print("The total error for the training data was: {}".format(total_error))
for num, error in error.items():
    print("Error rate for class {}: {}".format(num, error))

#test paramaters on test data
predicted = helpers.predict_one_vs_all_full(test_images, betas, alphas)
error = genHelpers.classwise_error_rate(predicted, test_labels) #error by
class
total_error = genHelpers.error_rate(predicted, test_labels) #total error
print("The total error for the training data was: {}".format(total_error))
for num, error in error.items():
    print("Error rate for class {}: {}".format(num, error))
```

This program outputs the below:

The total error for the training data was: 0.14226666666666668
Error rate for class 0: 4.07%
Error rate for class 1: 2.88%
Error rate for class 2: 19.57%
Error rate for class 3: 15.87%
Error rate for class 4: 10.78%
Error rate for class 5: 26.38%
Error rate for class 6: 7.47%
Error rate for class 7: 13.39%
Error rate for class 8: 24.59%
Error rate for class 9: 19.87%

The total error for the training data was: 0.1397
Error rate for class 0: 3.67%
Error rate for class 1: 2.47%
Error rate for class 2: 21.22%
Error rate for class 3: 12.87%
Error rate for class 4: 10.29%
Error rate for class 5: 26.12%
Error rate for class 6: 8.66%
Error rate for class 7: 14.01%
Error rate for class 8: 22.07%
Error rate for class 9: 20.61%

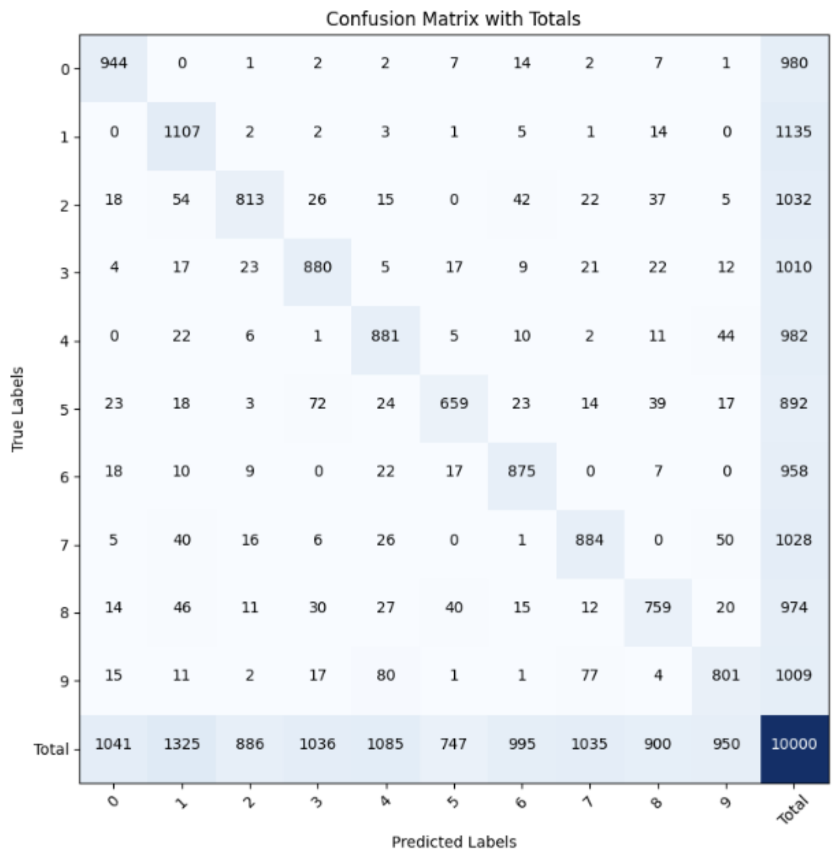
Below is the implementation and result for the confusion matrix for this multi-class classifier:



Confusion Matrix with Totals

True Labels \ Predicted Labels	0	1	2	3	4	5	6	7	8	9	Total
0	5682	7	18	14	24	43	64	4	61	6	5923
1	2	6548	40	15	19	31	14	12	55	6	6742
2	99	264	4792	149	108	11	234	91	192	18	5958
3	42	167	176	5158	32	125	56	115	135	125	6131
4	10	99	42	6	5212	50	39	23	59	302	5842
5	164	95	28	432	105	3991	192	36	235	143	5421
6	108	74	61	1	70	90	5476	0	35	3	5918
7	55	189	37	47	170	9	2	5426	10	320	6265
8	75	493	63	226	105	221	56	20	4412	180	5851
9	68	60	20	117	371	12	4	492	38	4767	5949
Total	6305	7996	5277	6165	6216	4583	6137	6219	5232	5870	60000

This is the confusion matrix for the training data



Confusion matrix for test data

See the one-versus-one classifier confusion matrices for the implementation of the confusion matrix function.

One-versus-one multi-class classifier:

The one-versus-one multi-class classifier consists of all 45 one-versus-one binary classifiers. For each pair, it assigns a "vote" to one of the digits 0-9 based on the output of the binary classifier. After collecting all the votes, it selects the digit with the highest tally.

Implementation and Results:

Please note that this implementation is the high-level script, and it calls helper functions from helpersOneVsOne.py. The contents of that file can be found in Appendix B of this document.

```
import helpersOneVsOne as helpers
import numpy as np
from scipy.io import loadmat
import genHelpers
```

```
import matplotlib.pyplot as plt

#import data
data = loadmat('mnist.mat')
images = data['trainX']  # 60000 images -> each an array of 784 pixels
test_images = data['testX']
training_labels = data['trainY']  # 60000 labels
test_labels = data['testY']

# Convert to float32 and normalize
images = images.astype(np.float32) / 255.0
test_images = test_images.astype(np.float32) / 255.0
num_classes = 10  # For MNIST, this would be 10 for digits 0-9

#get theta = [beta, alpha] for the 45 relevant binary classifiers where i<j
binary_classifiers = helpers.train_ovo_classifiers(images, training_labels,
num_classes=10)

#put the training images through the binary classifiers and get predicted
labels
predicted_labels = helpers.predict_ovo(images, binary_classifiers,
num_classes=10)
error = genHelpers.classwise_error_rate(predicted_labels, training_labels)
#classwise error
total_error = genHelpers.error_rate(predicted_labels, training_labels) #total
error on training data
print("The total error for the training data was: {}".format(total_error))
for num, error in error.items():
    print("Error rate for class {}: {}".format(num, error))


#put the testing images through the binary classifiers and get predicted
labels
predicted_labels = helpers.predict_ovo(test_images, binary_classifiers,
num_classes=10)
error = genHelpers.classwise_error_rate(predicted_labels, test_labels)
#classwise error
total_error = genHelpers.error_rate(predicted_labels, test_labels) #total
error on testing data
print("The total error for the test data was: {}".format(total_error))
for num, error in error.items():
    print("Error rate for class {}: {}".format(num, error))
```

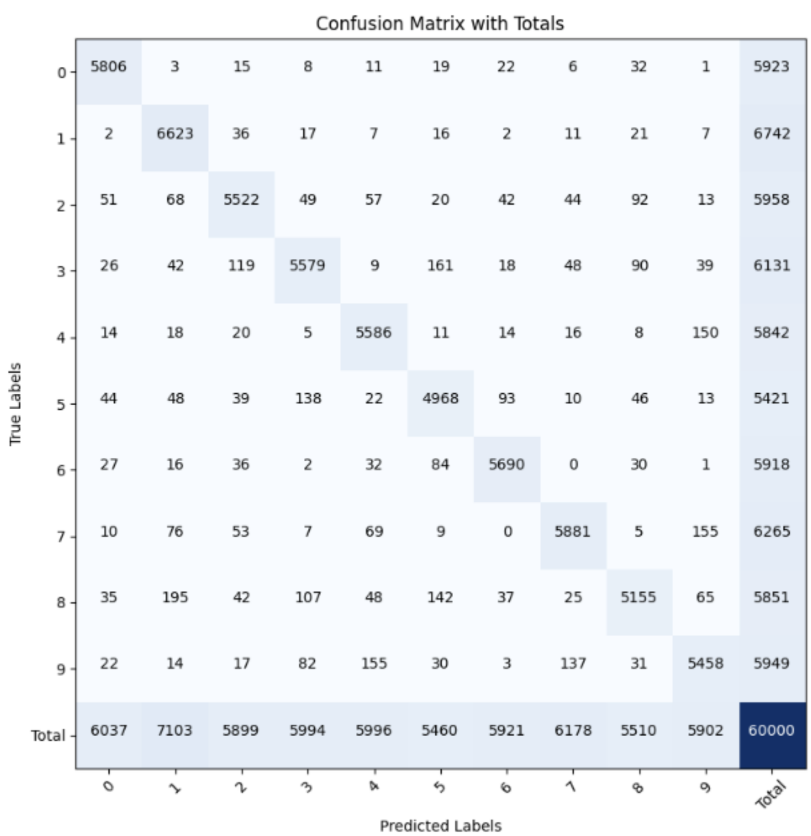
This program outputs the below:

The total error for the test data was: 0.0703
Error rate for class 0: 1.94%
Error rate for class 1: 1.32%
Error rate for class 2: 9.30%
Error rate for class 3: 8.32%
Error rate for class 4: 5.19%
Error rate for class 5: 10.31%
Error rate for class 6: 5.22%
Error rate for class 7: 7.10%
Error rate for class 8: 13.76%
Error rate for class 9: 8.82%

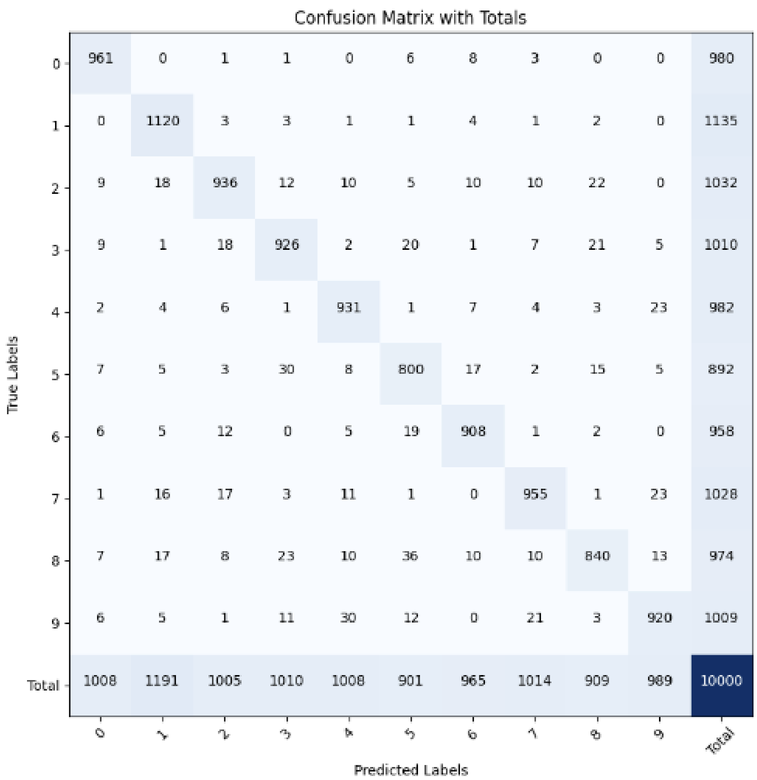
The total error for the training data was: 0.0622
Error rate for class 0: 1.98%
Error rate for class 1: 1.77%
Error rate for class 2: 7.32%
Error rate for class 3: 9.00%
Error rate for class 4: 4.38%
Error rate for class 5: 8.36%
Error rate for class 6: 3.85%
Error rate for class 7: 6.13%
Error rate for class 8: 11.90%
Error rate for class 9: 8.25%

Where the number class corresponds to the number handwritten digit in the MNIST dataset.

Below is the implementation and result for the confusion matrix for this multi-class classifier:



Confusion matrix for testing data



Confusion matrix for test data

Implementation:

```
def plot_confusion_matrix(true_labels, predicted_labels, num_classes):

    # Compute the confusion matrix
    cm = confusion_matrix(true_labels, predicted_labels,
labels=np.arange(num_classes))

    # Compute row and column totals
    row_totals = cm.sum(axis=1)
    col_totals = cm.sum(axis=0)
    overall_total = cm.sum()

    # Extend the confusion matrix to include totals
    cm_with_totals = np.zeros((num_classes + 1, num_classes + 1), dtype=int)
    cm_with_totals[:num_classes, :num_classes] = cm
    cm_with_totals[:num_classes, -1] = row_totals  # Add row totals
    cm_with_totals[-1, :num_classes] = col_totals  # Add column totals
    cm_with_totals[-1, -1] = overall_total  # Add overall total

    # Plot the extended confusion matrix as a heatmap
    plt.figure(figsize=(10, 8))
    plt.imshow(cm_with_totals, interpolation='nearest', cmap=plt.cm.Blues)
    plt.title("Confusion Matrix with Totals")
    plt.colorbar()

    # Label the axes
    tick_marks = np.arange(num_classes + 1)
```

```
    labels = [str(i) for i in range(num_classes)] + ['Total']
    plt.xticks(tick_marks, labels, rotation=45)
    plt.yticks(tick_marks, labels)

    # Add the numerical values to the confusion matrix cells
    for i in range(num_classes + 1):
        for j in range(num_classes + 1):
            plt.text(j, i, f"{cm_with_totals[i, j]}",
                         horizontalalignment="center",
                         color="white" if cm_with_totals[i, j] >
cm_with_totals.max() / 2 else "black")

    # Add axis labels
    plt.ylabel("True Labels")
    plt.xlabel("Predicted Labels")
    plt.tight_layout()
    plt.show()
```

Problem 1 Takeaways

It appears that for the training and test data, the one-vs-one (OvO) multi-class classifier works better than the one-vs-all (OvA) multi-class classifier. I suspect this is because the OvO classifier has a better tie-breaking mechanism than the OvA classifier due to the increased number of classifiers. Many of the digits such as 0/6, 4/9, 3/8 were confused for each other due to their similar characteristics and ambiguity throughout the dataset. Almost all the errors I noticed were because of this sort of similarity between digits, as can be observed in the confusion matrices. An easy way to fix the OvA classifier would be to use a different classifier once a tie is detected which would greatly improve its accuracy.

Both models generalized well on the test data, meaning that their accuracy did not greatly drop off when tested on the test data. The digits that showed the most errors were 5,8,9 for the OvA multi-class classifier and 3,8,9 for the OvO multi-class classifier.

Problem 2: Randomized Feature Based Least Square Classifiers

Intuition

Here we aim to solve the same least squares problem as before, but the input vector is mapped to the "feature space" by the map $h: R^d \to R^L$.

$$\min_{\beta,\alpha} \sum_{i=1}^{N} (y_i - \beta^T h(x_i) - \alpha)^2$$

Where $x \in R^d, y_i \in \{-1, 1\}$, (x_i, y_i) is the i^{th} data point with x_i being the image pixel data and y is the corresponding label of -1 or 1.

This problem is only different because the input is $h(x_i)$ which is in the feature space L, where L is a whole number greater than or equal to 0. Each $h(x_i)$ is given by:

$$\mathbf{h}(\mathbf{x}) := \begin{bmatrix} g(\mathbf{w}_1^T\mathbf{x} + b_1) \\ g(\mathbf{w}_2^T\mathbf{x} + b_2) \\ \vdots \\ g(\mathbf{w}_L^T\mathbf{x} + b_L) \end{bmatrix}$$

And g(.) is any real-valued function. In this report, I will compare performance for L=1000 for the training and test data for both the one-versus-one and one-versus-all multi-class classifiers. Additionally, graphs will be created comparing the value of L to the error rate, again, for both the one-versus-one and one-versus-all multi-class classifiers (training and test data). In these trials I used:

- g(x) = x
- g(x) = sigmoid(x)
- g(x) = sin(x)
- g(x) = relu(x)

Implementation for Feature Mapping

This is the full function for the feature mapping creation

```python
def identity(x):
    return x


def sigmoid(x):
    return expit(x)


def sinusoidal(x):
    return np.sin(x)


def relu(x):
    return np.maximum(x, 0)


# Randomized feature mapping function

def randomized_feature_mapping(X, L, g=np.identity):
    """
    Generate a random feature mapping for the input data X using the function
g.

    Parameters:
    - X: numpy array of shape (n_samples, n_features), input data
    - L: int, number of random features to generate
    - g: function, activation function to use in the feature mapping

    Returns:
    - h_X: numpy array of shape (n_samples, L), transformed feature space
    - W: numpy array of shape (L, n_features), random weight matrix
    - b: numpy array of shape (L,), random bias vector
    """
    n_samples, n_features = X.shape

    # Generate random matrix W and bias vector b
    W = np.random.normal(0, 1, (L, n_features))  # W ~ N(0, 1) with shape (L,
n_features)
    b = np.random.normal(0, 1, L)  # b ~ N(0, 1) with shape (L,)

    # Compute random feature mapping
    h_X = g(np.dot(X, W.T) + b)  # h(X) = g(W * X^T + b)

    return h_X, W, b

def randomized_feature_mapping_with_params(X, W, b, g=np.identity):
    """
    Apply a precomputed random feature mapping to new data X using the given
W and b.

    Parameters:
    - X: numpy array of shape (n_samples, n_features), input data
    - W: numpy array of shape (L, n_features), random weight matrix
    - b: numpy array of shape (L,), random bias vector
    - g: function, activation function to use in the feature mapping

    Returns:
```

```
    - h_X: numpy array of shape (n_samples, L), transformed feature space
    """
    return g(np.dot(X, W.T) + b)  # h(X) = g(W * X^T + b)
```

Note that there is a randomized feature mapping with params so I can apply an identical feature mapping to the test data that was used on the training data.

One Versus One with Randomized Features

Implementation of L=1000 for various feature mappings:

Note that this implementation does not feature imports or the data loading, that is trivial for this part, and similar to up above. Here genHelpers refers to helper functions that are common between both OneVsOne and OneVsAll. This entire file -- genHelpers.py – can be found in Appendix C. In this script, helpers refers to helpersOneVsOne.py

```
activationFunctions = [genHelpers.identity, genHelpers.relu,
                       genHelpers.sigmoid, genHelpers.sinusoidal]
for activationFunction in activationFunctions:
    # Generate consistent random mapping
    transformed_training_images, W, b =
genHelpers.randomized_feature_mapping(images, L=1000, g=activationFunction)
    transformed_images =
genHelpers.randomized_feature_mapping_with_params(test_images, W, b,
g=activationFunction)



    # Train classifiers
    binary_classifiers =
helpers.train_ovo_classifiers(transformed_training_images, labels,
num_classes=10)

    # Predict on test and training data
    predicted_labels = helpers.predict_ovo(transformed_images,
binary_classifiers, num_classes=10)
    predicted_training_labels =
helpers.predict_ovo(transformed_training_images, binary_classifiers,
num_classes=10)

    # Calculate errors
    error = genHelpers.classwise_error_rate(predicted_labels, test_labels)
    total_error = genHelpers.error_rate(predicted_labels, test_labels)
    error2 = genHelpers.classwise_error_rate(predicted_training_labels,
labels)
    total_error2 = genHelpers.error_rate(predicted_training_labels, labels)

    # Print results
    print(f"Stats for {activationFunction.__name__} non-linearity for test
data:")
    print(f"The total error for the test data was: {total_error}")
```

```
    for num, err in error.items():
        print(f"Error rate for class {num}: {err}")

    print(f"Stats for {activationFunction.__name__} non-linearity for
training data:")
    print(f"The total error for the training data was: {total_error2}")
    for num, err in error2.items():
        print(f"Error rate for class {num}: {err}")
```

This program's output is below:

Stats for **identity** non-linearity for **test** data:
The total error for the test data was: 0.0704
Error rate for class 0: 1.94%
Error rate for class 1: 1.41%
Error rate for class 2: 9.11%
Error rate for class 3: 8.42%
Error rate for class 4: 5.30%
Error rate for class 5: 10.43%
Error rate for class 6: 5.32%
Error rate for class 7: 7.00%
Error rate for class 8: 13.76%
Error rate for class 9: 8.72%

Stats for **identity** non-linearity for **training** data:
The total error for the training data was: 0.06195
Error rate for class 0: 1.98%
Error rate for class 1: 1.77%
Error rate for class 2: 7.30%
Error rate for class 3: 8.95%
Error rate for class 4: 4.38%
Error rate for class 5: 8.25%
Error rate for class 6: 3.84%
Error rate for class 7: 6.05%
Error rate for class 8: 11.90%
Error rate for class 9: 8.27%

Stats for **relu** non-linearity for **test** data:
The total error for the test data was: 0.0347
Error rate for class 0: 1.02%
Error rate for class 1: 0.79%
Error rate for class 2: 4.75%
Error rate for class 3: 3.66%
Error rate for class 4: 2.85%
Error rate for class 5: 4.37%

Error rate for class 6: 1.98%
Error rate for class 7: 4.47%
Error rate for class 8: 5.34%
Error rate for class 9: 5.75%

Stats for **relu** non-linearity for **training** data:
The total error for the training data was: 0.021983333333333334
Error rate for class 0: 0.86%
Error rate for class 1: 0.76%
Error rate for class 2: 2.20%
Error rate for class 3: 3.36%
Error rate for class 4: 1.90%
Error rate for class 5: 2.69%
Error rate for class 6: 1.12%
Error rate for class 7: 2.33%
Error rate for class 8: 3.09%
Error rate for class 9: 3.87%

Stats for **sigmoid** non-linearity for **test** data:
The total error for the test data was: 0.0425
Error rate for class 0: 1.63%
Error rate for class 1: 0.97%
Error rate for class 2: 4.55%
Error rate for class 3: 5.15%
Error rate for class 4: 3.67%
Error rate for class 5: 6.39%
Error rate for class 6: 2.40%
Error rate for class 7: 5.16%
Error rate for class 8: 6.16%
Error rate for class 9: 6.94%

Stats for **sigmoid** non-linearity for **training** data:
The total error for the training data was: 0.029483333333333334
Error rate for class 0: 0.96%
Error rate for class 1: 1.05%
Error rate for class 2: 3.24%
Error rate for class 3: 4.70%
Error rate for class 4: 2.58%
Error rate for class 5: 3.76%
Error rate for class 6: 1.61%
Error rate for class 7: 3.26%
Error rate for class 8: 3.97%
Error rate for class 9: 4.61%

Stats for **sinusoidal** non-linearity for **test** data:
The total error for the test data was: 0.8661
Error rate for class 0: 89.49%
Error rate for class 1: 51.89%
Error rate for class 2: 90.50%
Error rate for class 3: 89.90%
Error rate for class 4: 90.94%
Error rate for class 5: 93.72%
Error rate for class 6: 91.02%
Error rate for class 7: 90.56%
Error rate for class 8: 91.68%
Error rate for class 9: 91.97%

Stats for **sinusoidal** non-linearity for **training** data:
The total error for the training data was: 0.78555
Error rate for class 0: 80.52%
Error rate for class 1: 45.30%
Error rate for class 2: 80.75%
Error rate for class 3: 80.00%
Error rate for class 4: 84.89%
Error rate for class 5: 86.28%
Error rate for class 6: 83.05%
Error rate for class 7: 81.23%
Error rate for class 8: 84.34%
Error rate for class 9: 84.37%

Now Continuing, we have the implementation for the error rate using various activation functions plotted as a function of L. This is done on both the training and test data as well. First, I present the implementation, then the results:

Note that the following scripts are the high-level script which calls helpers from Appendix B and Appendix C.

Training Data Graph Script

```
#initialize variables for graph
L_values = range(100, 1501, 50)
error_rates = []
activationFunctions = [genHelpers.sigmoid, genHelpers.identity, 
genHelpers.relu, genHelpers.sinusoidal]
for activationFunction in activationFunctions:
    print(activationFunction.__name__)
    error_rates.clear()
    #loop through L values
    for L in L_values:
        #create random features and train classifiers on them
```

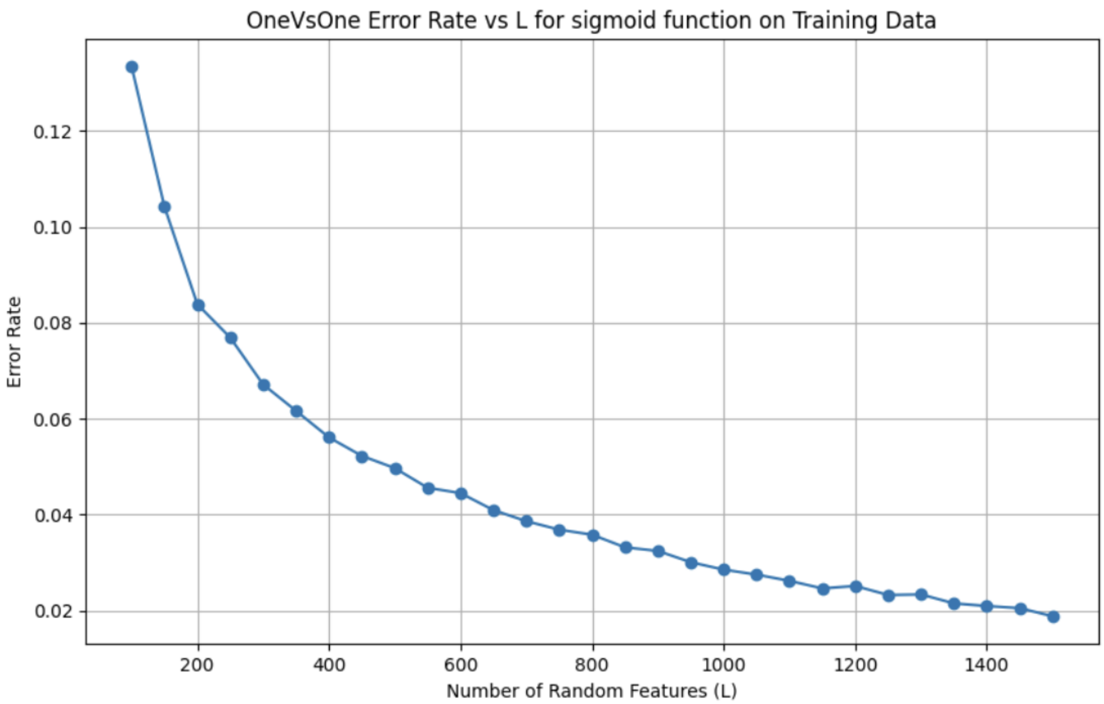
```
        transformed_training_images, W, b =
genHelpers.randomized_feature_mapping(images, L, g=activationFunction)

        # Train classifiers
        binary_classifiers =
helpers.train_ovo_classifiers(transformed_training_images, labels,
num_classes=10)
        #predict labels for training data
        predicted_labels = helpers.predict_ovo(transformed_training_images,
binary_classifiers, num_classes=10)

        #calc error and reset for next loop
        total_error = genHelpers.error_rate(predicted_labels, labels)
        error_rates.append(total_error)
        predicted_labels = np.array([])
        binary_classifiers = np.array([])

        print(L)
        print(error_rates)

    #plot L vs Error Rates
    plt.figure(figsize=(10, 6))
    plt.plot(L_values, error_rates, marker='o')
    plt.xlabel("Number of Random Features (L)")
    plt.ylabel("Error Rate")
    plt.title("OneVsOne Error Rate vs L for {} function on Training
Data".format(activationFunction.__name__))
    plt.grid()
    plt.show()
```

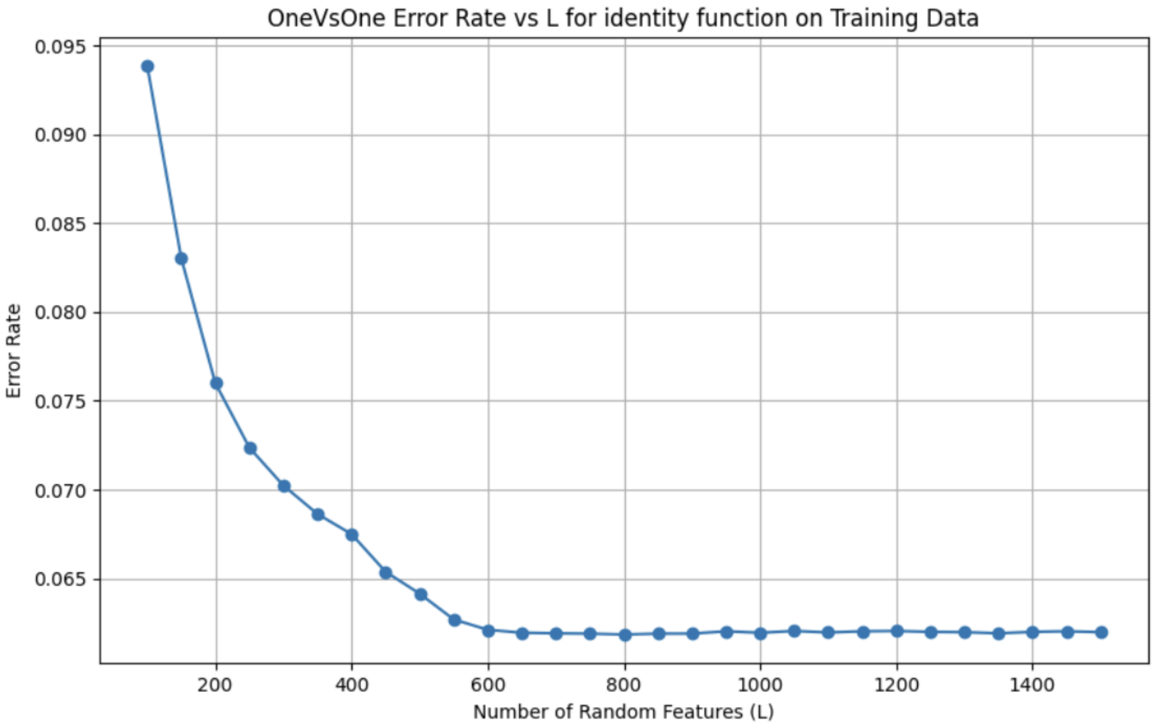
Test Data Graph Script:

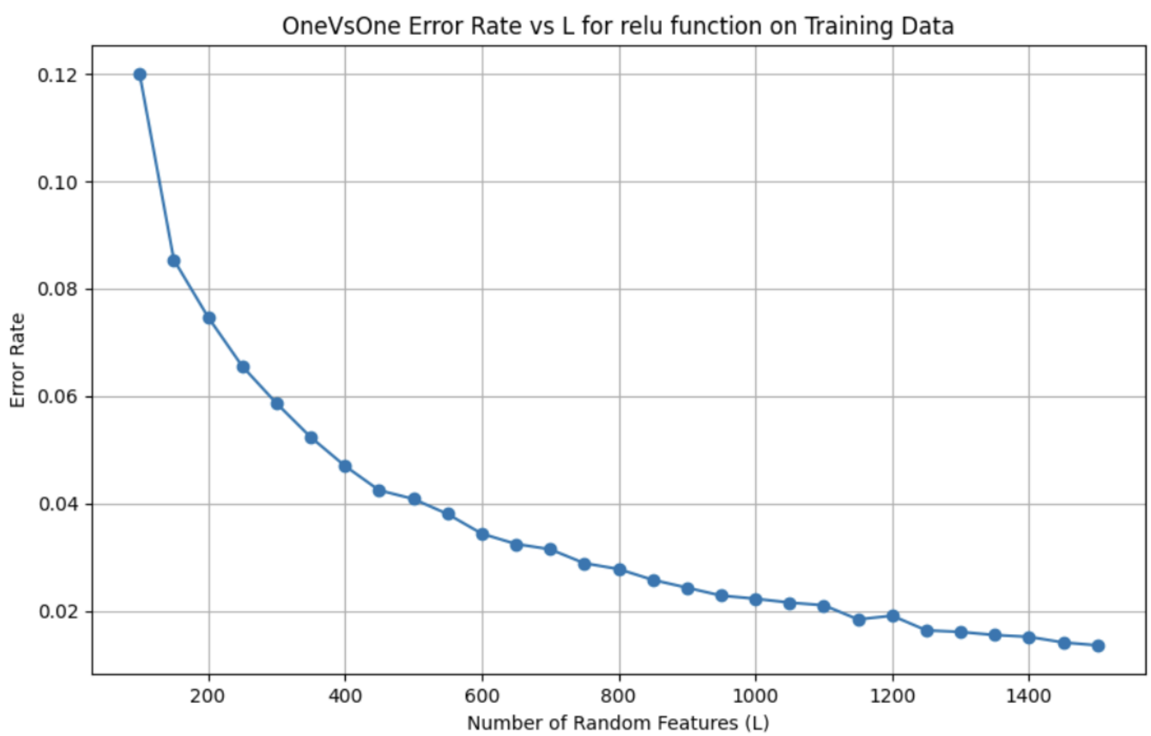
```
# initialize variables for graph
L_values = range(100, 1501, 50)
error_rates = []
error_rates.clear()

# loop through L values
for L in L_values:
    # create random features and train classifiers on them
    transformed_images,W, b = genHelpers.randomized_feature_mapping(images,
L, g=genHelpers.identity)
    binary_classifiers = helpers.train_ovo_classifiers(transformed_images,
labels, num_classes=10)

    transformed_test_images =
genHelpers.randomized_feature_mapping_with_params(test_images, W, b,
g=genHelpers.identity)        # predict labels for test data
    predicted_labels = helpers.predict_ovo(transformed_test_images,
binary_classifiers, num_classes=10)

    # calc error and reset for next loop
    total_error = genHelpers.error_rate(predicted_labels, test_labels)
    error_rates.append(total_error)

    print(L)
```

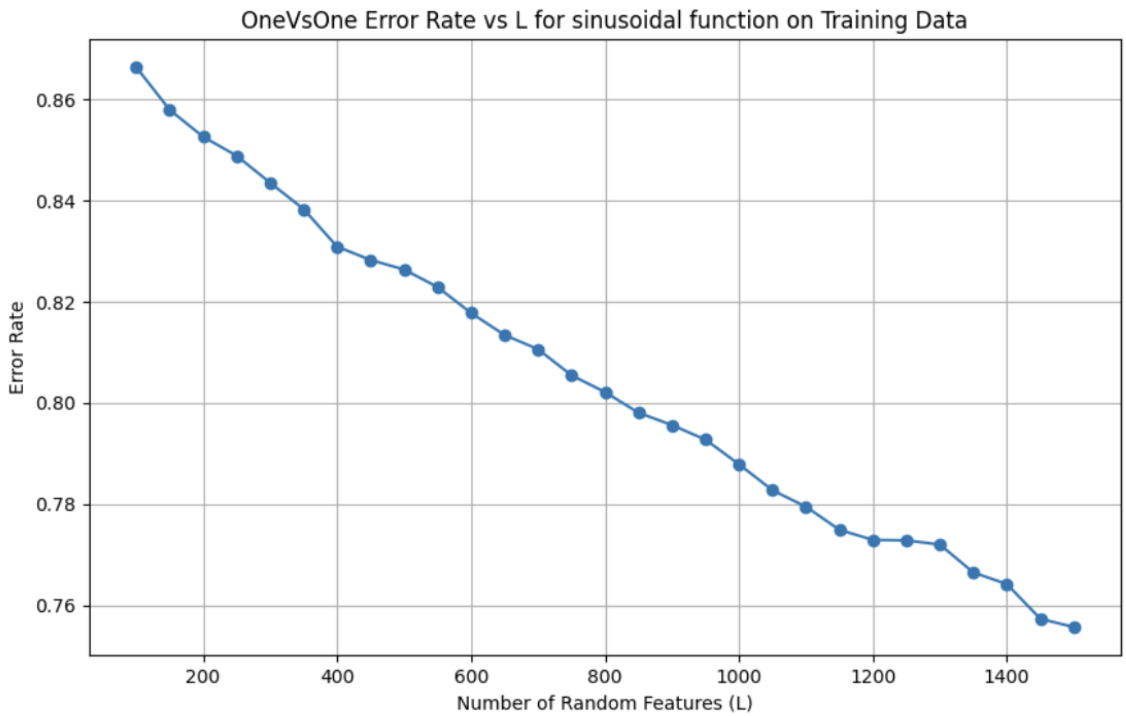
```
    print(error_rates)

# plot L vs Error Rates
plt.figure(figsize=(10, 6))
plt.plot(L_values, error_rates, marker='o')
plt.xlabel("Number of Random Features (L)")
plt.ylabel("Error Rate")
plt.title("Error Rate as a Function of Number of Random Features (L) for
Identity function on Test Data")
plt.grid()
plt.show()
```

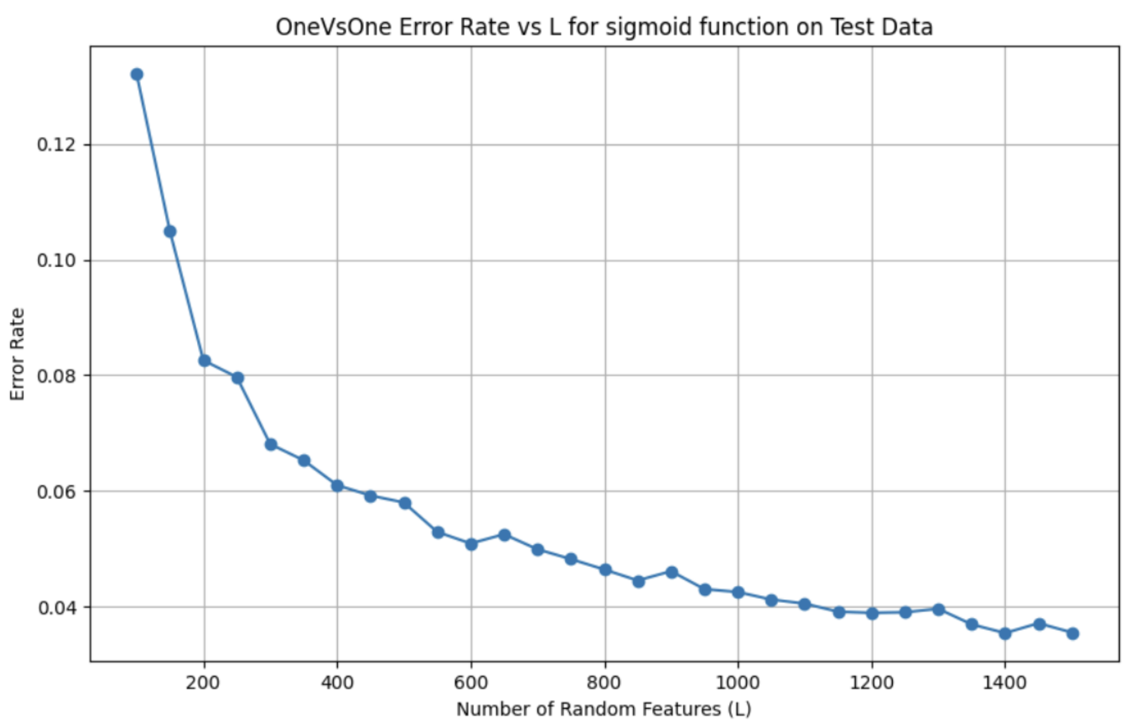
Graphs for Training and Test Data:

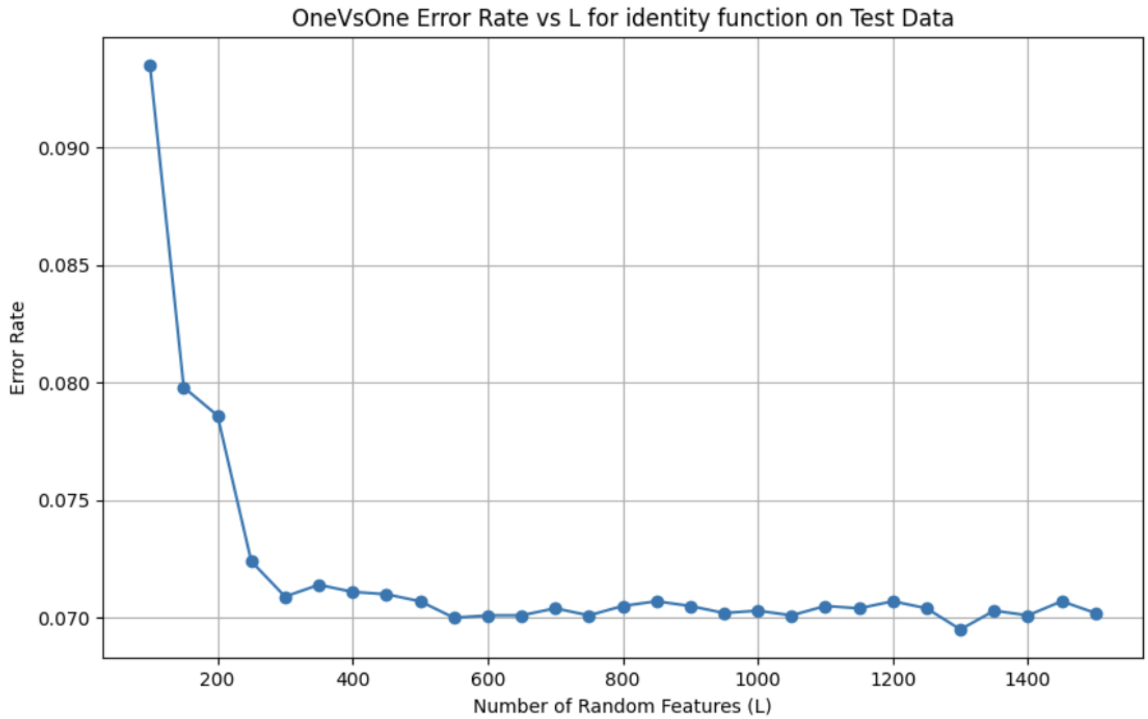


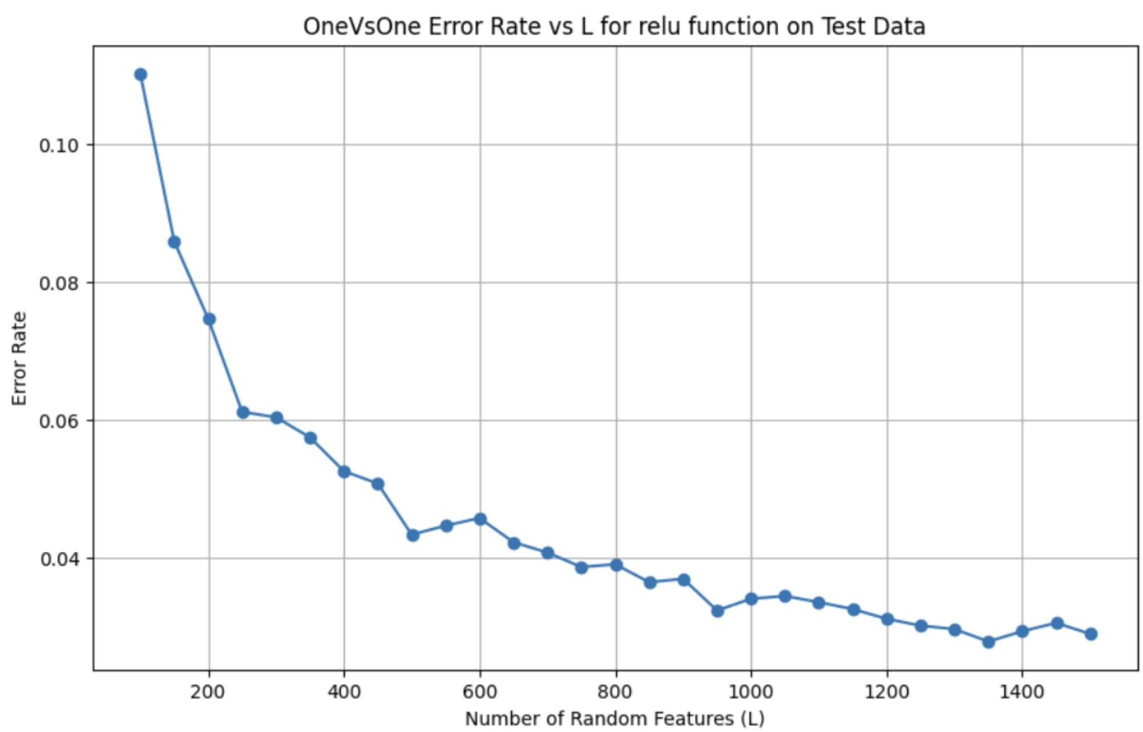

OneVsOne Error Rate vs L for identity function on Training Data
Error Rate
Number of Random Features (L)
0.095
0.090
0.085
0.080
0.075
0.070
0.065
200
400
600
800
1000
1200
1400

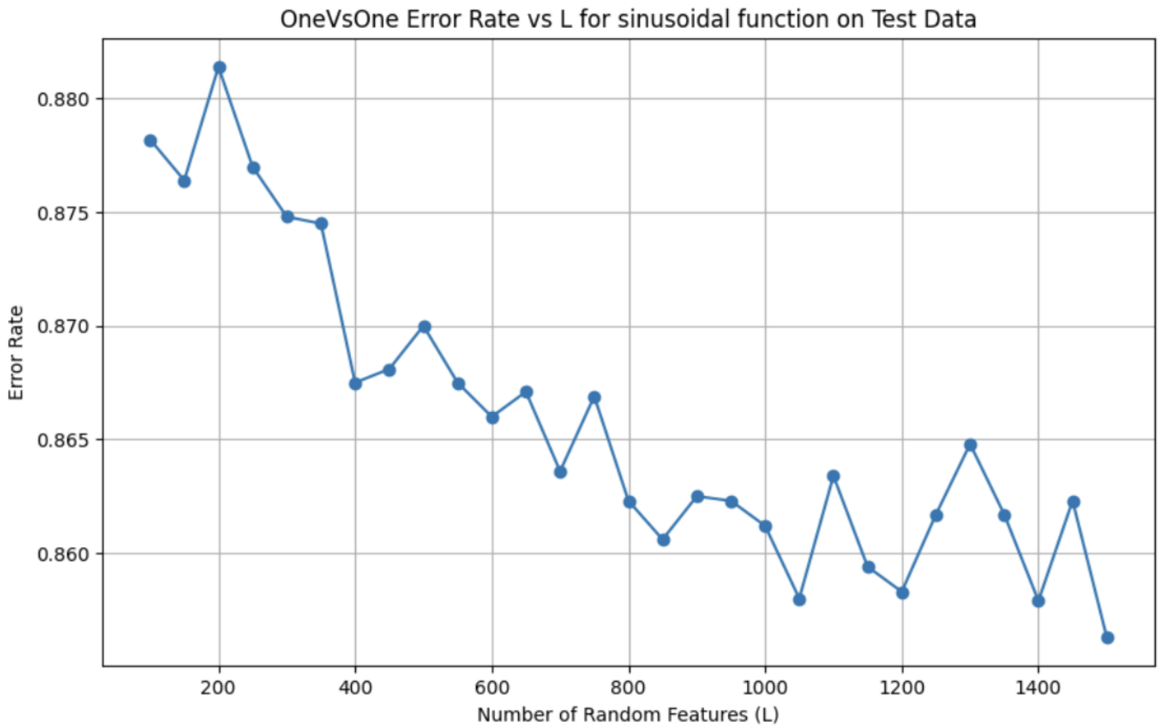

OneVsOne Error Rate vs L for relu function on Training Data
Error Rate
Number of Random Features (L)
0.12
0.10
0.08
0.06
0.04
0.02
200
400
600
800
1000
1200
1400


OneVsOne Error Rate vs L for sinusoidal function on Training Data
Error Rate
Number of Random Features (L)


OneVsOne Error Rate vs L for sigmoid function on Test Data
Error Rate
Number of Random Features (L)


OneVsOne Error Rate vs L for identity function on Test Data
Error Rate
0.090
0.085
0.080
0.075
0.070
200
400
600
800
1000
1200
1400
Number of Random Features (L)


OneVsOne Error Rate vs L for relu function on Test Data
Error Rate
0.10
0.08
0.06
0.04
200
400
600
800
1000
1200
1400
Number of Random Features (L)



One Versus All with Randomized Features

Implementation of L=1000 for various feature mappings:

Here genHelpers refers to helper functions that are common between both OneVsOne and OneVsAll. This entire file -- genHelpers.py – can be found in Appendix C. In this script, helpers refers to helpersOneVsAll.py

```
activationFunctions = [genHelpers.identity, genHelpers.relu,
genHelpers.sigmoid, genHelpers.sinusoidal]
L = 1000  # Number of random features
num_classes = 10  # Number of classes (e.g., digits 0-9 for MNIST)

for activationFunction in activationFunctions:
    print(f"Feature mapping using {activationFunction.__name__} non-
linearity:")

    # Generate consistent random feature mapping for training
    transformed_training_images, W, b =
genHelpers.randomized_feature_mapping(images, L=L, g=activationFunction)

    # Apply the same random mapping to test data
```

```
    transformed_test_images =
genHelpers.randomized_feature_mapping_with_params(test_images, W, b,

g=activationFunction)

    # Train classifiers
    betas = []
    alphas = []
    for k in range(num_classes):
        # Label data for class k vs all others
        binary_labels = helpers.labelBinaryData(k, labels)

        # Train a binary classifier for this class
        beta, alpha =
helpers.train_binary_classifier(transformed_training_images, binary_labels)

        # Store the classifier's parameters
        betas.append(beta)
        alphas.append(alpha)

    # Predict on test data
    predicted_test_labels =
helpers.predict_one_vs_all_full(transformed_test_images, betas, alphas)

    # Predict on training data
    predicted_training_labels =
helpers.predict_one_vs_all_full(transformed_training_images, betas, alphas)

    # Calculate errors for test data
    test_error = genHelpers.error_rate(predicted_test_labels, test_labels)
    test_classwise_error =
genHelpers.classwise_error_rate(predicted_test_labels, test_labels)

    # Calculate errors for training data
    train_error = genHelpers.error_rate(predicted_training_labels, labels)
    train_classwise_error =
genHelpers.classwise_error_rate(predicted_training_labels, labels)

    # Print stats for test data
    print(f"Stats for {activationFunction.__name__} non-linearity on test
data:")
    print(f"The total error for the test data was: {test_error}")
    for num, err in test_classwise_error.items():
        print(f"Error rate for class {num}: {err}")

    # Print stats for training data
    print(f"Stats for {activationFunction.__name__} non-linearity on training
data:")
    print(f"The total error for the training data was: {train_error}")
    for num, err in train_classwise_error.items():
        print(f"Error rate for class {num}: {err}")

    print()  # Add a blank line between activation functions
```

Below are the results for this script:

Feature mapping using identity non-linearity:
Stats for **identity** non-linearity on **test** data:
The total error for the test data was: 0.1397
Error rate for class 0: 3.67%
Error rate for class 1: 2.47%
Error rate for class 2: 21.22%
Error rate for class 3: 12.87%
Error rate for class 4: 10.29%
Error rate for class 5: 26.12%
Error rate for class 6: 8.66%
Error rate for class 7: 14.01%
Error rate for class 8: 22.07%
Error rate for class 9: 20.61%

Stats for **identity** non-linearity on **training** data:
The total error for the training data was: 0.14226666666666668
Error rate for class 0: 4.07%
Error rate for class 1: 2.88%
Error rate for class 2: 19.57%
Error rate for class 3: 15.87%
Error rate for class 4: 10.78%
Error rate for class 5: 26.38%
Error rate for class 6: 7.47%
Error rate for class 7: 13.39%
Error rate for class 8: 24.59%
Error rate for class 9: 19.87%

Feature mapping using relu non-linearity:
Stats for **relu** non-linearity on **test** data:
The total error for the test data was: 0.0567
Error rate for class 0: 1.73%
Error rate for class 1: 1.23%
Error rate for class 2: 7.75%
Error rate for class 3: 6.44%
Error rate for class 4: 6.01%
Error rate for class 5: 7.06%
Error rate for class 6: 3.65%
Error rate for class 7: 6.91%
Error rate for class 8: 8.52%
Error rate for class 9: 7.93%

Stats for **relu** non-linearity on **training** data:
The total error for the training data was: 0.05483333333333333
Error rate for class 0: 1.94%

Error rate for class 1: 1.59%
Error rate for class 2: 6.55%
Error rate for class 3: 7.18%
Error rate for class 4: 5.60%
Error rate for class 5: 7.29%
Error rate for class 6: 2.70%
Error rate for class 7: 5.59%
Error rate for class 8: 8.89%
Error rate for class 9: 8.17%

Feature mapping using sigmoid non-linearity:
Stats for **sigmoid** non-linearity on **test** data:
The total error for the test data was: 0.0643
Error rate for class 0: 1.53%
Error rate for class 1: 1.15%
Error rate for class 2: 9.59%
Error rate for class 3: 7.72%
Error rate for class 4: 4.79%
Error rate for class 5: 10.09%
Error rate for class 6: 3.65%
Error rate for class 7: 7.78%
Error rate for class 8: 9.75%
Error rate for class 9: 9.02%

Stats for **sigmoid** non-linearity on **training** data:
The total error for the training data was: 0.06545
Error rate for class 0: 2.21%
Error rate for class 1: 2.06%
Error rate for class 2: 7.94%
Error rate for class 3: 9.12%
Error rate for class 4: 5.87%
Error rate for class 5: 9.52%
Error rate for class 6: 3.89%
Error rate for class 7: 6.37%
Error rate for class 8: 10.20%
Error rate for class 9: 9.08%

Feature mapping using sinusoidal non-linearity:
Stats for **sinusoidal** non-linearity on **test** data:
The total error for the test data was: 0.8561
Error rate for class 0: 90.31%
Error rate for class 1: 36.83%
Error rate for class 2: 89.92%
Error rate for class 3: 90.00%

Error rate for class 4: 93.18%
Error rate for class 5: 95.29%
Error rate for class 6: 92.69%
Error rate for class 7: 90.76%
Error rate for class 8: 91.89%
Error rate for class 9: 93.16%

Stats for **sinusoidal** non-linearity on **training** data:
The total error for the training data was: 0.79915
Error rate for class 0: 86.11%
Error rate for class 1: 31.52%
Error rate for class 2: 84.99%
Error rate for class 3: 84.90%
Error rate for class 4: 87.56%
Error rate for class 5: 89.23%
Error rate for class 6: 86.87%
Error rate for class 7: 81.98%
Error rate for class 8: 86.52%
Error rate for class 9: 86.80%

Now Continuing, we have the implementation for the error rate using various activation functions plotted as a function of L. This is done on both the training and test data as well. First, I present the implementation, then the results:

Note that this is the high-level script which calls helpers from Appendix A and Appendix C.

Training Data Graph Script

```
L_values = range(100, 1501, 50)
num_classes = 10  # For MNIST, this would be 10 for digits 0-9
betas = []
alphas = []
error_rates = []
activationFunctions = [genHelpers.sigmoid, genHelpers.relu,
genHelpers.sinusoidal, genHelpers.identity]
for activationFunction in activationFunctions:
    error_rates.clear()
    for L in L_values:
        betas.clear()
        alphas.clear()
        transformed_training_images, W, b =
genHelpers.randomized_feature_mapping(images, L=L, g=activationFunction)
        for k in range(num_classes):
            # Label data for class k vs all others
            binary_labels = helpers.labelBinaryData(k, labels)
```

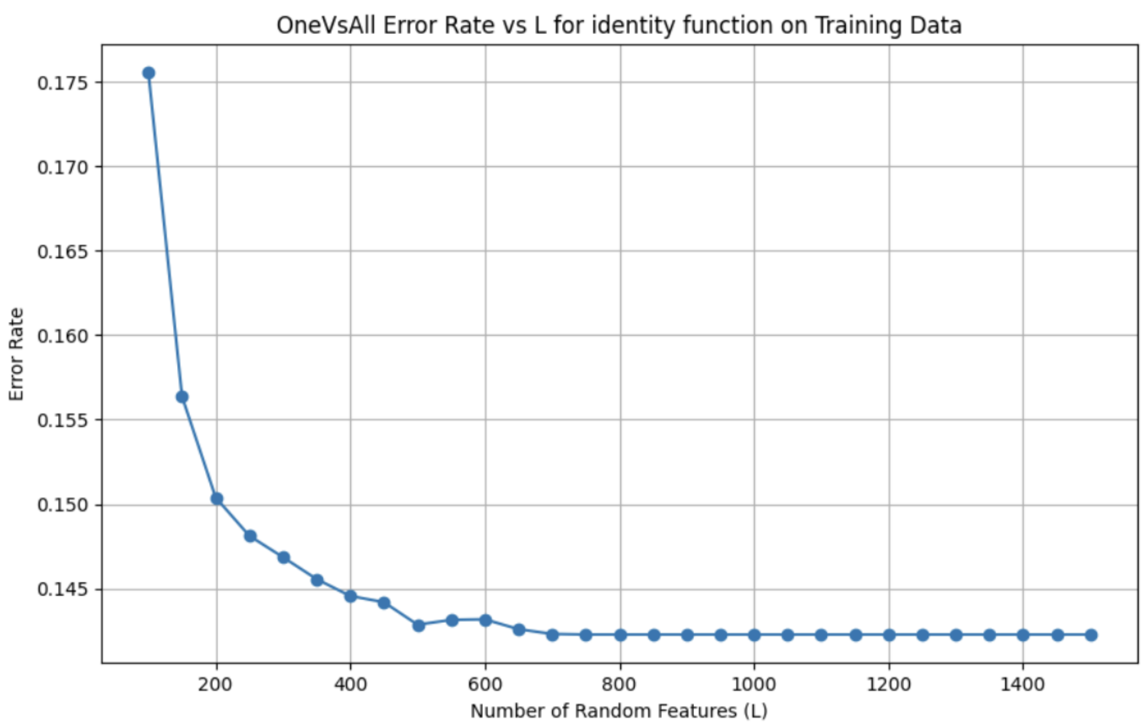
```
            # Train a binary classifier for this class
            beta, alpha =
helpers.train_binary_classifier(transformed_training_images, binary_labels)

            # Store the classifier's parameters
            betas.append(beta)
            alphas.append(alpha)
        predicted =
helpers.predict_one_vs_all_full(transformed_training_images, betas, alphas)
        total_error = genHelpers.error_rate(predicted, labels)
        error_rates.append(total_error)
        print(L)
        print(error_rates)

    plt.figure(figsize=(10, 6))
    plt.plot(L_values, error_rates, marker='o')
    plt.xlabel("Number of Random Features (L)")
    plt.ylabel("Error Rate")
    plt.title("OneVsAll Error Rate vs L for {} function on Training
Data".format(activationFunction.__name__))
    plt.grid()
    plt.show()
```

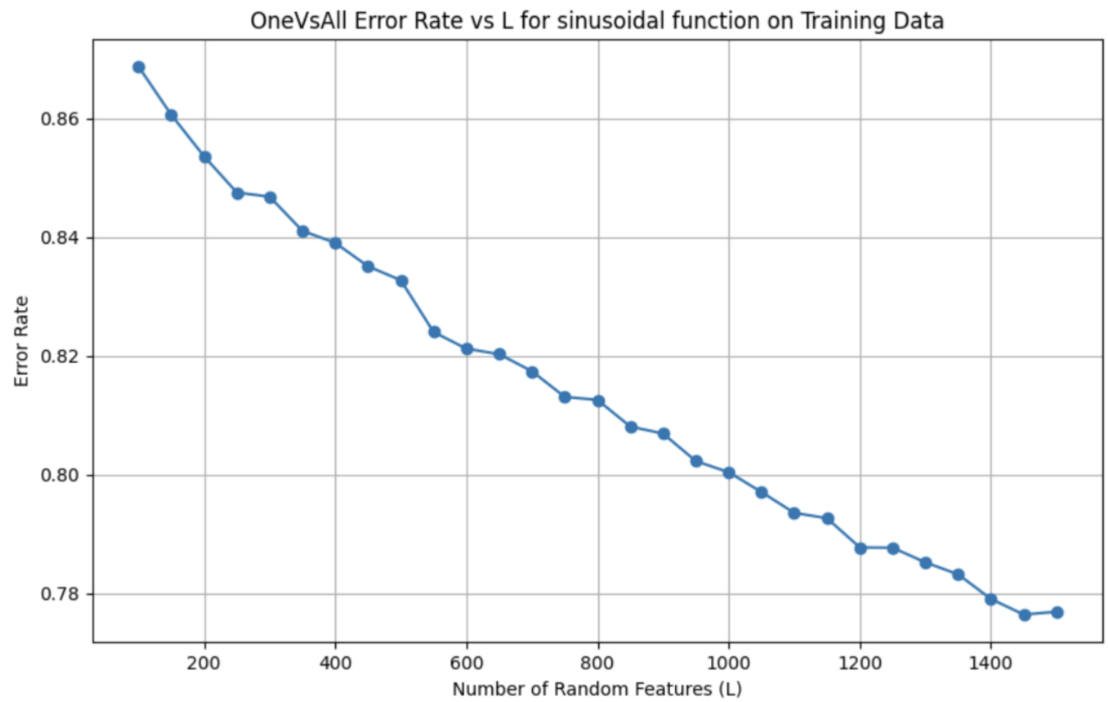
Test Data Graph Script:

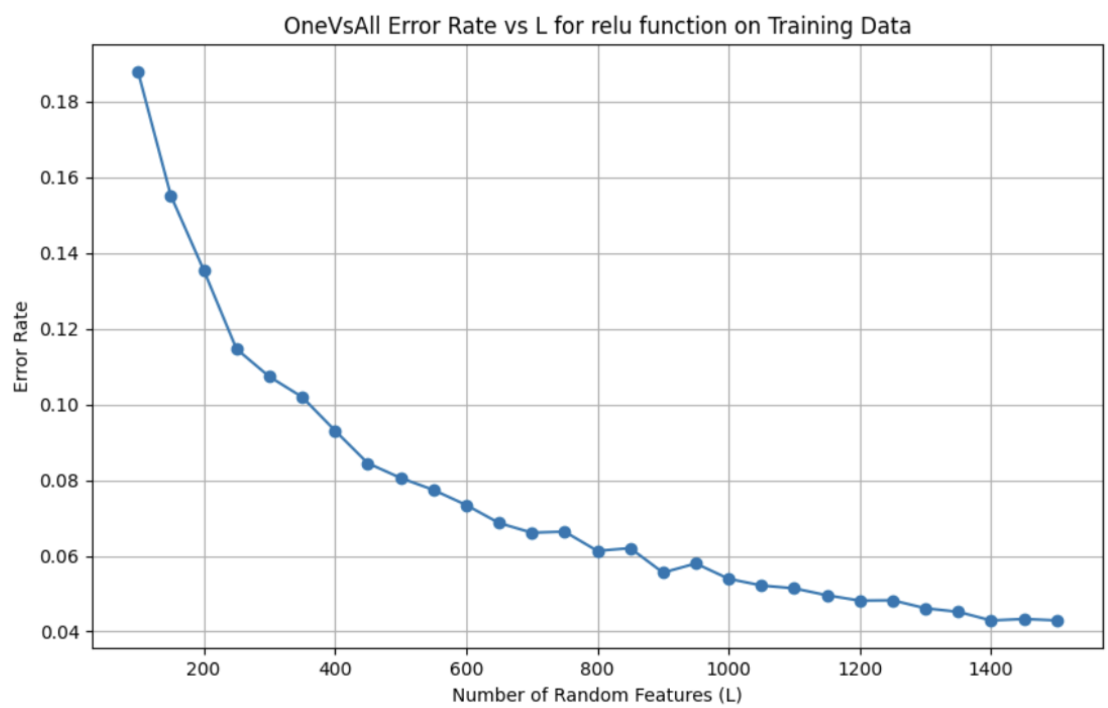
```
L_values = range(100, 1501, 50)
num_classes = 10  # For MNIST, this would be 10 for digits 0-9
betas = []
alphas = []
error_rates = []
activationFunctions = [genHelpers.sigmoid, genHelpers.relu,
genHelpers.sinusoidal, genHelpers.identity]
for activationFunction in activationFunctions:
    error_rates.clear()
    for L in L_values:
        betas.clear()
        alphas.clear()
        transformed_images, W, b =
genHelpers.randomized_feature_mapping(images, L, g=activationFunction)
        transformed_test_images =
genHelpers.randomized_feature_mapping_with_params(test_images, W, b ,
g=activationFunction)
        for k in range(num_classes):
            # Label data for class k vs all others
            binary_labels = helpers.labelBinaryData(k, labels)

            # Train a binary classifier for this class
            beta, alpha = helpers.train_binary_classifier(transformed_images,
binary_labels)

            # Store the classifier's parameters
            betas.append(beta)
            alphas.append(alpha)
```

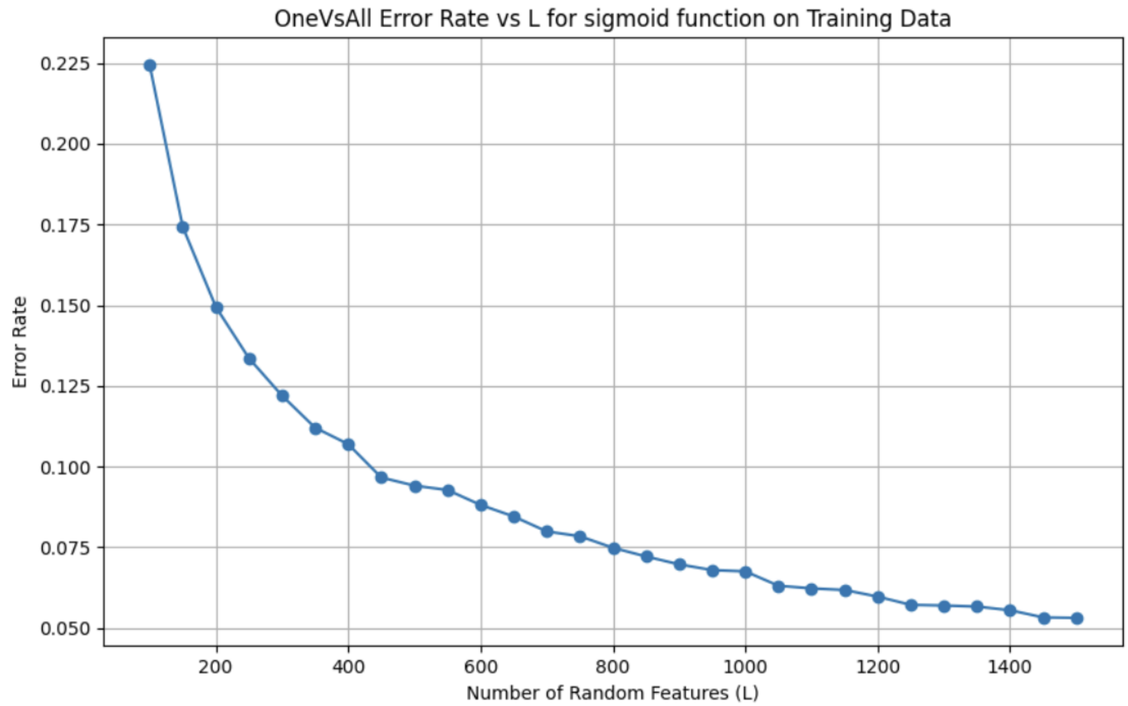
```
        predicted = helpers.predict_one_vs_all_full(transformed_test_images,
betas, alphas)
        total_error = genHelpers.error_rate(predicted, test_labels)
        error_rates.append(total_error)
        print(L)
        print(error_rates)

    plt.figure(figsize=(10, 6))
    plt.plot(L_values, error_rates, marker='o')
    plt.xlabel("Number of Random Features (L)")
    plt.ylabel("Error Rate")
    plt.title("OneVsAll Error Rate vs L for {} function for Test
Data".format(activationFunction.__name__))
    plt.grid()
    plt.show()
```

Graphs for Training and Test Data:

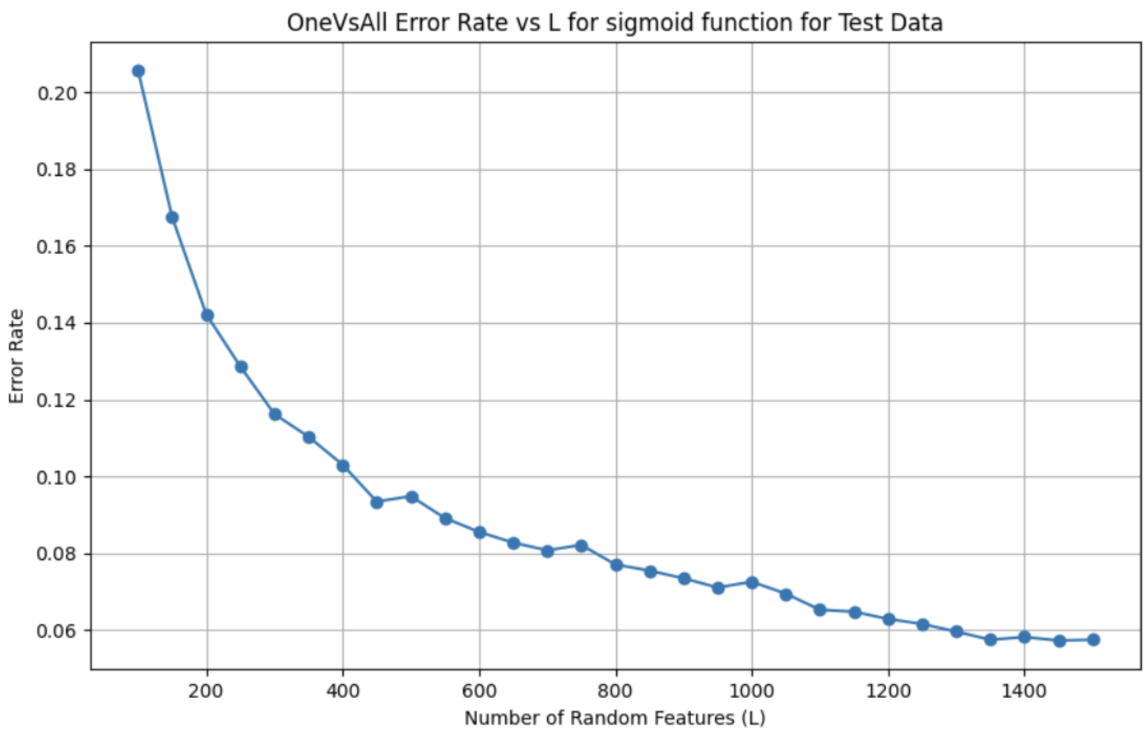



OneVsAll Error Rate vs L for sinusoidal function on Training Data
Error Rate
0.86
0.84
0.82
0.80
0.78
200
400
600
800
1000
1200
1400
Number of Random Features (L)

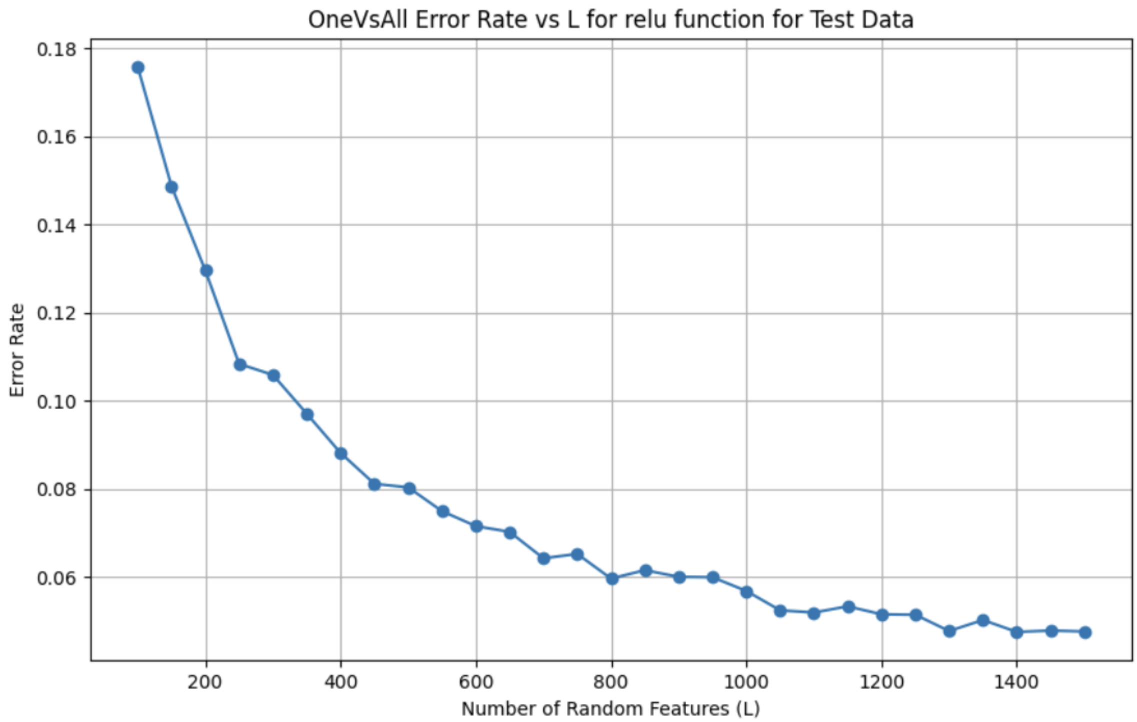

OneVsAll Error Rate vs L for relu function on Training Data
Error Rate
0.18
0.16
0.14
0.12
0.10
0.08
0.06
0.04
200
400
600
800
1000
1200
1400
Number of Random Features (L)

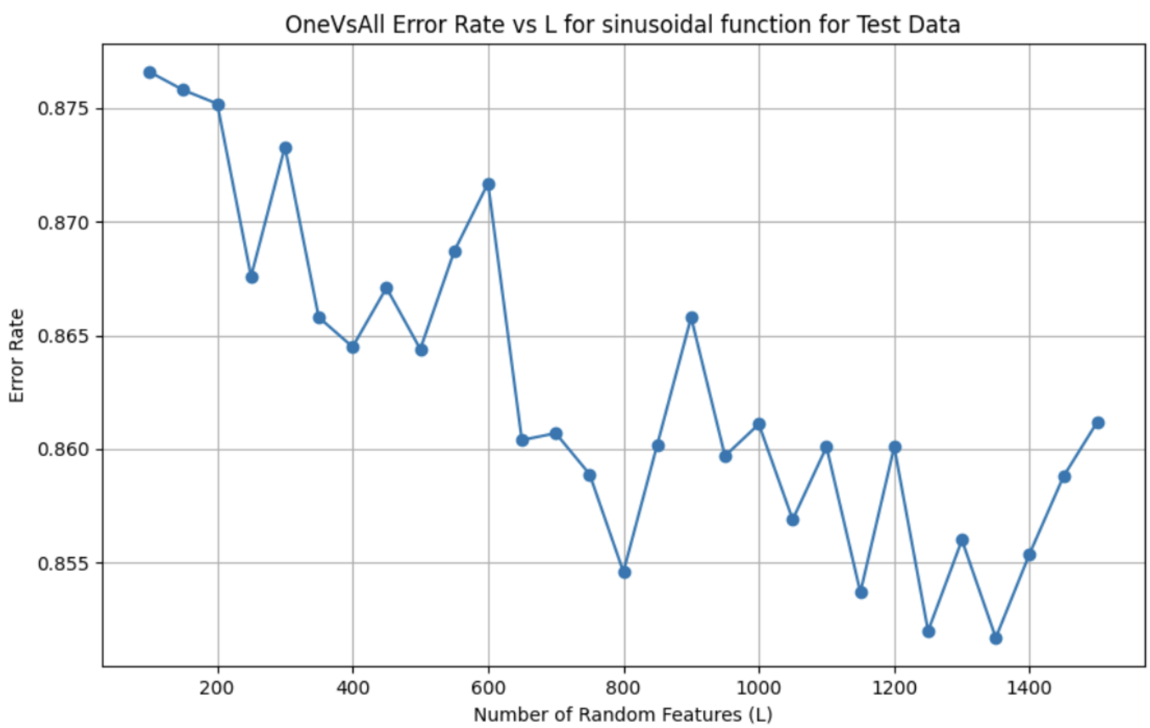


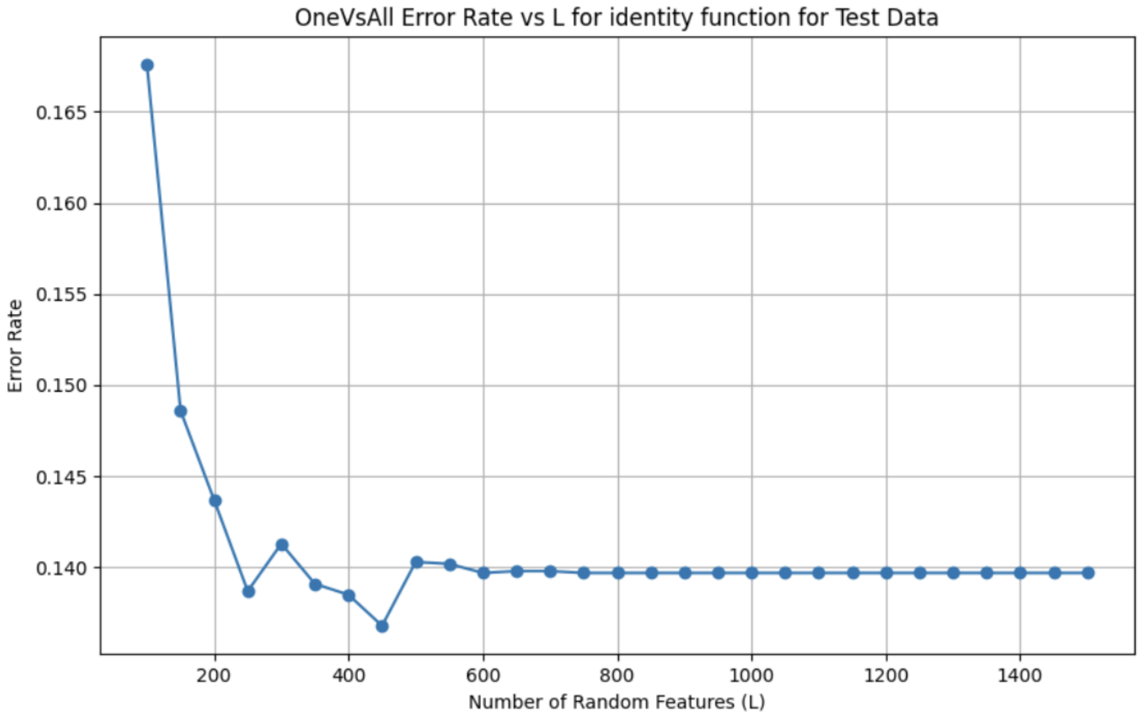
OneVsAll Error Rate vs L for sigmoid function on Training Data
Error Rate
Number of Random Features (L)



OneVsAll Error Rate vs L for sigmoid function for Test Data
Error Rate
Number of Random Features (L)


OneVsAll Error Rate vs L for relu function for Test Data
0.18
0.16
0.14
0.12
0.10
0.08
0.06
Error Rate
200
400
600
800
1000
1200
1400
Number of Random Features (L)


OneVsAll Error Rate vs L for sinusoidal function for Test Data
0.875
0.870
0.865
0.860
0.855
Error Rate
200
400
600
800
1000
1200
1400
Number of Random Features (L)



Problem 2 Takeaways

On training data, for L = 1000, the performance of the OvA multi-class classifier improved/worsened as follows: for sigmoid the error rate went from 13.97% -> 6.54%, for identity the error rate went from 13.97% -> 14.22% (Something I chalk up to the randomized feature mapping), for relu the error rate went from 13.97% -> 5.48%, and for sinusoidal the error rate went from 13.97% -> 79.91%. Therefore, for relu and sigmoid, the classifiers with random features worked better on the training data. Similar results were obtained for the test data, with minimal changes for the identity function, significant improvement for relu and sigmoid, and horrible error rate for sinusoidal (exact values above). Therefore, the feature-mapped classifiers generalized well to the test data, specifically relu and sigmoid.

On training data, for L=1000, the performance of the OvO multi-class classifier improved/worsened as follows: for sigmoid the error rate went from 6.2% -> 2.94%, for identity the error rate went from 6.2% -> 6.19% (Something I chalk up to the randomized feature mapping), for relu the error rate went from 6.2% -> 2.19%, and for sinusoidal the error rate went from 6.2% -> 78.55%. Therefore, for relu and sigmoid, the classifiers with random features worked better on the training data. Similar results were obtained for the test data, with minimal changes for the identity function, significant improvement for relu and sigmoid, and horrible error rate for sinusoidal (exact values above). Therefore, the feature-mapped classifiers generalized well to the test data, specifically relu and sigmoid.

When the number of features L was varied, similar observations occurred for both the OvO and OvA multi-class classifiers. For the sigmoid and relu functions, the error rate consisently decreased as L was increased in the range 100-1500, but as L increased, the rate at which error rate improved decreased. For the identity function, significant error rate improvement was observed until L = 500, after which, any additional dimensions to the feature space provided no improvement to the error rate. For the sinusoidal function, it seems that variation in error rate as a function of L can be chalked up to the randomness of the mapping and not the number of features L, i.e. no significant improvement was seen as L increased in the range 100-1500.

Appendix A

Below, is the entire helpersOneVsAll.py file:

```
import numpy as np

#if label is equal to the class, set y(i) = 1, otherwise y(i) = -1
def labelBinaryData(classifierNum, uncleanedLabels):
    cleanedLabels = []
    for label in uncleanedLabels[0]:
        if label == classifierNum:
            cleanedLabels.append(1)
        else:
            cleanedLabels.append(-1)
    return cleanedLabels

#solves the Normal equations
def train_binary_classifier(X, y):
    # Ensure y is a NumPy array, then reshape it to be a column vector
    y = np.array(y).reshape(-1, 1)

    # Add a bias term by augmenting X with a column of ones
    X_augmented = np.hstack([X, np.ones((X.shape[0], 1))])

    # Use the pseudo-inverse instead of the regular inverse
    params = np.linalg.pinv(X_augmented.T @ X_augmented) @ X_augmented.T @ y

    # Separate the β and α from the computed parameters
    beta = params[:-1].flatten()  # All but the last element
    alpha = params[-1, 0]  # Last element (bias term)

    return beta, alpha

#uses betas and alphas to predict the images
def predict_one_vs_all_full(X, betas, alphas):
    # Collect scores for each class
    scores = np.array([X @ beta + alpha for beta, alpha in zip(betas,
alphas)])

    # Choose the class with the highest score
    predicted_labels = np.argmax(scores, axis=0)

    return predicted_labels
```

Appendix B

Below, is the entire helpersOneVsOne.py file:

```
import numpy as np

#solve normal equation for every binary classifier
def train_ovo_classifier(X, Y, class_i, class_j):
    # Ensure Y is a 1D array for proper indexing
    y = Y.flatten()

    # Select only the samples belonging to class_i and class_j
    indices = (y == class_i) | (y == class_j)
    X_filtered = X[indices]
    y_filtered = y[indices]

    # Relabel: +1 for class_i, -1 for class_j
    y_binary = np.where(y_filtered == class_i, 1, -1)

    # Add a bias term to X for the linear model
    X_augmented = np.hstack([X_filtered, np.ones((X_filtered.shape[0], 1))])

    # Solve for [β; α] using the pseudo-inverse
    params = np.linalg.pinv(X_augmented.T @ X_augmented) @ X_augmented.T @
y_binary

    # Separate the β and α from params
    beta = params[:-1]  # All but the last element
    alpha = params[-1]  # Last element is the bias term

    return beta, alpha


#loop through every combination of binary classifiers with class combos i,j
#such that i<j and create binary classifier
def train_ovo_classifiers(X, y, num_classes=10):
    classifiers = {}

    for i in range(num_classes):
        for j in range(i + 1, num_classes):
            # Train the classifier for class i vs class j
            beta, alpha = train_ovo_classifier(X, y, i, j)
            classifiers[(i, j)] = (beta, alpha)

    return classifiers

#count votes up from the various classifiers and pick the number with the
most votes
def predict_ovo(X, classifiers, num_classes=10):
    votes = np.zeros((X.shape[0], num_classes), dtype=int)  # Matrix to count
votes for each class

    # Iterate through each classifier (i, j)
    for (i, j), (beta, alpha) in classifiers.items():
        # Calculate the decision for class i vs class j
        scores = X @ beta + alpha
```

```
        predictions = np.sign(scores)

        # Increment votes based on predictions
        votes[:, i] += (predictions == 1)  # Vote for class i
        votes[:, j] += (predictions == -1)  # Vote for class j

    # Choose the class with the maximum votes for each sample
    predicted_labels = np.argmax(votes, axis=1)

    return predicted_labels
```

Appendix C

Here is the entire genHelpers.py file:

```
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
from scipy.special import expit

#calculates the error for each class i.e. how many 7's did you get wrong?
etc.
def classwise_error_rate(predictions, true_labels, num_classes=10):
    # Ensure predictions and true_labels are numpy arrays and are 1-
dimensional
    predictions = np.asarray(predictions).flatten()
    true_labels = np.asarray(true_labels).flatten()

    error_rates = {}

    for cls in range(num_classes):
        # Get indices where the true label is the current class
        class_indices = (true_labels == cls)

        # Use the boolean mask directly on the 1D arrays
        class_predictions = predictions[class_indices]
        class_true_labels = true_labels[class_indices]

        # Calculate error rate for the current class
        error_rate = np.mean(class_predictions != class_true_labels)

        # Store the error rate for this class formatted as a percentage
        error_rates[cls] = f"{error_rate:.2%}"

    return error_rates

def error_rate(predictions, true_labels):
    return np.mean(predictions != true_labels)


import numpy as np


# Define activation functions
def identity(x):
    return x


def sigmoid(x):
    return expit(x)


def sinusoidal(x):
    return np.sin(x)


def relu(x):
    return np.maximum(x, 0)
```

```
# Randomized feature mapping function

def randomized_feature_mapping(X, L, g=np.identity):
    """
    Generate a random feature mapping for the input data X using the function
g.

    Parameters:
    - X: numpy array of shape (n_samples, n_features), input data
    - L: int, number of random features to generate
    - g: function, activation function to use in the feature mapping

    Returns:
    - h_X: numpy array of shape (n_samples, L), transformed feature space
    - W: numpy array of shape (L, n_features), random weight matrix
    - b: numpy array of shape (L,), random bias vector
    """
    n_samples, n_features = X.shape

    # Generate random matrix W and bias vector b
    W = np.random.normal(0, 1, (L, n_features))  # W ~ N(0, 1) with shape (L,
n_features)
    b = np.random.normal(0, 1, L)  # b ~ N(0, 1) with shape (L,)

    # Compute random feature mapping
    h_X = g(np.dot(X, W.T) + b)  # h(X) = g(W * X^T + b)

    return h_X, W, b

def randomized_feature_mapping_with_params(X, W, b, g=np.identity):
    """
    Apply a precomputed random feature mapping to new data X using the given
W and b.

    Parameters:
    - X: numpy array of shape (n_samples, n_features), input data
    - W: numpy array of shape (L, n_features), random weight matrix
    - b: numpy array of shape (L,), random bias vector
    - g: function, activation function to use in the feature mapping

    Returns:
    - h_X: numpy array of shape (n_samples, L), transformed feature space
    """
    return g(np.dot(X, W.T) + b)  # h(X) = g(W * X^T + b)


def plot_confusion_matrix(true_labels, predicted_labels, num_classes):
    """
    Generate and plot a confusion matrix for the given true and predicted
labels,
    including row and column totals.

    Parameters:
    true_labels (numpy.ndarray): The ground truth labels.
    predicted_labels (numpy.ndarray): The predicted labels.
```

```
    num_classes (int): The number of classes in the dataset.
    """
    # Compute the confusion matrix
    cm = confusion_matrix(true_labels, predicted_labels,
labels=np.arange(num_classes))

    # Compute row and column totals
    row_totals = cm.sum(axis=1)
    col_totals = cm.sum(axis=0)
    overall_total = cm.sum()

    # Extend the confusion matrix to include totals
    cm_with_totals = np.zeros((num_classes + 1, num_classes + 1), dtype=int)
    cm_with_totals[:num_classes, :num_classes] = cm
    cm_with_totals[:num_classes, -1] = row_totals  # Add row totals
    cm_with_totals[-1, :num_classes] = col_totals  # Add column totals
    cm_with_totals[-1, -1] = overall_total  # Add overall total

    # Plot the extended confusion matrix as a heatmap
    plt.figure(figsize=(10, 8))
    plt.imshow(cm_with_totals, interpolation='nearest', cmap=plt.cm.Blues)
    plt.title("Confusion Matrix with Totals")
    plt.colorbar()

    # Label the axes
    tick_marks = np.arange(num_classes + 1)
    labels = [str(i) for i in range(num_classes)] + ['Total']
    plt.xticks(tick_marks, labels, rotation=45)
    plt.yticks(tick_marks, labels)

    # Add the numerical values to the confusion matrix cells
    for i in range(num_classes + 1):
        for j in range(num_classes + 1):
            plt.text(j, i, f"{cm_with_totals[i, j]}",
                     horizontalalignment="center",
                     color="white" if cm_with_totals[i, j] >
cm_with_totals.max() / 2 else "black")

    # Add axis labels
    plt.ylabel("True Labels")
    plt.xlabel("Predicted Labels")
    plt.tight_layout()
    plt.show()
```