project

November 20, 2023

[1]:
```python
import scipy
import numpy as np
from tqdm.notebook import tqdm
from numba import njit

def np_array(shape):
    return np.array(shape, dtype=np.float32)

def np_zeros(shape):
    return np.zeros(shape, dtype=np.float32)

def np_ones(shape):
    return np.ones(shape, dtype=np.float32)
```

[2]:
```python
data = scipy.io.loadmat('mnist.mat')
```

[3]:
```python
testX, testY, trainX, trainY = data['testX'], data['testY'], data['trainX'],␣
↪data['trainY']
# Normalize data
testX = (testX / 255.0).astype(np.float32)
trainX = (trainX / 255.0).astype(np.float32)
trainY = (trainY.T).astype(np.float32)
testY = (testY.T).astype(np.float32)
```

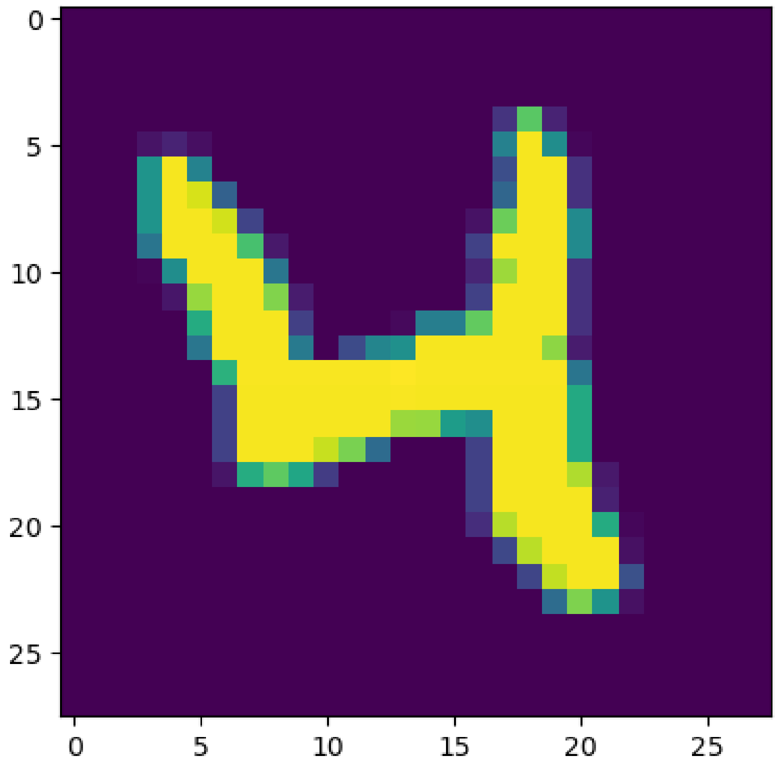
[4]:
```python
print(f"{testX.shape=}")
print(f"{testY.shape=}")
print(f"{trainX.shape=}")
print(f"{trainY.shape=}")
```

```
testX.shape=(10000, 784)
testY.shape=(10000, 1)
trainX.shape=(60000, 784)
trainY.shape=(60000, 1)
```

[5]:
```python
%matplotlib inline
from matplotlib import pyplot as plt
def display_image(normalized_data):
    # data should contain values in [0,1]
```

```
    scaled = normalized_data.reshape((28,-1)) * 255
    plt.imshow(scaled, interpolation='nearest')
    plt.show()
```

[6]:
```
i = 20
display_image(trainX[i]), trainY[i]
```



[6]: (None, array([4.], dtype=float32))

1 Part 1

1.1 One-Versus-All Classifier

We want to solve the problem

$$\min_x ||y - Ax||$$

where A is our training data and x is a set of weights for the data.

From the work shown in class, we show that the following must be true for a least squares solution x

$$A^T y = A^T A x$$

where B is the matrix of basis vectors of the subspace given by A.

Then, we can solve for x, which must exist.

$$x = (A^T A)^{-1} A^T y$$

First, we want to compare class i against all classes $\neq i$. To do this, we create a new y marked either 1 or -1 depending on whether $y_i = i$ or $y_i \neq i$.

[7]:
```
# Add column of ones for bias
def create_one_v_all_classifier(trainX, trainY, n):
    A = np.column_stack((trainX, np_ones(trainX.shape[0]))).astype(np.float32)
    y = np.copy(trainY)
    mask1, mask2 = trainY == n, trainY != n
    y[mask1] = 1.0
    y[mask2] = -1.0
    return A, y
```

Now we use the pseudoinverse function, which solves least-squares.

[8]:
```
@njit
def solve_one_v_all_classifier(A, y):
    A = A.astype(np.float32)
    y = y.astype(np.float32)
    B = np.linalg.pinv(A) @ y
    return B.astype(np.float32)
```

And let's make some functions that compute and display the confusion matrices of the model.

[9]:
```
@njit
def binary_confusion_matrix(A, y, x):
    m = np.zeros((2,2))
    pred = np.sign(A @ x)
    tp = np.sum((pred == 1) & (y == 1))
    tn = np.sum((pred == -1) & (y == -1))
    fp = np.sum((pred == 1) & (y == -1))
    fn = np.sum((pred == -1) & (y == 1))
    return np.array([
        [tp, tn],
        [fp, fn]
    ])

@njit
def evaluate(X_test, y_test, x, n):
    m = binary_confusion_matrix(X_test, y_test, x)
    display_binary_confusion_matrix(m, n)
```

```
@njit
def display_binary_confusion_matrix(conf_matrix, n, ax):
    error_rate = conf_matrix[1, :].sum() / conf_matrix.sum()
    # fig, ax2 = plt.subplots(figsize=(8, 8))
    cax = ax.matshow(conf_matrix, cmap='Blues')
    # Add colorbar
    fig.colorbar(cax)
    # Set labels for the x and y axes
    ax.xlabel('Predicted')
    ax.ylabel('True')
    # Add a title with the error rate
    ax.title(f'Confusion Matrix for digit {n}\nError Rate: {error_rate:.2%}')
    # Display the values inside the matrix
    for i in range(conf_matrix.shape[0]):
        for j in range(conf_matrix.shape[1]):
            plt.text(j, i, f'{int(conf_matrix[i, j])}', ha='center',␣
 ↪va='center', color='black')

    plt.show()
```

Now let's see what the results are for each class $0, \ldots, 9$

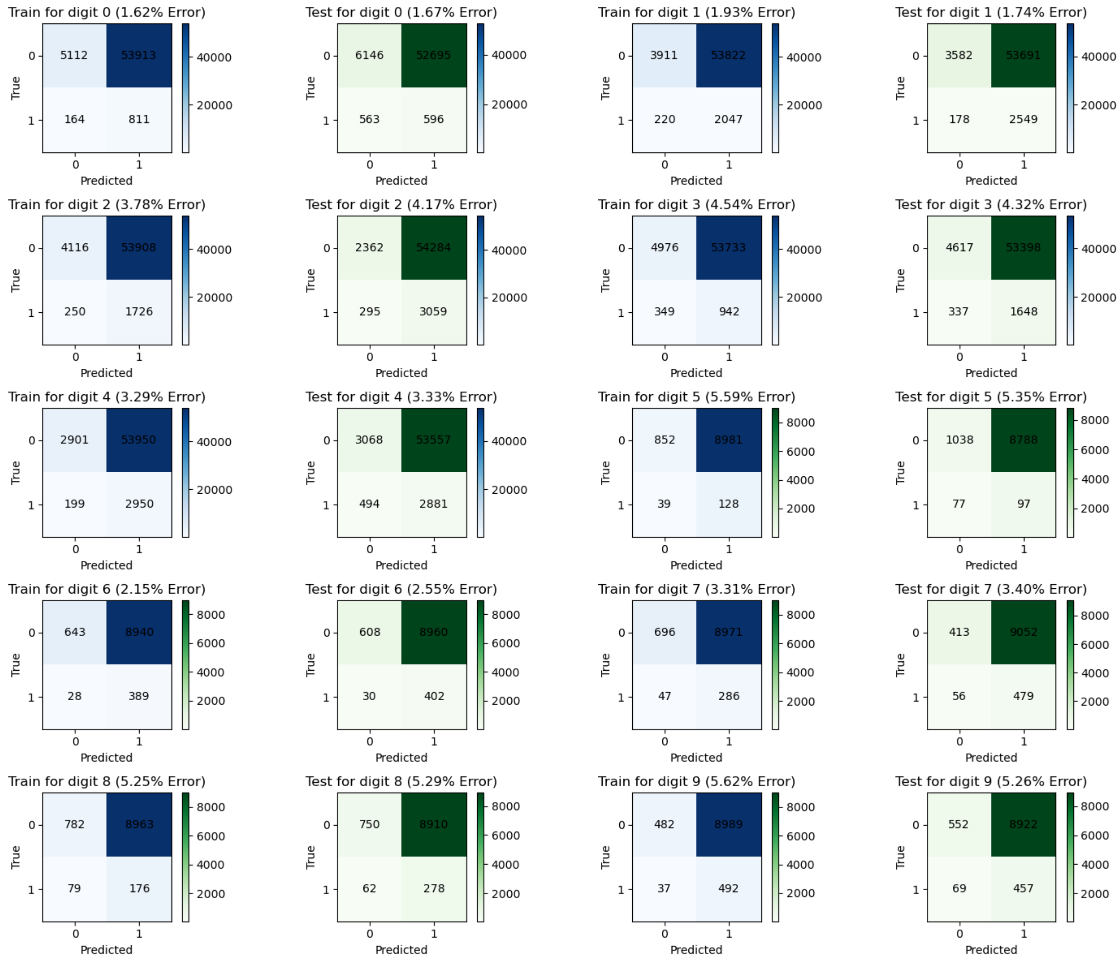
[10]:
```
m_trains, m_tests = [], []
one_v_all_models = []
for n in tqdm(range(10)):
    A, y = create_one_v_all_classifier(trainX, trainY, n)
    x = solve_one_v_all_classifier(A, y)
    one_v_all_models.append(x) # save for later use
    m_train = binary_confusion_matrix(A, y, x)
    m_trains.append(m_train)
    A_test, y_test = create_one_v_all_classifier(testX, testY, n)
    m_test = binary_confusion_matrix(A_test, y_test, x)
    m_tests.append(m_test)
one_v_all_models = np.array(one_v_all_models).reshape((10, -1))
```

```
  0%|          | 0/10 [00:00<?, ?it/s]
```

[11]:
```
# source: ChatGPT
# Create a figure with 4 rows and 5 columns of subplots
def create_binary_confusion_matrix_plot(m_trains, m_tests, start=0):
    fig, axs = plt.subplots(5, 4, figsize=(15, 12))

    # Flatten the axs array to make it easier to iterate
    axs_flat = axs.flatten()

    # Iterate through the confusion matrices and plot them
    for i, (m_train, m_test) in enumerate(zip(m_trains, m_tests)):
```

```
        # Plot the confusion matrix for training data
        im = axs_flat[2*i].imshow(m_train, cmap='Blues',␣
↪interpolation='nearest')

        # Add numbers to each square
        for x in range(2):
            for y in range(2):
                axs_flat[i].text(y, x, str(m_train[x, y]), color="black",␣
↪ha="center", va="center")

        # Add labels, title, and colorbar
        axs_flat[2*i].set_title(f'Train for digit {i} ({100*m_train[1,:].sum()/
↪m_train.sum():.2f}% Error)')
        axs_flat[2*i].set_xticks(np.arange(2))
        axs_flat[2*i].set_yticks(np.arange(2))
        axs_flat[2*i].set_xticklabels(['0', '1'])
        axs_flat[2*i].set_yticklabels(['0', '1'])
        axs_flat[2*i].set_xlabel('Predicted')
        axs_flat[2*i].set_ylabel('True')
        plt.colorbar(im, ax=axs_flat[i], fraction=0.046, pad=0.04)

        # Plot the confusion matrix for testing data
        im = axs_flat[2*i+1].imshow(m_test, cmap='Greens',␣
↪interpolation='nearest')

        # Add numbers to each square
        for x in range(2):
            for y in range(2):
                axs_flat[i + 10].text(y, x, str(m_test[x, y]), color="black",␣
↪ha="center", va="center")

        # Add labels, title, and colorbar
        axs_flat[2*i+1].set_title(f'Test for digit {i} ({100*m_test[1,:].sum()/
↪m_test.sum():.2f}% Error)')
        axs_flat[2*i+1].set_xticks(np.arange(2))
        axs_flat[2*i+1].set_yticks(np.arange(2))
        axs_flat[2*i+1].set_xticklabels(['0', '1'])
        axs_flat[2*i+1].set_yticklabels(['0', '1'])
        axs_flat[2*i+1].set_xlabel('Predicted')
        axs_flat[2*i+1].set_ylabel('True')
        plt.colorbar(im, ax=axs_flat[i + 10], fraction=0.046, pad=0.04)

    # Adjust layout for better spacing
    plt.tight_layout()
    plt.show()
```

```
create_binary_confusion_matrix_plot(m_trains, m_tests)
```



From the data, we see that digits 9, 8, 5, and 2 are the hardest to recognize.

Using the one-v-all classifier, we can build a multiclass classifier given by

$$\hat{f}(\mathbf{x}) = \arg\max_{k=0,\ldots 9} g_k(\mathbf{x})$$

[12]:
```
@njit
def multiclass_predict(models, x) -> int:
    preds = models @ x
    return int(np.argmax(preds))
```

[13]:
```
conf_matrix_ova = np.zeros((10, 10))
testX_ones = np.c_[testX, np.ones(testX.shape[0])].astype(np.float32)
n = 0
for row, actual in zip(testX_ones, testY):
```

```
    pred = multiclass_predict(one_v_all_models, row)
    actual = int(actual)
    conf_matrix_ova[actual][pred] += 1
```

/var/folders/qs/qtpxyfmd5bv20vyzlfl11ncc0000gn/T/ipykernel_3616/274643382.py:3:
NumbaPerformanceWarning: **'@' is faster on contiguous arrays, called on**

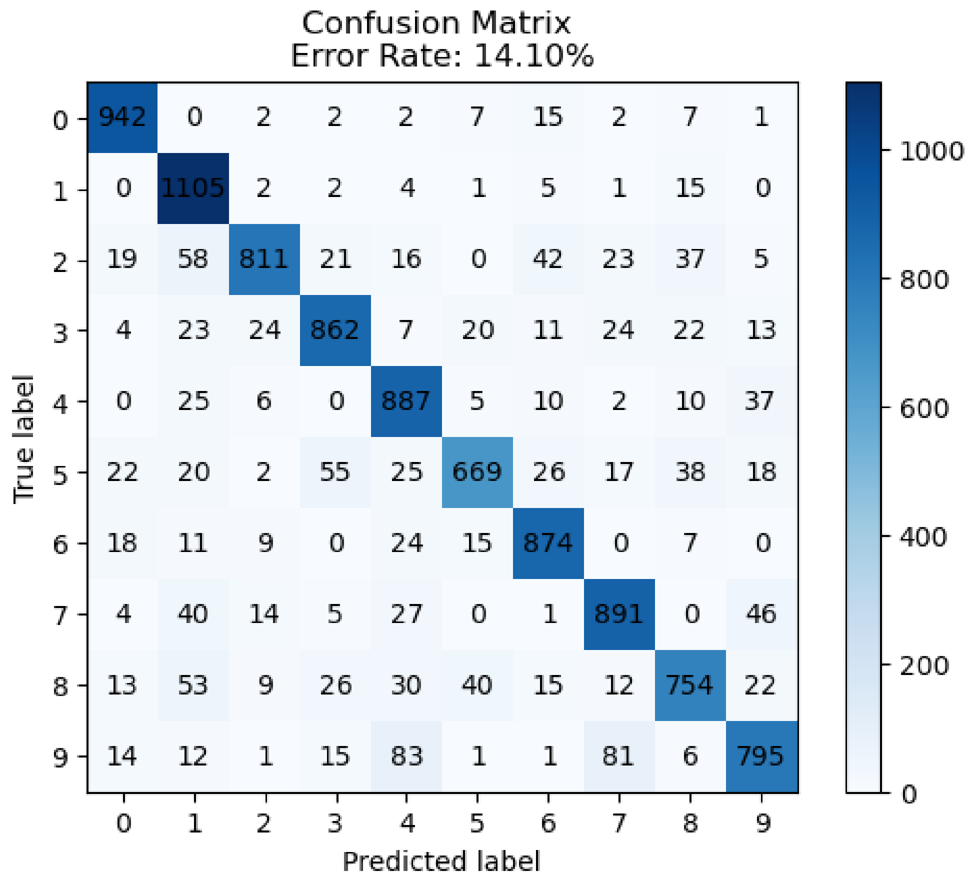
(Array(float32, 2, 'C', False, aligned=True), Array(float32, 1, 'A', False,

aligned=True))
  preds = models @ x

[14]:
```
def plot_multiclass_confusion_matrix(confusion_matrix, title=''):
    """
    Plot a multiclass confusion matrix with row and column totals.

    Parameters:
    - confusion_matrix: numpy array, the confusion matrix to be plotted

    Returns:
    None
    """
    # Calculate row and column totals
    row_totals = np.sum(confusion_matrix, axis=1)
    col_totals = np.sum(confusion_matrix, axis=0)

    # Calculate the error rate
    total_samples = np.sum(confusion_matrix)
    correct_predictions = np.trace(confusion_matrix)
    error_rate = 1 - (correct_predictions / total_samples)

    # Create a figure and axes
    fig, ax = plt.subplots()

    # Plot the confusion matrix using imshow
    cax = ax.imshow(confusion_matrix, cmap='Blues', interpolation='nearest')

    # Add text annotations for each element in the matrix
    for i in range(confusion_matrix.shape[0]):
        for j in range(confusion_matrix.shape[1]):
            ax.text(j, i, int(confusion_matrix[i, j]), va='center', ha='center')

    # Set labels for rows and columns
    ax.set_xticks(np.arange(confusion_matrix.shape[1]))
    ax.set_yticks(np.arange(confusion_matrix.shape[0]))
    ax.set_xticklabels(np.arange(confusion_matrix.shape[1]))
    ax.set_yticklabels(np.arange(confusion_matrix.shape[0]))
```

```
    # Set labels and title
    plt.xlabel('Predicted label')
    plt.ylabel('True label')
    plt.title(f'Confusion Matrix {title}\nError Rate: {error_rate:.2%}')

    # Add colorbar
    plt.colorbar(cax)

    # Display the plot
    plt.show()
```

[15]:
```
plot_multiclass_confusion_matrix(conf_matrix_ova)
```



1.2 One-vs-One Classifiers

Now let's create a multiclass classifier built out of 1v1 classifiers. These will compare classes i and j.

$$\hat{f}(\mathbf{x}) = \text{sign}(\ \mathbf{x} + \alpha) = \begin{cases} 1 & \text{if label} = i \\ -1 & \text{if label} = j \end{cases}$$

To do this, we need to filter out only the data that belongs to classes i and j and train the model on this only.

[16]:
```
def create_1v1_classifier(X, y, i, j):
    # Return matrix A such that only digits with labels with i or j are included
    # Return matrix y such that only labels with i or j are included
    mask = (y == i) | (y == j)
    big_mask = np.tile(mask, (1, X.shape[1]))
    A = X[big_mask].reshape((-1, X.shape[1]))
    A = np.column_stack((A, np.ones(A.shape[0]))) # add ones
    # Select only relevant labels
    _y = y[mask]
    # Re-label
    i_mask = _y == i
    j_mask = _y == j
    _y[i_mask] = 1
    _y[j_mask] = -1
    assert _y.shape[0] == A.shape[0] # should have same number of rows

    return A.astype(np.float32), _y.astype(np.float32)
```

The solution and prediction will be exactly the same.

[17]:
```
@njit
def solve_1v1_classifier(A, y):
    B = np.dot(np.linalg.pinv(A), y)
    return B.astype(np.float32)

@njit
def predict_1v1(x, dig):
    return np.sign(dig @ x, dtype=np.int8)
```

Now that we can create a classifier for any pair (i, j), we can combine all possible pairs in the class set. Then, the multiclass model will evaluate all of these and count which has the highest "vote" count.

[18]:
```
import itertools
import functools
# All pairs of numbers under 10
pairs = list(itertools.combinations(range(10), 2))

def create_1v1_confusion_matrix(X, y, i, j, x):
    A, _y = create_1v1_classifier(X, y, i, j)
    return binary_confusion_matrix(A, _y, x)
```

```
[19]: def solve_multiclass_classifier(pairs, trainX, trainY):
          gs = [] # list of (i, j, solution)
          for i, j in tqdm(pairs):
              # Parse data for (i, j)
              A, y = create_1v1_classifier(trainX, trainY, i, j)
              # Solve
              x = solve_1v1_classifier(A, y)
              gs.append((i, j, x))
          return gs

      def compute_1v1_confusion_matrices(gs, test_x, test_y):
          # compute confusion matrices for each pair
          return [(i, j, create_1v1_confusion_matrix(test_x, test_y, i, j, x)) for i,␣
       ↪j, x in gs]
```

```
[20]: gs = solve_multiclass_classifier(pairs, trainX, trainY)
```

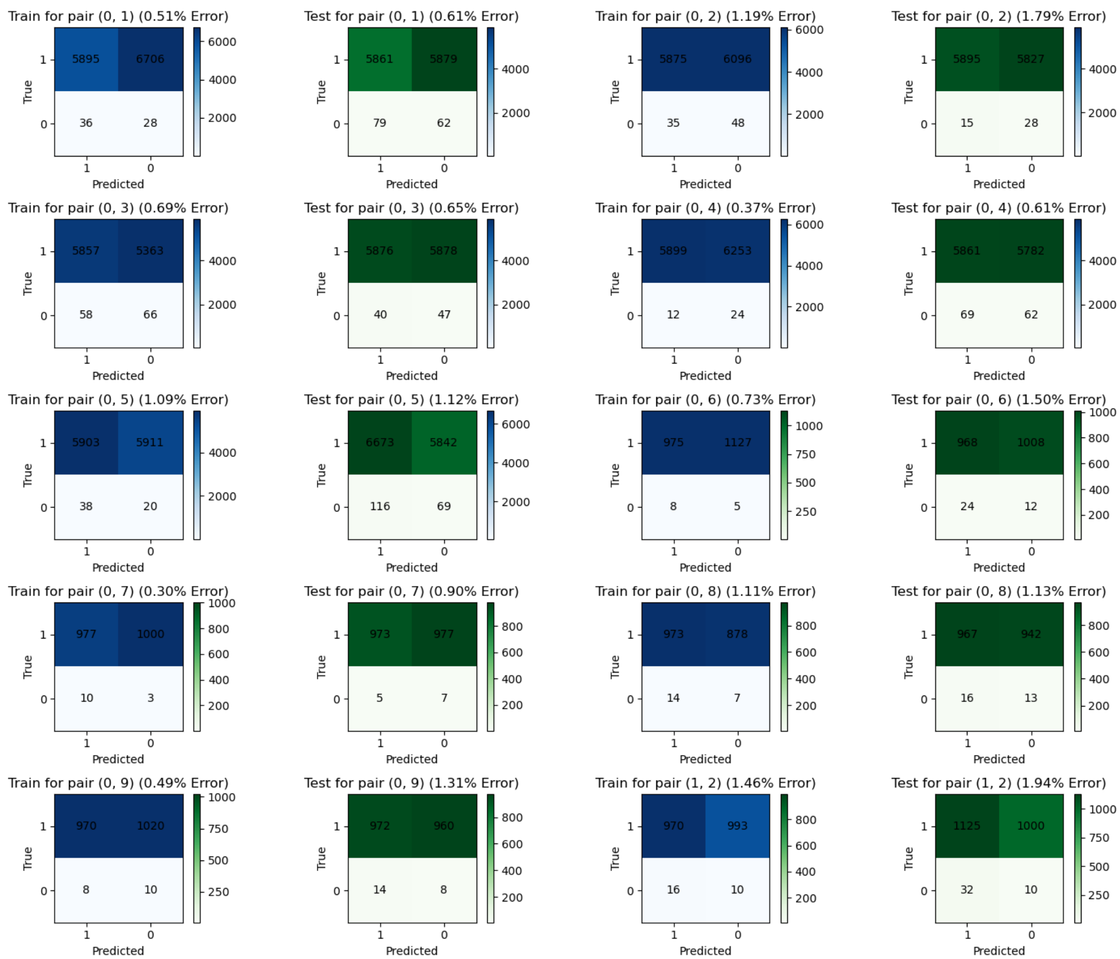
```
  0%|          | 0/45 [00:00<?, ?it/s]
```

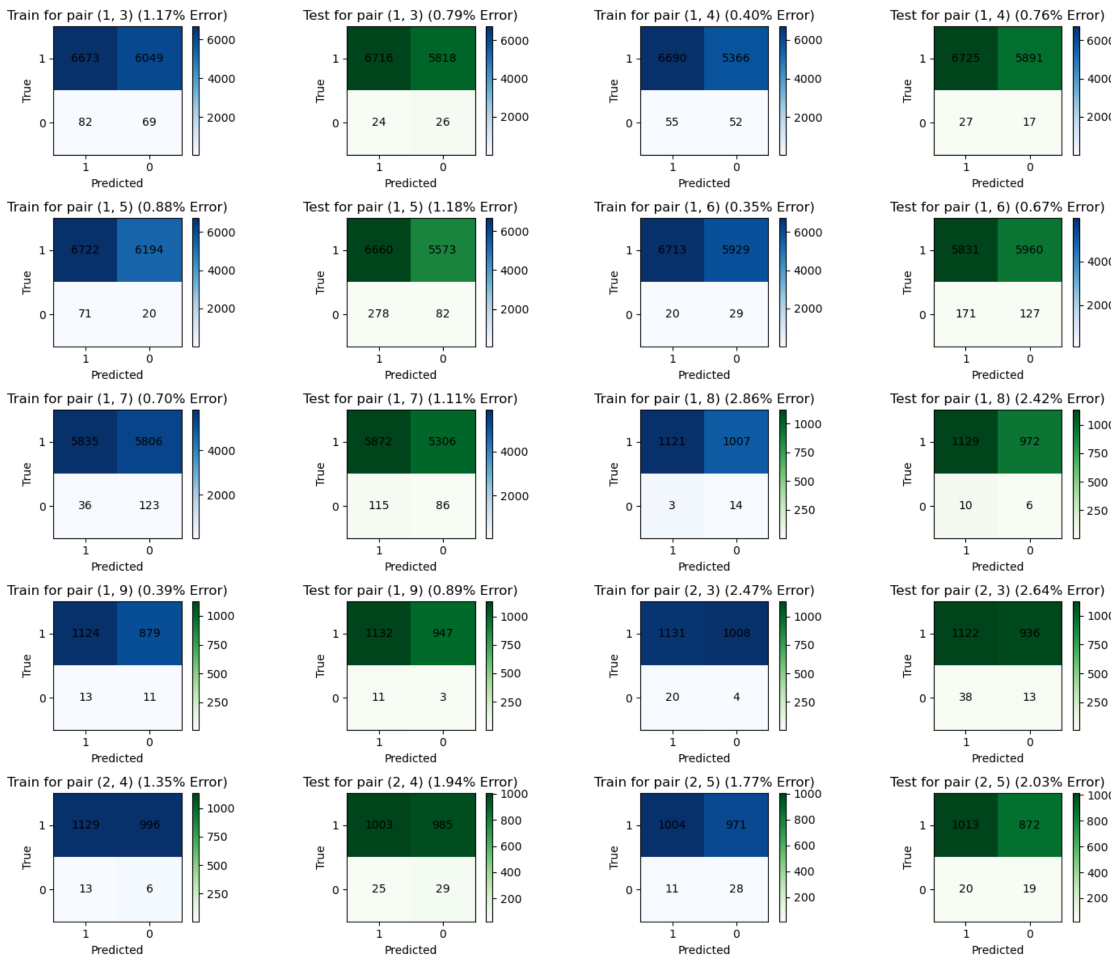
```
[21]: m_train_1v1 = compute_1v1_confusion_matrices(gs, trainX, trainY)
      m_test_1v1 = compute_1v1_confusion_matrices(gs, testX, testY)
```

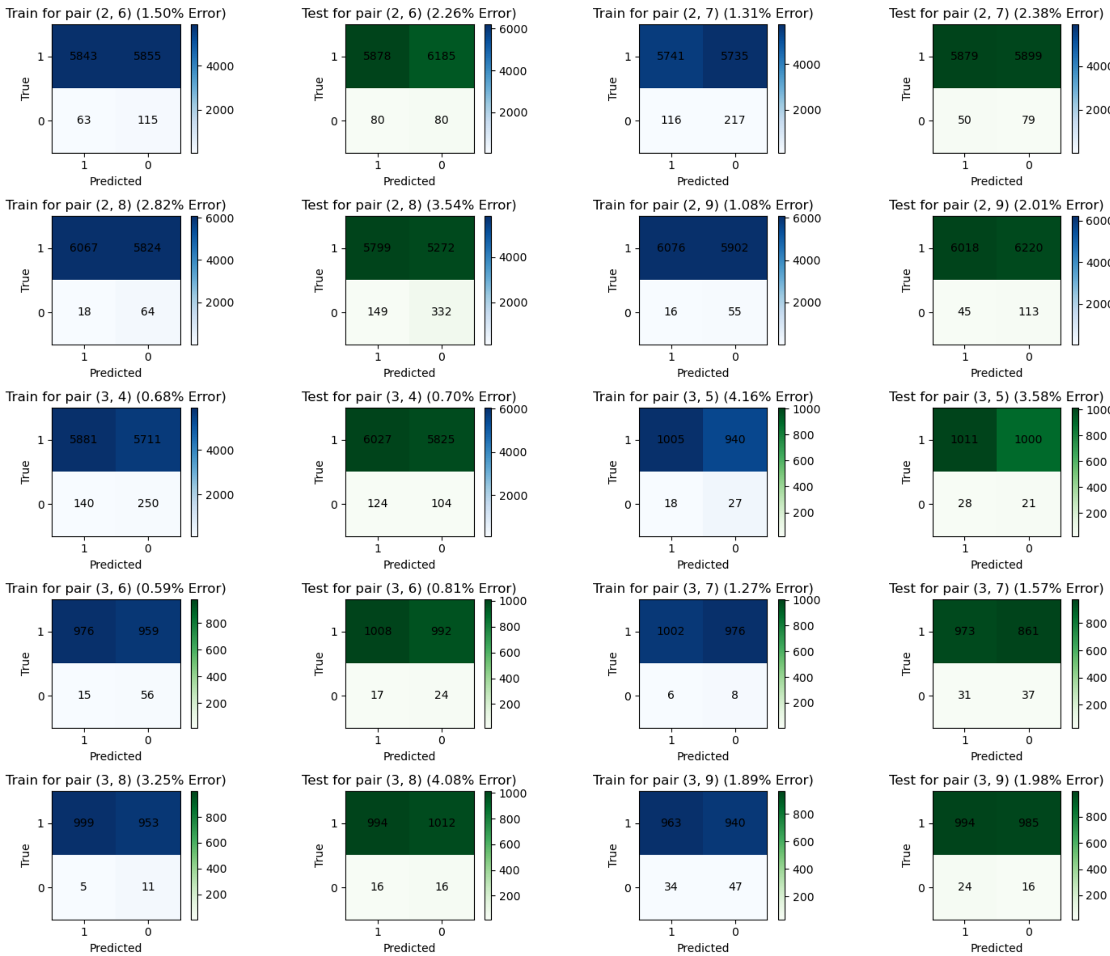
```
[22]: def grouper(iterable, n, *, incomplete='fill', fillvalue=None):
          "Collect data into non-overlapping fixed-length chunks or blocks"
          # grouper('ABCDEFG', 3, fillvalue='x') --> ABC DEF Gxx
          # grouper('ABCDEFG', 3, incomplete='strict') --> ABC DEF ValueError
          # grouper('ABCDEFG', 3, incomplete='ignore') --> ABC DEF
          args = [iter(iterable)] * n
          if incomplete == 'fill':
              return itertools.zip_longest(*args, fillvalue=fillvalue)
          if incomplete == 'strict':
              return zip(*args, strict=True)
          if incomplete == 'ignore':
              return zip(*args)
          else:
              raise ValueError('Expected fill, strict, or ignore')


      def create_binary_confusion_matrix_plot(m_trains, m_tests):
          pairs = [(i, j) for i, j, x in m_trains]
          m_trains = [x for _, _, x in m_trains]
          m_tests = [x for _, _, x in m_tests]
          assert len(m_trains) == len(m_tests)
          fig, axs = plt.subplots(5, 4, figsize=(15, 12))

          # Flatten the axs array to make it easier to iterate
          axs_flat = axs.flatten()
```

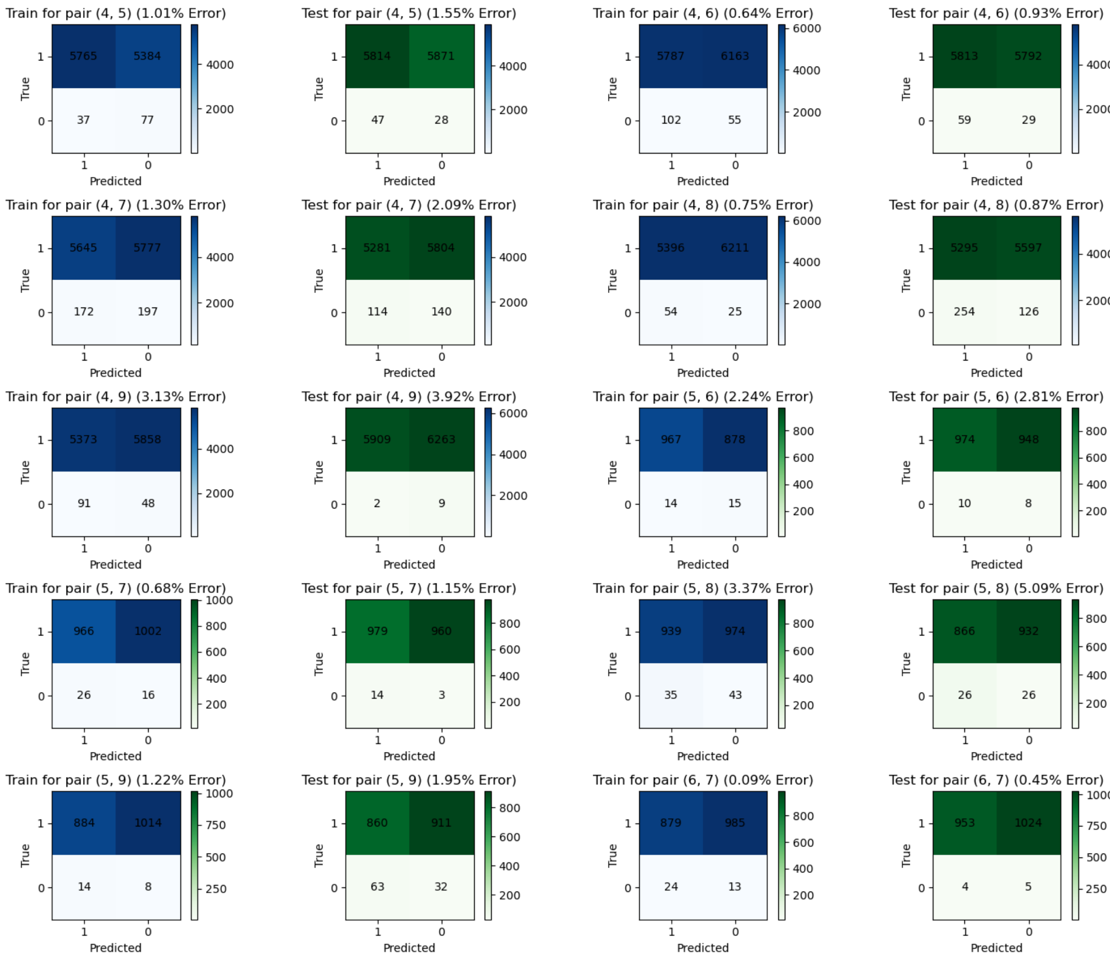
```
    # Iterate through the confusion matrices and plot them
    for i, (m_train, m_test) in enumerate(zip(m_trains, m_tests)):
        j, k = pairs[i]
        # Plot the confusion matrix for training data
        im = axs_flat[2*i].imshow(m_train, cmap='Blues',␣
↪interpolation='nearest')

        # Add numbers to each square
        for x in range(2):
            for y in range(2):
                axs_flat[i].text(y, x, str(m_train[x, y]), color="black",␣
↪ha="center", va="center")

        # Add labels, title, and colorbar
        axs_flat[2*i].set_title(f'Train for pair ({j}, {k}) ({100*m_train[1,:].
↪sum()/m_train.sum():.2f}% Error)')
        axs_flat[2*i].set_xticks(np.arange(2))
        axs_flat[2*i].set_yticks(np.arange(2))
        axs_flat[2*i].set_xticklabels(['1', '0'])
        axs_flat[2*i].set_yticklabels(['1', '0'])
        axs_flat[2*i].set_xlabel('Predicted')
        axs_flat[2*i].set_ylabel('True')
        plt.colorbar(im, ax=axs_flat[i], fraction=0.046, pad=0.04)

        # Plot the confusion matrix for testing data
        im = axs_flat[2*i+1].imshow(m_test, cmap='Greens',␣
↪interpolation='nearest')

        # Add numbers to each square
        for x in range(2):
            for y in range(2):
                axs_flat[i + 10].text(y, x, str(m_test[x, y]), color="black",␣
↪ha="center", va="center")

        # Add labels, title, and colorbar
        axs_flat[2*i+1].set_title(f'Test for pair ({j}, {k}) ({100*m_test[1,:].
↪sum()/m_test.sum():.2f}% Error)')
        axs_flat[2*i+1].set_xticks(np.arange(2))
        axs_flat[2*i+1].set_yticks(np.arange(2))
        axs_flat[2*i+1].set_xticklabels(['1', '0'])
        axs_flat[2*i+1].set_yticklabels(['1', '0'])
        axs_flat[2*i+1].set_xlabel('Predicted')
        axs_flat[2*i+1].set_ylabel('True')
        plt.colorbar(im, ax=axs_flat[i + 10], fraction=0.046, pad=0.04)

    # Adjust layout for better spacing
```

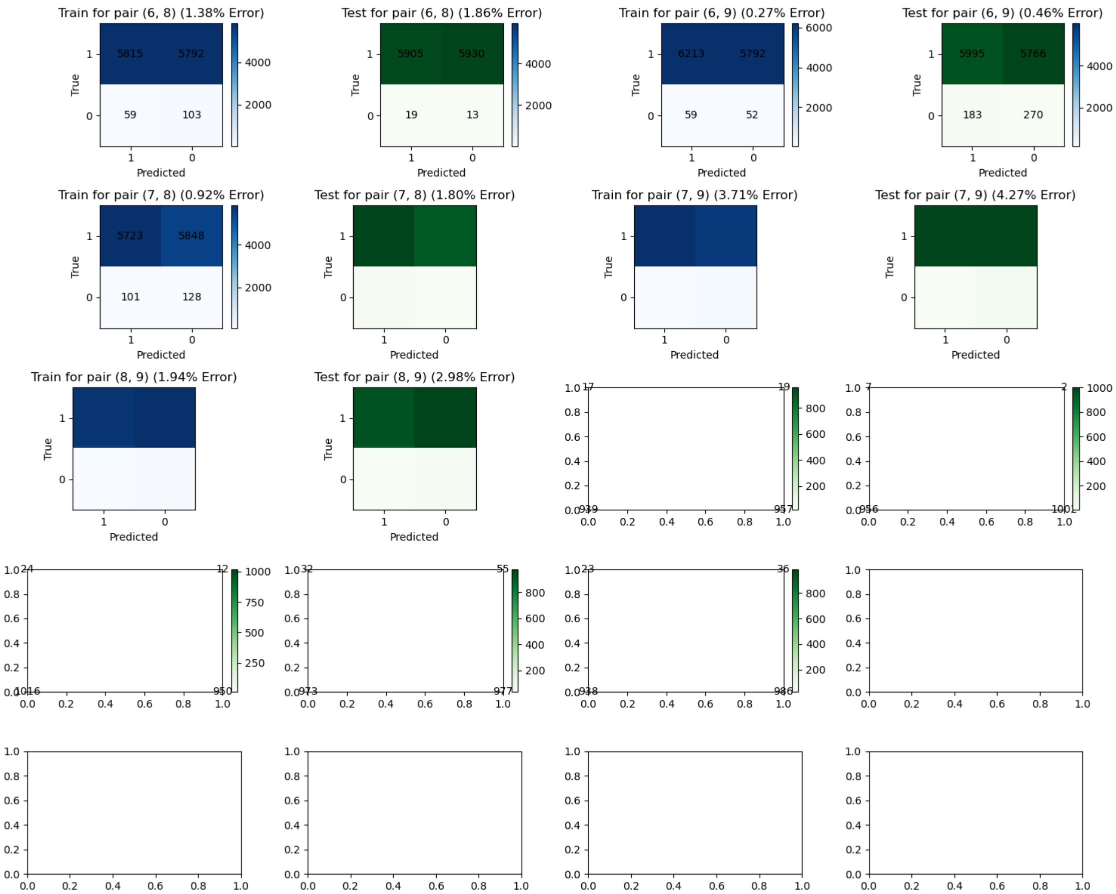
```
    plt.tight_layout()
    plt.show()


def filter_none(it):
    return [x for x in it if x is not None]

train_batches = grouper(m_train_1v1, 10)
test_batches = grouper(m_test_1v1, 10)

for b1, b2 in zip(train_batches, test_batches):
    create_binary_confusion_matrix_plot(filter_none(b1), filter_none(b2))
```




Train for pair (1, 3) (1.17% Error)
6673 6049 82 69
Test for pair (1, 3) (0.79% Error)
6716 5818 24 26
Train for pair (1, 4) (0.40% Error)
6690 5366 55 52
Test for pair (1, 4) (0.76% Error)
6725 5891 27 17
Train for pair (1, 5) (0.88% Error)
6722 6194 71 20
Test for pair (1, 5) (1.18% Error)
6660 5573 278 82
Train for pair (1, 6) (0.35% Error)
6713 5929 20 29
Test for pair (1, 6) (0.67% Error)
5831 5960 171 127
Train for pair (1, 7) (0.70% Error)
5835 5806 36 123
Test for pair (1, 7) (1.11% Error)
5872 5306 115 86
Train for pair (1, 8) (2.86% Error)
1121 1007 3 14
Test for pair (1, 8) (2.42% Error)
1129 972 10 6
Train for pair (1, 9) (0.39% Error)
1124 879 13 11
Test for pair (1, 9) (0.89% Error)
1132 947 11 3
Train for pair (2, 3) (2.47% Error)
1131 1008 20 4
Test for pair (2, 3) (2.64% Error)
1122 936 38 13
Train for pair (2, 4) (1.35% Error)
1129 996 13 6
Test for pair (2, 4) (1.94% Error)
1003 985 25 29
Train for pair (2, 5) (1.77% Error)
1004 971 11 28
Test for pair (2, 5) (2.03% Error)
1013 872 20 19
True
Predicted


Train for pair (2, 6) (1.50% Error)
5843 5855 63 115
Test for pair (2, 6) (2.26% Error)
5878 6185 80 80
Train for pair (2, 7) (1.31% Error)
5741 5735 116 217
Test for pair (2, 7) (2.38% Error)
5879 5899 50 79
Train for pair (2, 8) (2.82% Error)
6067 5824 18 64
Test for pair (2, 8) (3.54% Error)
5799 5272 149 332
Train for pair (2, 9) (1.08% Error)
6076 5902 16 55
Test for pair (2, 9) (2.01% Error)
6018 6220 45 113
Train for pair (3, 4) (0.68% Error)
5881 5711 140 250
Test for pair (3, 4) (0.70% Error)
6027 5825 124 104
Train for pair (3, 5) (4.16% Error)
1005 940 18 27
Test for pair (3, 5) (3.58% Error)
1011 1000 28 21
Train for pair (3, 6) (0.59% Error)
976 959 15 56
Test for pair (3, 6) (0.81% Error)
1008 992 17 24
Train for pair (3, 7) (1.27% Error)
1002 976 6 8
Test for pair (3, 7) (1.57% Error)
973 861 31 37
Train for pair (3, 8) (3.25% Error)
999 953 5 11
Test for pair (3, 8) (4.08% Error)
994 1012 16 16
Train for pair (3, 9) (1.89% Error)
963 940 34 47
Test for pair (3, 9) (1.98% Error)
994 985 24 16
True
Predicted


Train for pair (4, 5) (1.01% Error)
5765 5384 37 77
Test for pair (4, 5) (1.55% Error)
5814 5871 47 28
Train for pair (4, 6) (0.64% Error)
5787 6163 102 55
Test for pair (4, 6) (0.93% Error)
5813 5792 59 29
Train for pair (4, 7) (1.30% Error)
5645 5777 172 197
Test for pair (4, 7) (2.09% Error)
5281 5804 114 140
Train for pair (4, 8) (0.75% Error)
5396 6211 54 25
Test for pair (4, 8) (0.87% Error)
5295 5597 254 126
Train for pair (4, 9) (3.13% Error)
5373 5858 91 48
Test for pair (4, 9) (3.92% Error)
5909 6263 2 9
Train for pair (5, 6) (2.24% Error)
967 878 14 15
Test for pair (5, 6) (2.81% Error)
974 948 10 8
Train for pair (5, 7) (0.68% Error)
966 1002 26 16
Test for pair (5, 7) (1.15% Error)
979 960 14 3
Train for pair (5, 8) (3.37% Error)
939 974 35 43
Test for pair (5, 8) (5.09% Error)
866 932 26 26
Train for pair (5, 9) (1.22% Error)
884 1014 14 8
Test for pair (5, 9) (1.95% Error)
860 911 63 32
Train for pair (6, 7) (0.09% Error)
879 985 24 13
Test for pair (6, 7) (0.45% Error)
953 1024 4 5
True
Predicted



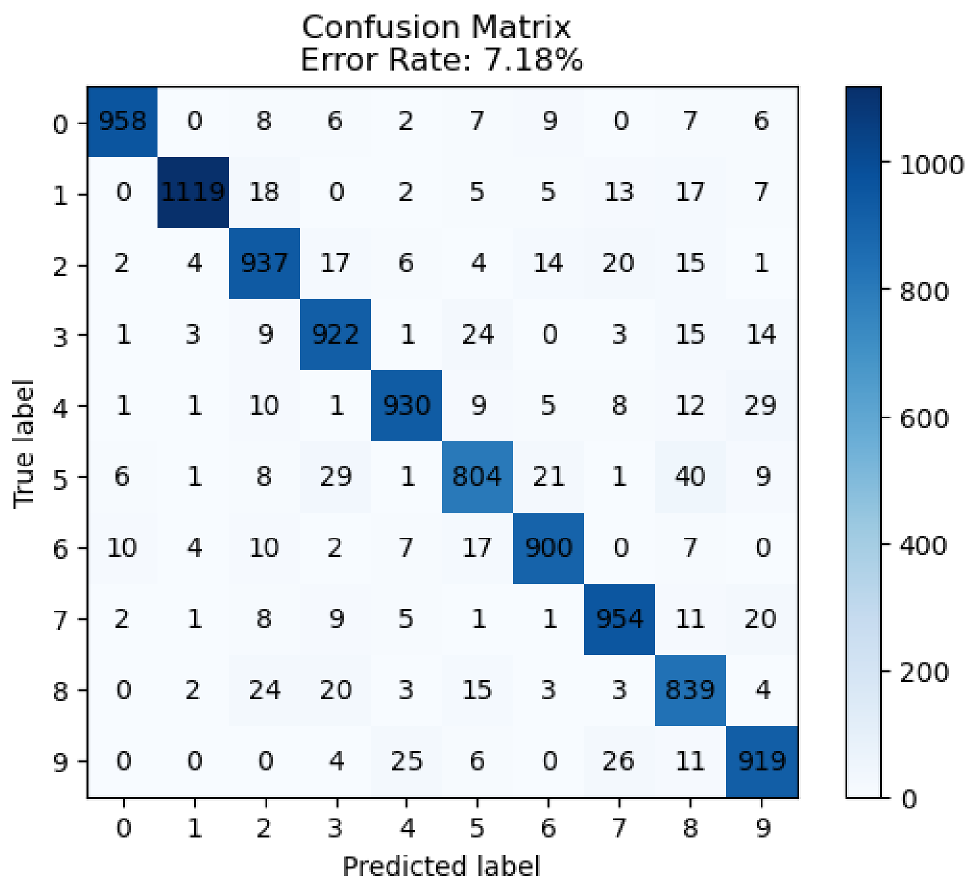
[23]:
```python
def predict_multiclass(gs, dig):
    scores = np.zeros(10)
    for i, j, x in gs:
        dig = dig.astype(np.float32)
        result = np.sign(dig @ x)
        if result == 1:
            scores[i] += 1
        else:
            scores[j] += 1
    return np.argmax(scores)
```

[24]:
```python
def create_multiclass_confusion_matrix(gs, X, y, progress=False):
    if not progress:
        tqdm = lambda x: x
    matrix = np.zeros((10, 10))
    for feature, actual in zip(tqdm(X), y):
        pred = predict_multiclass(gs, np.append(feature, 1))
        # Set the rows as the predicted value and the column as the actual value
        matrix[int(pred)][int(actual)] += 1
```

```
    return matrix
```

[25]:
```
mult_m = create_multiclass_confusion_matrix(gs, testX, testY)
```

[26]:
```
plot_multiclass_confusion_matrix(mult_m)
```



	0	1	2	3	4	5	6	7	8	9
0	958	0	8	6	2	7	9	0	7	6
1	0	1119	18	0	2	5	5	13	17	7
2	2	4	937	17	6	4	14	20	15	1
3	1	3	9	922	1	24	0	3	15	14
4	1	1	10	1	930	9	5	8	12	29
5	6	1	8	29	1	804	21	1	40	9
6	10	4	10	2	7	17	900	0	7	0
7	2	1	8	9	5	1	1	954	11	20
8	0	2	24	20	3	15	3	3	839	4
9	0	0	0	4	25	6	0	26	11	919

1.3 Task 3

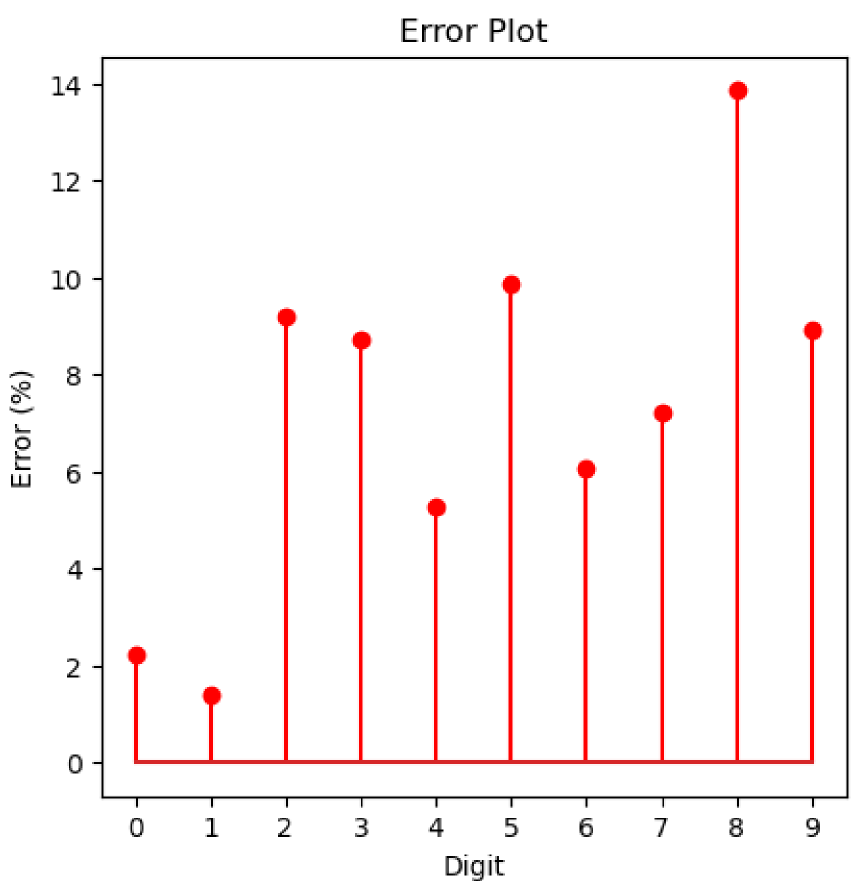
The 1v1 classifiers have a much lower error rate than the multiclass classfier, which has an error of 7.1%. They generalize well on the test data, offering a similar error rate as the training data. We can calculate the error for class k by doing

$$\frac{\text{number of correctly predicted } k}{\text{total k}}$$

[27]:
```
error = np.zeros(10, dtype=np.float64)
for i in range(10):
    total = np.sum(mult_m[:, i])
    e = (total-mult_m[i][i])/ total
```

```
    error[i] = e
```

[28]:
```
plt.figure(figsize=(5, 5))
plt.stem(range(len(error)), 100.0*error, linefmt='r-', markerfmt='ro')


# Adding labels and title
plt.xlabel('Digit')
plt.ylabel('Error (%)')
plt.title('Error Plot')
plt.xticks(list(range(len(error))))

# Display the plot
plt.show()
print("Error rates, highest to lowest")
for i, e in sorted(enumerate(error), key=lambda x: x[1], reverse=True):
    print(f"{i}: {100*e:.03f}%")
```




```
Error rates, highest to lowest
```

```
8: 13.860%
5: 9.865%
2: 9.205%
9: 8.920%
3: 8.713%
7: 7.198%
6: 6.054%
4: 5.295%
0: 2.245%
1: 1.410%
```

We see that the hardest ones to predict are 8, 5, 2, 9, and 3. The easiest ones are 0 and 1.

1.4 Problem 2

First, let's create a random matrix with normally distributed values.

[29]:
```
L = 1000
j = 28*28
```

[30]:
```
identity = lambda x: x
sigmoid = lambda x: 1.0 / (1 + np.exp(-x))
sin = np.sin
relu = lambda x: np.maximum(x, 0)
```

[31]:
```
funcs = {
    'relu': relu,
    'identity': identity,
    'sigmoid': sigmoid,
    'sin': sin,
}

@njit
def get_feature_params(L, j):
    W = np.random.normal(0, 1, (L, j))
    b = np.random.normal(0, 1, L)
    return W.astype(np.float32), b.astype(np.float32)

@njit
def map_to_feature_space(X, W, b):
    X = X.astype(np.float32)
    Y = X @ W.T + b
    return Y.astype(np.float32)
```

[32]:
```
def gen_features(trainX, testX, L):
    W, b = get_feature_params(L, trainX.shape[1])
    train_feat = map_to_feature_space(trainX, W, b)
    test_feat = map_to_feature_space(testX, W, b)
```

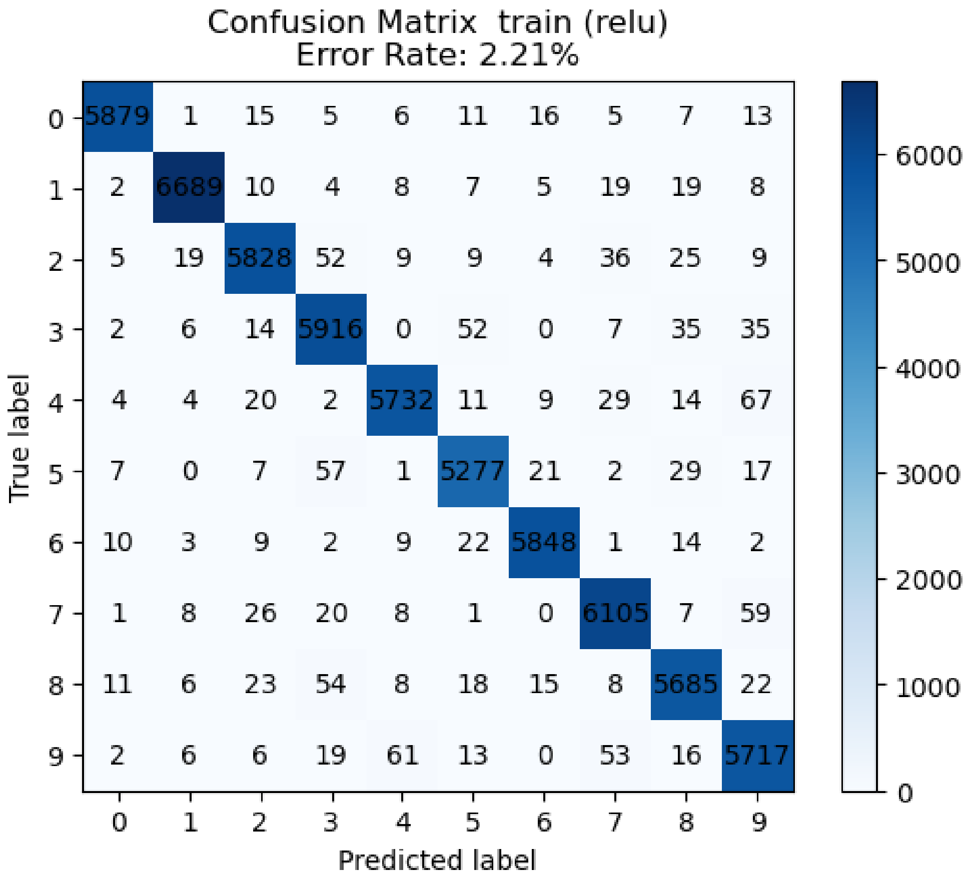
```
    return train_feat, test_feat

train_feat, test_feat = gen_features(trainX, testX, L)
```

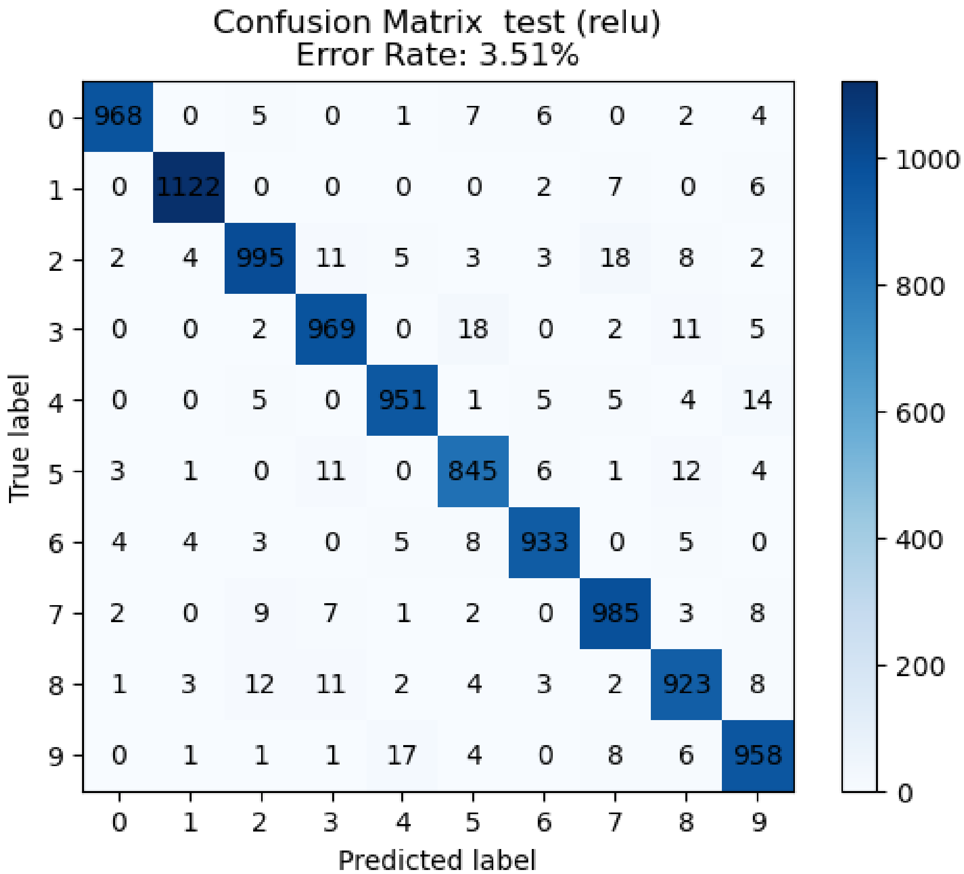
[33]:
```
def solve_for_each_func(pairs, train_feat, trainY, funcs):
    d = {}
    for name, f in tqdm(funcs.items(), desc='Solving'):
        train_feat_g = f(train_feat)
        gs = solve_multiclass_classifier(pairs, train_feat_g, trainY)
        d[name] = gs
    return d

def evaluate_multiclass_feature_map(pairs, train_feat, trainY, test_feat,␣
 ↪testY, solutions):
    for name, sol in solutions.items():
        train_feat_g = funcs[name](train_feat).astype(np.float32)
        test_feat_g = funcs[name](test_feat).astype(np.float32)
        mult_m = create_multiclass_confusion_matrix(sol, train_feat_g, trainY)
        plot_multiclass_confusion_matrix(mult_m, title=f" train ({name})")
        mult_m = create_multiclass_confusion_matrix(sol, test_feat_g, testY)
        plot_multiclass_confusion_matrix(mult_m, title=f" test ({name})")
```

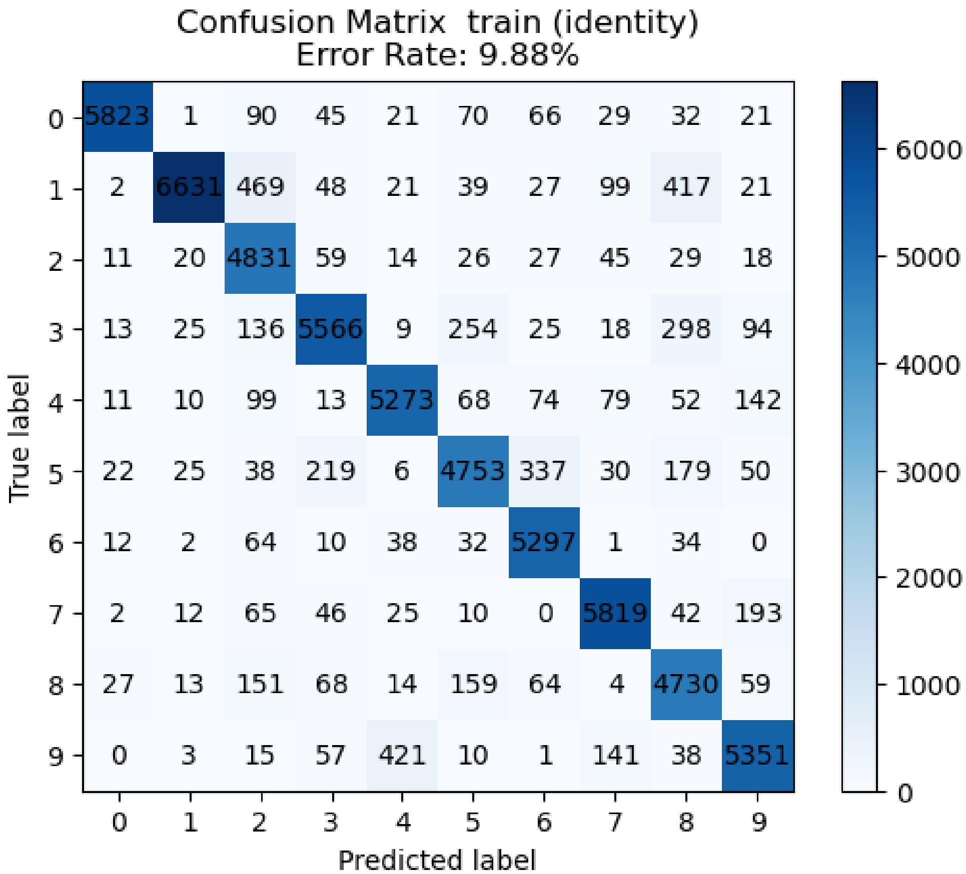
[34]:
```
sols = solve_for_each_func(pairs, train_feat, trainY, funcs)
```

```
Solving:   0%|          | 0/4 [00:00<?, ?it/s]

  0%|          | 0/45 [00:00<?, ?it/s]

  0%|          | 0/45 [00:00<?, ?it/s]

  0%|          | 0/45 [00:00<?, ?it/s]

  0%|          | 0/45 [00:00<?, ?it/s]
```

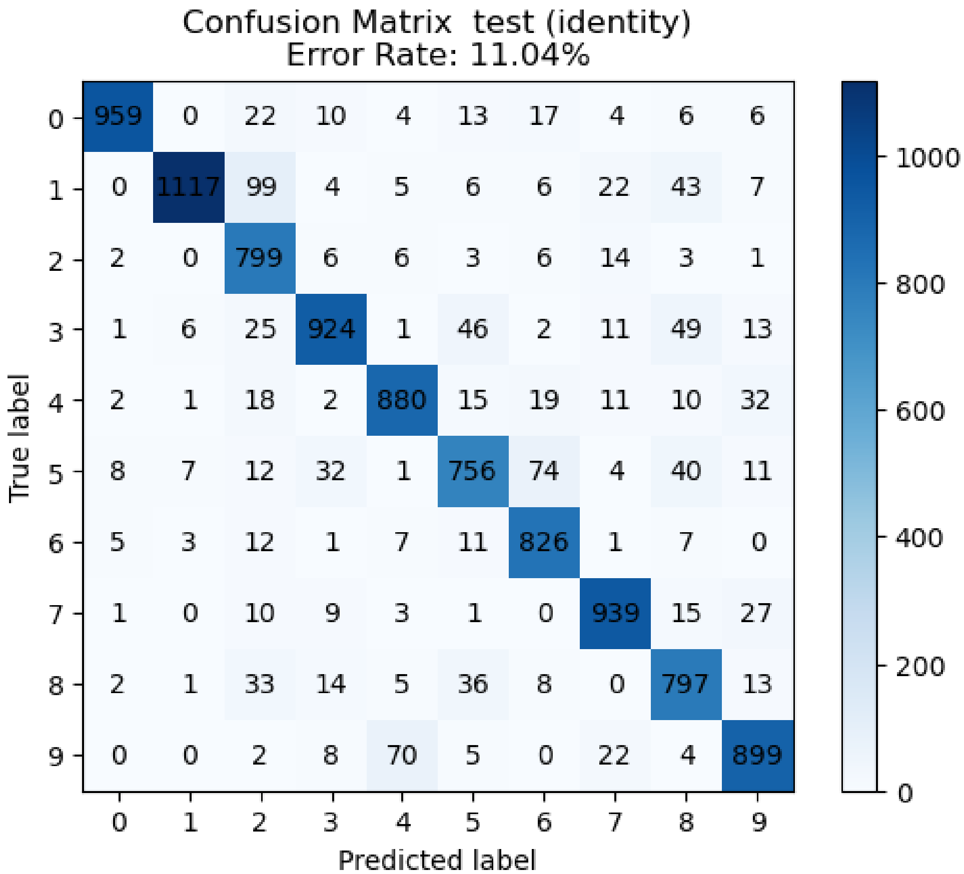
[35]:
```
evaluate_multiclass_feature_map(pairs, train_feat, trainY, test_feat, testY,␣
 ↪sols)
```

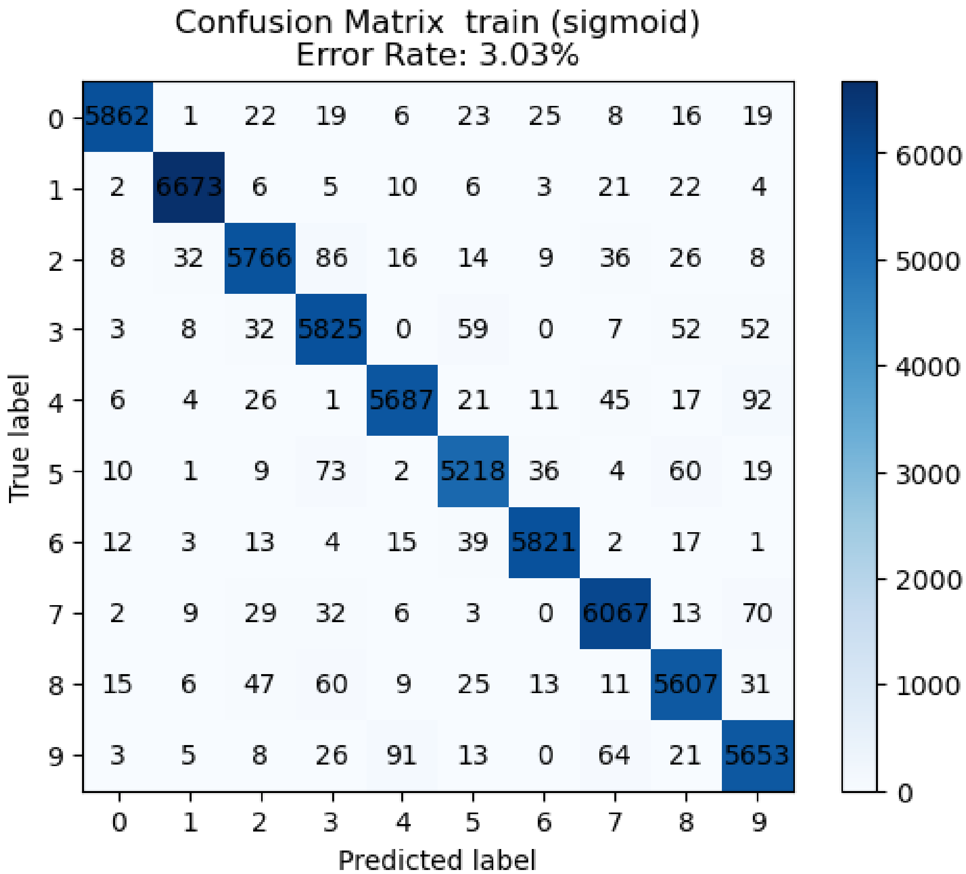

Confusion Matrix train (relu)
Error Rate: 2.21%
True label
Predicted label
0
1
2
3
4
5
6
7
8
9
5879 1 15 5 6 11 16 5 7 13
2 6689 10 4 8 7 5 19 19 8
5 19 5828 52 9 9 4 36 25 9
2 6 14 5916 0 52 0 7 35 35
4 4 20 2 5732 11 9 29 14 67
7 0 7 57 1 5277 21 2 29 17
10 3 9 2 9 22 5848 1 14 2
1 8 26 20 8 1 0 6105 7 59
11 6 23 54 8 18 15 8 5685 22
2 6 6 19 61 13 0 53 16 5717
6000
5000
4000
3000
2000
1000
0

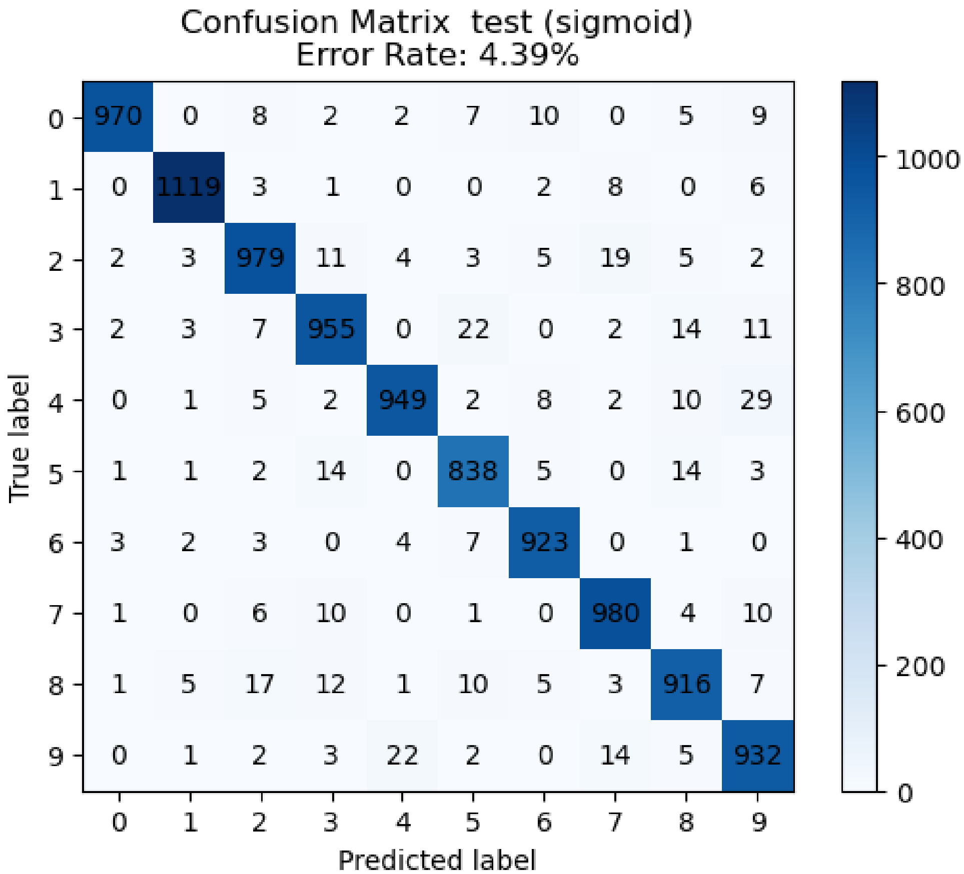

Confusion Matrix test (relu)
Error Rate: 3.51%
True label
Predicted label
0 1 2 3 4 5 6 7 8 9
968 0 5 0 1 7 6 0 2 4
0 1122 0 0 0 0 2 7 0 6
2 4 995 11 5 3 3 18 8 2
0 0 2 969 0 18 0 2 11 5
0 0 5 0 951 1 5 5 4 14
3 1 0 11 0 845 6 1 12 4
4 4 3 0 5 8 933 0 5 0
2 0 9 7 1 2 0 985 3 8
1 3 12 11 2 4 3 2 923 8
0 1 1 1 17 4 0 8 6 958
1000
800
600
400
200
0

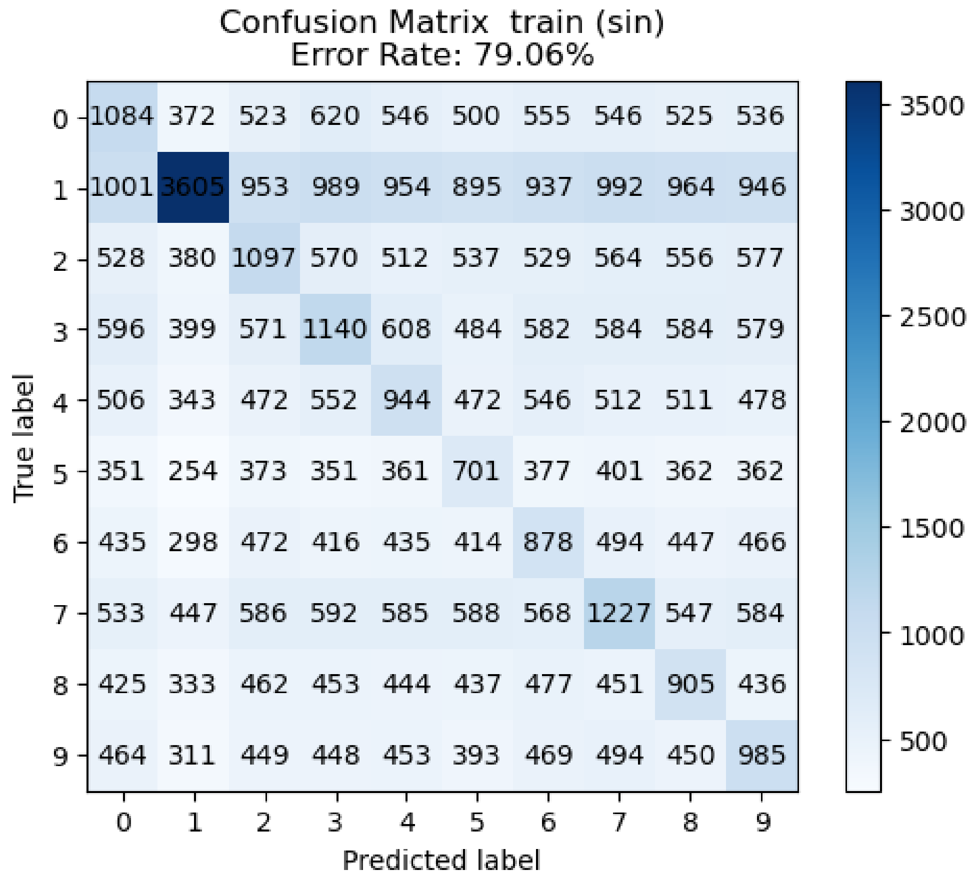


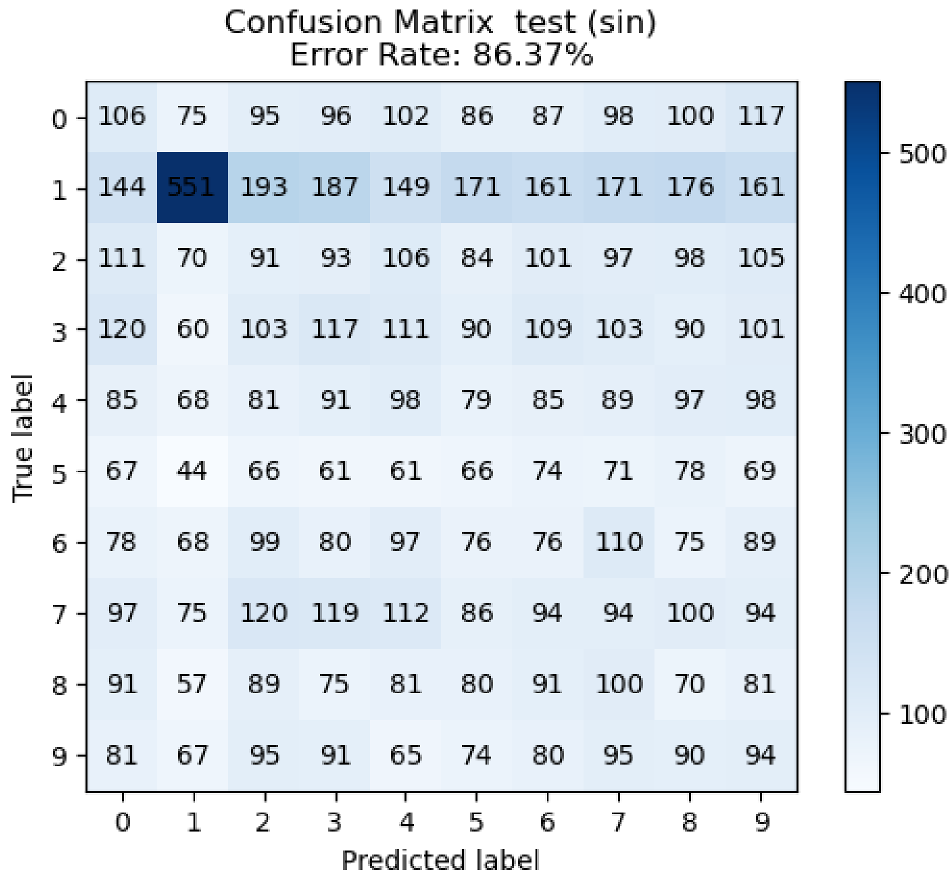
Confusion Matrix train (identity)
Error Rate: 9.88%
True label
Predicted label
0 1 2 3 4 5 6 7 8 9
5823 1 90 45 21 70 66 29 32 21
2 6631 469 48 21 39 27 99 417 21
11 20 4831 59 14 26 27 45 29 18
13 25 136 5566 9 254 25 18 298 94
11 10 99 13 5273 68 74 79 52 142
22 25 38 219 6 4753 337 30 179 50
12 2 64 10 38 32 5297 1 34 0
2 12 65 46 25 10 0 5819 42 193
27 13 151 68 14 159 64 4 4730 59
0 3 15 57 421 10 1 141 38 5351
6000
5000
4000
3000
2000
1000
0


Confusion Matrix test (identity)
Error Rate: 11.04%
True label
Predicted label
0 1 2 3 4 5 6 7 8 9
959 0 22 10 4 13 17 4 6 6
0 1117 99 4 5 6 6 22 43 7
2 0 799 6 6 3 6 14 3 1
1 6 25 924 1 46 2 11 49 13
2 1 18 2 880 15 19 11 10 32
8 7 12 32 1 756 74 4 40 11
5 3 12 1 7 11 826 1 7 0
1 0 10 9 3 1 0 939 15 27
2 1 33 14 5 36 8 0 797 13
0 0 2 8 70 5 0 22 4 899
0 200 400 600 800 1000


Confusion Matrix train (sigmoid)
Error Rate: 3.03%
True label
Predicted label
0
1
2
3
4
5
6
7
8
9
5862 1 22 19 6 23 25 8 16 19
2 6673 6 5 10 6 3 21 22 4
8 32 5766 86 16 14 9 36 26 8
3 8 32 5825 0 59 0 7 52 52
6 4 26 1 5687 21 11 45 17 92
10 1 9 73 2 5218 36 4 60 19
12 3 13 4 15 39 5821 2 17 1
2 9 29 32 6 3 0 6067 13 70
15 6 47 60 9 25 13 11 5607 31
3 5 8 26 91 13 0 64 21 5653
6000
5000
4000
3000
2000
1000
0


Confusion Matrix test (sigmoid)
Error Rate: 4.39%
True label
Predicted label
0 1 2 3 4 5 6 7 8 9
970 0 8 2 2 7 10 0 5 9
0 1119 3 1 0 0 2 8 0 6
2 3 979 11 4 3 5 19 5 2
2 3 7 955 0 22 0 2 14 11
0 1 5 2 949 2 8 2 10 29
1 1 2 14 0 838 5 0 14 3
3 2 3 0 4 7 923 0 1 0
1 0 6 10 0 1 0 980 4 10
1 5 17 12 1 10 5 3 916 7
0 1 2 3 22 2 0 14 5 932
0
200
400
600
800
1000


Confusion Matrix train (sin)
Error Rate: 79.06%
True label
Predicted label
0 1 2 3 4 5 6 7 8 9
0: 1084 372 523 620 546 500 555 546 525 536
1: 1001 3605 953 989 954 895 937 992 964 946
2: 528 380 1097 570 512 537 529 564 556 577
3: 596 399 571 1140 608 484 582 584 584 579
4: 506 343 472 552 944 472 546 512 511 478
5: 351 254 373 351 361 701 377 401 362 362
6: 435 298 472 416 435 414 878 494 447 466
7: 533 447 586 592 585 588 568 1227 547 584
8: 425 333 462 453 444 437 477 451 905 436
9: 464 311 449 448 453 393 469 494 450 985
3500
3000
2500
2000
1500
1000
500



We see the best performance with the ReLU non-linearity.

Now let's see which values of L result in the best error. Let's plot L from 1000, which is slightly below the input data size, all the way to 3000 in 5 increments.

[36]:
```
def multiclass_error_rate(conf_matrix):
    return 100.0 * (1 - conf_matrix.trace()/conf_matrix.sum())

Ls = np.linspace(1000, 3000, 5)
g = relu
solutions_Ls = []
train_feat_gs = []
test_feat_gs = []
for L in tqdm(Ls):
    L = int(L)
    train_feat, test_feat = gen_features(trainX, testX, L)
    train_feat_g = g(train_feat)
    test_feat_g = g(test_feat)
    train_feat_gs.append(train_feat_g)
    test_feat_gs.append(test_feat_g)
```

```
    gs = solve_multiclass_classifier(pairs, train_feat_g, trainY)
    solutions_Ls.append(gs)
```

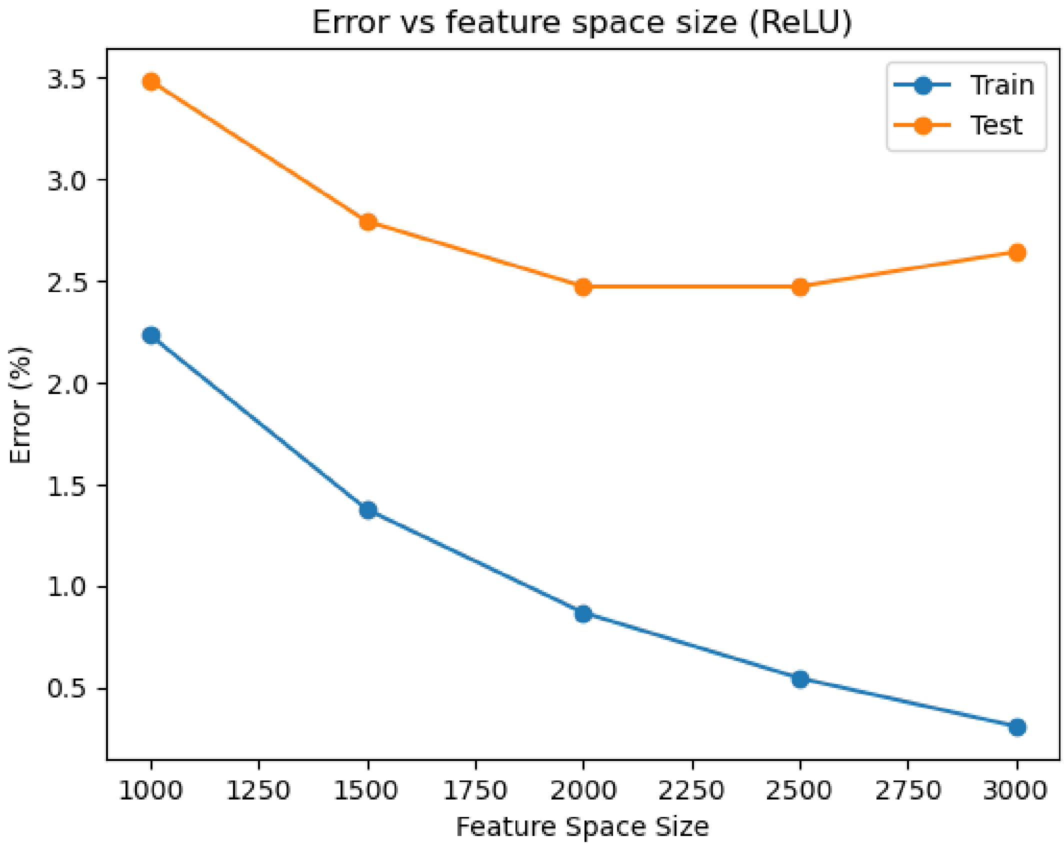
```
0%|          | 0/5 [00:00<?, ?it/s]

0%|          | 0/45 [00:00<?, ?it/s]

0%|          | 0/45 [00:00<?, ?it/s]

0%|          | 0/45 [00:00<?, ?it/s]

0%|          | 0/45 [00:00<?, ?it/s]

0%|          | 0/45 [00:00<?, ?it/s]
```

[37]:
```
test_error = []
train_error = []
mult_matrices = []
for gs, train_feat_g, test_feat_g in zip(tqdm(solutions_Ls), train_feat_gs,␣
 ↪test_feat_gs):
    mult_m = create_multiclass_confusion_matrix(gs, train_feat_g, trainY)
    train_error.append(multiclass_error_rate(mult_m))
    mult_m = create_multiclass_confusion_matrix(gs, test_feat_g, testY)
    test_error.append(multiclass_error_rate(mult_m))
    mult_matrices.append(mult_m)
```

```
0%|          | 0/5 [00:00<?, ?it/s]
```

[38]:
```
# Plotting the data with labels
plt.plot(Ls, train_error, label='Train', marker='o')
plt.plot(Ls, test_error, label='Test', marker='o')

# Adding title and labels
plt.title('Error vs feature space size (ReLU)')
plt.xlabel('Feature Space Size')
plt.ylabel('Error (%)')

# Adding a legend with labels for each line
plt.legend(['Train', 'Test'])
```

[38]:
```
<matplotlib.legend.Legend at 0x14c174cd0>
```



We see that the error rate on the training set reliably goes down as L increases. However, we see that this is simply overfitting the training data since the test performance stagnates or even becomes worse.

The optimal L seems to be ≈ 2500. If we increase L past that, training and inference time increases needlessly.