ECE174 Introduction to Linear and Nonlinear Optimization

Created by: Aleksandar Jeremic
For: Professor Piya Pal, Christine Lind
November 19th 2023
Fall 2023

0.1 Introduction

In Mini Project 1 I created a linear classifier. We are given a MNIST data set which has 70000 28x28 pixel handwritten digits (0-9). In my project I used Pandas, mostly for formatting my confusion tables, finding error and accuracy of my predictions. I also used numpy, numpy is a very useful python package for dealing with matricies when you need faster computation and certain functionalities like matrix multiplication. I also imported scipy.io to load the dataset into python and scipy.linalg to compute the pseudo inverse in my code (explained later). This code then predicts the handwritten digit of the test data using the data it was trained on. This code also only uses pseudo inverse, matrix multiplication, scalar multiplication, and vector addition to find the handwritten digits. (No high level functionality calling such as .lstsq())

0.1.1 Least Squares

The Least Squares problem

$$\min_{\mathbf{x}} \|\mathbf{y} - \mathbf{A}\mathbf{x}\|_2^2$$

Is a function used to find a line with the minimum squared error of a set of data (in our case) It finds the best linear relationship between the given dataset, in our case we use our linear relationship to classify digits, something "below" our classification "line" is considered not the digit we are looking for and anything "above" we consider the digit we are looking for. Solving the Least Squares problem can also be related to the orthogonal projection of the vector y onto the subspace spanned by the matrix A. We showed in class that solving the Least Squares problem above is the same as solving the normal equations below

$$\mathbf{A}^T\mathbf{A}\mathbf{x} = \mathbf{A}^T\mathbf{y}$$

We also know that the normal equations always have a solution. In the next section I will show how I solved the normal equations in my code.

0.1.2 scipy.linalg.pinv

The function pinv inside of the scipy library runs the PsuedoInverse, this equation solves the following equation

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

The PseudoInverse computes right hand side of the equation up untill the y, from there we multiply by our y to achieve our weights, this vector β along with the α we have constructed is our affine function which splits our "classes" and when we multiply these weights to a dataset it will be able to basically determine True or False,

0.1.3 Strategies for Solving

I use the Pseudo Inverse to solve our data matrix. The reason I used the Pseudo Inverse is because our Data Matrix is not full rank, this is obvious because there exist pixels in all of the training and testing image sets which will always have values of 0. There are some remedies to increase computation speed and to solve the normal function. The first Idea to speed up computation speed is to remove all the columns of our data matrix which involve only the value 0, these columns are not useful information hence we remove them. We do not need to do this because we are taking the pseudo inverse but it should improve performance and decrease computation time. The other option is a form of "regularization" this isnt as obvious as regularizing along the diagonal, this method would involve adding a normal distribution of noise to the entire matrix.

$$\mathbf{B} = \mathbf{A} + \epsilon\mathcal{N}(0,\, 1)$$

this regularization scalar epsilon makes our matrix of normal distribution appear as noise in our matrix and will almost always make our matrix A in our least squares problem full rank, with this regularization we could then compute the inverse not the pseudo inverse to solve our equations. The psuedo inverse still works because if the matrix is invertible the pseudo inverse equals the inverse. The documentation asks us not to use regularization so I decided to stick with the PseudoInverse.

0.1.4 Problem Statement

We are trying to solve the equation

$$\min_{\mathbf{\Theta}} \|\mathbf{y} - \mathbf{\Theta}\mathbf{X}\|_2^2$$

Where our $\mathbf{\Theta}$ is the weights matrix that has been concatenated with our scalar bias

$$\mathbf{\Theta} = \begin{bmatrix} \beta & \alpha \end{bmatrix}$$

and our X is our x vector with an addition of a 1's column to account for the α bias in the θ vector

$$\mathbf{X} = \begin{bmatrix} x & 1 \end{bmatrix}$$

where β is the weights and our α is our bias for our affine function. The solution to this specific least squares problem is

$$\mathbf{z}* = \mathbf{\Theta}(\mathbf{\Theta}^T\mathbf{\Theta})^{-1}\mathbf{\Theta}^T\mathbf{y}$$

This will be the solution for our specific testing set, when we take the sign of our $\mathbf{z}*$ We are left with our binary classification, Later when explaining the code I explain the difference between when and why we use sign to determine our prediction and when we use numpy argmax. I also explain in more detail as we approach those problems in this report

0.2 Binary Classifier

This is the very first steps taken in the project, I created a Binary classifier to determine if the handwritten digit is a 0 or not a zero, (One Vs All classifier). To do this we need to create a few functions preemptively. 2 of these functions are extremely important for my code.

0.2.1 Function: create weights

The first function is called create weights. This code finds the solution to the normal equation using pseudoinverse. and multiplies our result with the processed expected values. These processed expected values will be explained in our next part Below is the code for my solution to the normal equations and finds the array of Beta and Alpha weights

```python
# -*- coding: utf-8 -*-
"""
Created on Mon Nov 13 23:00:30 2023

@author: jerem
"""
import scipy.io as sp
import scipy.linalg
import numpy as np
import pandas as pd
def create_weights(train_x,processed_train_y):
    """

    Parameters
    ----------
    train_x : TYPE np.array
        DESCRIPTION.

        this is our training set we will use this in conjunction with the normal equations (pseudoinverse)

    processed_train_y : TYPE
        DESCRIPTION.

        this is our expected values (0-9) except we have alreafy run these expected values through a processor which sets
        1 -> to the value we are looking for
        -1 -> to any other digit

    Returns
    -------
    pre_signed : TYPE
        DESCRIPTION.

        these are our weights matrix times our y, this gives us the final solution to the normal equation

    """

    solution = scipy.linalg.pinv(train_x)

    pre_signed = solution@processed_train_y #gives a matrix that is our weights

    return pre_signed

```

Figure 1: Function: create weights

0.2.2 Function: if matches

The next function is my if matches function, this function serves as a processor for our expected values (y arrays) the function requires 2 inputs, an array, this numpy array is our y (expected) values, the second input (expected) is our digit we want to set our y array around, the array will be turned into values of 1 and -1. A value of 1 means that position of the array corresponds to our expected digit, if the y array values is converted to a -1 it means it is any other digit. For instance if expected is set to the integer 0 everywhere in our y array that corresponds to 0 will be converted to the digit 1 and every where in y that isn't 9 (digits 1-9) is converted to a -1.
The image of the code is on the next page

```python
def if_matches(testing,expected):
    """
    Parameters
    ----------
    testing : TYPE np.array
        DESCRIPTION.

        this is what our array is which we want to change to 1 or -1 based on what our expected value is,

    expected : TYPE integer
        DESCRIPTION.

        The integer which we are going to use in relation to setting our y = 1 for the digit we want to find and y=-1 for the
        other digits

    Returns
    -------
    prediction_train_conversion : TYPE np.array
        DESCRIPTION.

        This is our new y (our training or expected results (the real digit value of our handwritten digits))
        that has been processed to -1 and 1 for the given digit


    """
    prediction_train_conversion = np.where(testing == expected,1,-1)
    return prediction_train_conversion
```

Figure 2: Function: if matches

0.2.3 Function: Binary Classifier

We can finally construct our binary classifier function. Our binary classifier takes in 5 inputs. The first input is the digit (0-9) which we want to predict. our second input is our test data set, which is an array that has all of our test data, these are images of sizes 28x28, The domain however was changed to be an image of size 1x784 this can be done because our pixels are independent of each other and we prefer to have a row vector rather than stacking 10000 28x28 arrays. Our third input is our test y, which is the expected digit value of our test x images, this is an array of size 10000x1 with integer values (0-9) our fourth input is our train x which is the same as our test data set except we are using this data set as a training set our fifth input is our train y this couples with our train x as the correct values for the images so train x's 5th image is a digit which we can find because train y's 5th value is equal to what train x's image is showing

Our function takes our train y and processes it with whatever our prediction digit is to get our 1, and -1 values. then we take this train x and our processed train y and solve the normal equation using the create weights function. Now we can multiply our testing x data with our weights, this will give us a range of numbers that span the negative and positive real axis. We don't care too much about the value we care more about the sign of the prediction. We assume a negative sign to mean that the prediction is not our number. This is because we trained our data using a training y where we converted all of our other digits to -1, this function would work in reverse we would just need to flip some signs in our code. After taking the sign of the floats in our array we have successfully predicted all of our testing data!

```python
def binary_classifier(digit_to_predict,test_x,test_y,train_x,train_y):
    """
    Parameters
    ----------
    digit_to_predict : TYPE
        DESCRIPTION.
        this is the digit we are going to predict for one vs all classifier

    Returns
    -------
    predictions_train_binary : TYPE
        DESCRIPTION.
        this reutrns our prediction in binary (1 means it is our digit) (-1 means not our digit)
    """

    weights = create_weights(train_x,if_matches(train_y,digit_to_predict)) #using test data now

    predictions_on_train = test_x@weights

    predictions_train_binary = np.sign(predictions_on_train)

    return predictions_train_binary

```

Figure 3: Function: Binary Classifier

0.2.4 Function: Binary confusion matrix

This code was created to determine the confusion matrix, specificity, accuracy, precision, and error rate of my binary classifier this function takes the predicted values (1,-1) and the expected values (-1,1) and computes a confusion matrix (pandas dataframe) based on these results

below is the code and the output guesses for predicting the digit 0

```python
def analyze_binary(predicted_value,expected_value):
    """


    Parameters
    ----------
    predicted_value : TYPE numpy array
        DESCRIPTION.

        the values that my binary classifier predicted

    expected_value : TYPE numpy array
        DESCRIPTION.
        ANALYZE THE BINARY CLASSIFIER USING confusion matrix

    Returns
    -------
    None.

    """
    true_positive = 0
    true_negative = 0
    false_positive = 0
    false_negative = 0
    expected_true = np.count_nonzero(expected_value == 1)
    expected_false = np.count_nonzero(expected_value == -1)
    for i in range(len(expected_value)): #iterate over the list while also taking value
        if predicted_value[i] == expected_value[i] and predicted_value[i] == 1:
            true_positive += 1
        elif predicted_value[i] == expected_value[i] and predicted_value[i] == -1:
            true_negative += 1
        elif predicted_value[i] != expected_value[i] and expected_value[i] == 1:
                false_negative += 1

        else:
            false_positive += 1
    data_confusion_matrix = [[true_positive,false_negative,true_positive+false_negative],
                             [false_positive,true_negative,false_positive+true_negative],
                             [true_positive+false_positive,false_negative+true_negative,false_negative+true_negative+true_positive+false_positive]]
    indexes = ["Expected True",'Expected False','All']
    columns = ["Predicted True","Predicted_False",'Total']

    confusion_matrix = pd.DataFrame(data_confusion_matrix,columns = columns,index = indexes)
    error_rate = (false_positive + false_negative) / len(expected_value)
    true_positive_rate = true_positive / expected_true
    false_positive_rate = false_positive / expected_false
    true_negative_rate = true_negative / expected_false
    precision = true_positive / (true_positive + false_positive)
    data = [error_rate,true_positive_rate,false_positive_rate,true_negative_rate,precision]
    rates = pd.DataFrame(data,columns = ["rates"], index = ["error_rate","true_positive_rate","false_positive_rate","true_negative_rate","precision"])
    print(confusion_matrix)
    print(rates)
```

Figure 4: Function: Analyze Binary

the function call at the end of my file.

```
predictions_train_binary = binary_classifier(0,test_x,test_y,train_x,train_y)

analyze_binary(predictions_train_binary,if_matches(test_y,0)) #running on test data now
```

Figure 5: Function Call for Binary classifier and analyzation

The computed confusion matrix along with other useful insights As we can see we have a low error rate of only 1.5%! this means that we can quite often determine what is or isn't a 0! our precision is also very high this means that my code is able to guess correctly whether the digit was truly a 0 95% of the time

	Predicted True	Predicted_False	Total
Expected True	866	114	980
Expected False	43	8977	9020
All	909	9091	10000

	rates
error_rate	0.015700
true_positive_rate	0.883673
false_positive_rate	0.004767
true_negative_rate	0.995233
False Negative	0.012639
precision	0.952695

Figure 6: Binary Confusion Matrix

0.3 One Vs All Multiclass Classifier

The One vs. all Multiclass Classifier works similarly to the one vs all single class, I create my weights for every single digit (0-9) ten in total. the key difference comes from how I represent my data, I represent every single guess in a 10000x10 matrix, and instead of taking the sign to find the guess. I find the highest value. In this case, the highest value relates to the highest confidence of the guess so when I take the argmax of a row and the output gives me the index 3 (4th index) I know the most confident guess is the digit 3 (we are zero indexing) then we append this index to our prediction set and go to the next test/training point. Below is a screenshot of the code

```python
def one_vs_all_multi(train_x,train_y,test_x,test_y):
    """
    Parameters
    ----------
    train_x : TYPE np.array
        DESCRIPTION.

        the training data, we will be using this data to train our linear classifiers of each digit

    train_y : TYPE np.array
        DESCRIPTION.

        the training datas corresponding correct digit value, if were at training point x_i the digit
        associated with that is y_i

    test_x : TYPE np.array
        DESCRIPTION.

        our data set we will be testing using our linear classifier we trained

    test_y : TYPE np.array
        DESCRIPTION.

        not actually used I just included it to look nice

    Returns
    -------
    prediction : TYPE np.array
        DESCRIPTION.

        the predicted values from (0-9)
    """
    one_vs_all_weights = np.empty((train_x.shape[1],0))
    for i in range(10):
        x= create_weights(train_x,if_matches(train_y,i))


        one_vs_all_weights = np.concatenate((one_vs_all_weights,x),axis = 1) # this is all the weights for the different numbers 0-9

    preprocess_pred = test_x@one_vs_all_weights

    prediction = np.argmax(preprocess_pred,axis = 1)

    prediction = prediction.reshape(-1,1)

    return prediction
```

Figure 7: Function: One vs All Multiclass Classifier

Below is the matrix of predicted versus expected values and then the error and accuracy of guessing each digit

	Predicted 0	Predicted 1	Predicted 2	Predicted 3	Predicted 4
Expected 0	5682	7	18	14	24
Expected 1	2	6548	40	15	19
Expected 2	99	264	4792	149	108
Expected 3	42	167	176	5158	32
Expected 4	10	99	42	6	5212
Expected 5	164	95	28	432	105
Expected 6	108	74	61	1	70
Expected 7	55	189	37	47	170
Expected 8	75	493	63	226	105
Expected 9	68	60	20	117	371
Totals	6305	7996	5277	6165	6216

Figure 8: First Half

	Predicted 5	Predicted 6	Predicted 7	Predicted 8	Predicted 9	Totals
Expected 0	43	64	4	61	6	5923
Expected 1	31	14	12	55	6	6742
Expected 2	11	234	91	192	18	5958
Expected 3	125	56	115	135	125	6131
Expected 4	50	39	23	59	302	5842
Expected 5	3991	192	36	235	143	5421
Expected 6	90	5476	0	35	3	5918
Expected 7	9	2	5426	10	320	6265
Expected 8	221	56	20	4412	180	5851
Expected 9	12	4	492	38	4767	5949
Totals	4583	6137	6219	5232	5870	60000

Figure 9: Second Half

```
Error is 0.14226666666666665


                 Accuracy
Accuracy for 0  0.959311
Accuracy for 1  0.971225
Accuracy for 2  0.804297
Accuracy for 3  0.841298
Accuracy for 4   0.89216
Accuracy for 5  0.736211
Accuracy for 6  0.925313
Accuracy for 7  0.866081
Accuracy for 8  0.754059
Accuracy for 9  0.801311
```

Figure 10: Error values and Accuracy of individual digits

Now let's run our code on the testing data to see if we get any significant difference in our results

	Predicted 0	Predicted 1	Predicted 2	Predicted 3	Predicted 4
Expected 0	944	0	1	2	2
Expected 1	0	1107	2	2	3
Expected 2	18	54	813	26	15
Expected 3	4	17	23	880	5
Expected 4	0	22	6	1	881
Expected 5	23	18	3	72	24
Expected 6	18	10	9	0	22
Expected 7	5	40	16	6	26
Expected 8	14	46	11	30	27
Expected 9	15	11	2	17	80
Totals	1041	1325	886	1036	1085

Figure 11: First Half

	Predicted 5	Predicted 6	Predicted 7	Predicted 8	Predicted 9	Totals
Expected 0	7	14	2	7	1	980
Expected 1	1	5	1	14	0	1135
Expected 2	0	42	22	37	5	1032
Expected 3	17	9	21	22	12	1010
Expected 4	5	10	2	11	44	982
Expected 5	659	23	14	39	17	892
Expected 6	17	875	0	7	0	958
Expected 7	0	1	884	0	50	1028
Expected 8	40	15	12	759	20	974
Expected 9	1	1	77	4	801	1009
Totals	747	995	1035	900	950	10000

Figure 12: Second Half

And our error rates and Accuracy for the testing data set

```
Error is 0.13970000000000005


                 Accuracy
Accuracy for 0  0.963265
Accuracy for 1   0.97533
Accuracy for 2  0.787791
Accuracy for 3  0.871287
Accuracy for 4  0.897149
Accuracy for 5  0.738789
Accuracy for 6  0.913361
Accuracy for 7  0.859922
Accuracy for 8  0.779261
Accuracy for 9  0.793855
```

Figure 13: Error values and Accuracy of individual digits

As we can see our accuracy improved from the training set to the testing set, this is a good sign that we have created a working linear classifier that isn't overfitting any of our training set (we haven't added any features so overfitting shouldnt be possible from my understanding of overfitting) We also can see that it is much easier in both training and testing data to determine what is a 0 and what is a 1 over something like the digit 5 which has an accuracy of 73% on the testing and training set. Here is the function call at the end of the code first we see where we test on the training set and the next line is us testing on the testing set

```
predicted_multi_train = one_vs_all_multi(train_x,train_y,train_x,train_y)
analyze_multi_class(predicted_multi_train, train_y)

predicted_multi_test = one_vs_all_multi(train_x,train_y,test_x,test_y)
analyze_multi_class(predicted_multi_test, test_y)

```

Figure 14: Function Call

0.4 One Vs. One Classifier

The One Vs. One Classifier also builds upon our binary classifier. When we run the One Vs One Classifier we take only 2 specific digits eg. (0,1) we need to train a classifier to determine between the digit 0 and the digit one. The big difference is that when we train this classifier we only let it see the 2 digits. Per the example, we would need to weed out any training data that involves the other 8 digits (2-9). Then we run this 2 digits classifier for every non-overlapping/repeating combination of (0-9) ((0,1) = (1,0) so we don't train both of these) this gives us 45 total one vs one classifiers. then we run all 45 against the set we want to test it over. then we set up a voting system that chooses the best candidate digit. Eg. if the digits we classify are (0,1) we will remove all of our training set that isn't 0 or 1, we then run the training set to find the weights, and we also have to remove all of the training y data that isn't 0 or 1 and we set our 0 value to be 1 and 1 to be -1 similar to our one versus all classifier. From this, we can determine for this first one that if the one vs one classifier returns a positive value, it is more confident the digit is a 0. With this, we have constructed our first of 45 classifiers. We repeat the process using 2 nested for loops to run from (0,1) to (8,9). When determining the final predicted digit we run the test data, we take the sign of our predictions to return to our standard 1,-1 format and run through our list of predictions of size 10000x45, if the first (0th) index returns a 1 we increment a count for the prediction 0, if the first index returned a -1 we would increment our count for the prediction of the digit we set equal to -1, we run this through all 45 columns and take the highest count value and consider that our prediction if there is a tie we consider the smaller digit to be the winner. Its also very important to return all the counts to 0 after you have made your prediction I spent a couple of minutes debugging why my one vs one classifier loved to choose the digit 8!

here is the implementation in my code

```python

def run_ovo_all(train_x,train_y,test_x,test_y):
    """
    Parameters
    ----------
    train_x : TYPE nparray
        DESCRIPTION.
    train_y : TYPE np array
        DESCRIPTION.
    test_x : TYPE np array
        DESCRIPTION.
    test_y : TYPE np array
        DESCRIPTION.
        we run the one vs all tecnique except only on 2 digits, then we stack them all
        horizontally and implement our counting scheme
    Returns
    -------
    final_guesses : TYPE
        DESCRIPTION.
        our guesses based on our 45 one vs one classifiers
    """
    list_of_weights = []
    counter = 0
    for i in range(9):
        for j in range(i+1,10):
            counter +=1
            (ovo_train_x,ovo_train_y) = one_vs_one_train_setup(train_x,train_y, (i,j))
            weights = create_weights(ovo_train_x,if_matches(ovo_train_y,i))

            list_of_weights.append(weights)
    weights = np.array(list_of_weights)
    weights = np.squeeze(weights)
    tested_values = test_x@(weights.transpose())
    tested_values_binary = np.sign(tested_values)
    tuple_columns = [(i, j) for i in range(9) for j in range(i+1, 10)]

    dataframe = pd.DataFrame(tested_values_binary, columns = tuple_columns)

    counts_dict = {0 : 0, 1 : 0, 2 : 0, 3 : 0, 4 : 0, 5 : 0, 6 : 0, 7 : 0, 8 : 0, 9 : 0}

    guess_value = []

    for index,rows in dataframe.iterrows():
        for col_tuple,value in rows.items():
            a,b = col_tuple
            if value == 1:
                counts_dict[a] += 1
            elif value == -1:
                counts_dict[b] += 1
        guess_value.append([max(counts_dict, key = counts_dict.get)])

        for key in counts_dict:
            counts_dict[key] = 0

    final_guesses = np.array((guess_value))

    return final_guesses

```

Figure 15: Function: One Vs One Multiclass Classifier

Below is the results of running my one vs one multiclass classifier on the training set first, then the test set

	Predicted 0	Predicted 1	Predicted 2	Predicted 3	Predicted 4
Expected 0	5806	2	15	8	11
Expected 1	2	6623	36	17	7
Expected 2	51	68	5521	49	57
Expected 3	26	42	119	5579	9
Expected 4	14	18	20	5	5586
Expected 5	44	48	39	138	23
Expected 6	27	16	36	2	32
Expected 7	10	76	53	7	69
Expected 8	35	195	42	107	48
Expected 9	22	14	17	82	155
Totals	6037	7102	5898	5994	5997

Figure 16: First Half

	Predicted 5	Predicted 6	Predicted 7	Predicted 8	Predicted 9	Totals
Expected 0	19	22	6	33	1	5923
Expected 1	16	2	11	21	7	6742
Expected 2	21	42	44	92	13	5958
Expected 3	161	18	48	90	39	6131
Expected 4	11	14	16	8	150	5842
Expected 5	4967	93	10	46	13	5421
Expected 6	84	5690	0	30	1	5918
Expected 7	9	0	5881	5	155	6265
Expected 8	142	37	25	5155	65	5851
Expected 9	30	3	137	31	5458	5949
Totals	5460	5921	6178	5511	5902	60000

Figure 17: Second Half

and here is our error when we test on our training dataset

Error is 0.06223333333333336

	Accuracy
Accuracy for 0	0.980246
Accuracy for 1	0.982349
Accuracy for 2	0.926653
Accuracy for 3	0.909966
Accuracy for 4	0.956179
Accuracy for 5	0.916252
Accuracy for 6	0.961473
Accuracy for 7	0.938707
Accuracy for 8	0.881046
Accuracy for 9	0.917465

Figure 18: Error and Accuracy of One Vs One Multiclass Classifier on training dataset

We see a much better accuracy overall in the one vs one multiclass classifier than the one vs all, the error rate is less than half the error rate of the one vs all multiclass classifier! We can also tell that 8 is the hardest digit to predict, I believe this is because it is fairly similar to other digits such as 3,9, and 0. Once again the accuracy of guessing 0 and 1 are very high due to their unique shape among the digits. to me the digit 8 is also fairly similar to a 9, a 3 and could be similar to other digits if its not done well.

Next, we will test our linear classifier on the testing dataset.

	Predicted 0	Predicted 1	Predicted 2	Predicted 3	Predicted 4
Expected 0	961	0	1	1	0
Expected 1	0	1120	3	3	1
Expected 2	9	18	936	12	10
Expected 3	9	1	18	926	2
Expected 4	2	4	6	1	931
Expected 5	7	5	3	30	8
Expected 6	6	5	12	0	5
Expected 7	1	16	17	3	11
Expected 8	7	17	8	23	10
Expected 9	6	5	1	11	30
Totals	1008	1191	1005	1010	1008

Figure 19: First Half

	Predicted 5	Predicted 6	Predicted 7	Predicted 8	Predicted 9	Totals
Expected 0	6	8	3	0	0	980
Expected 1	1	4	1	2	0	1135
Expected 2	5	10	10	22	0	1032
Expected 3	20	1	7	21	5	1010
Expected 4	1	7	4	3	23	982
Expected 5	800	17	2	15	5	892
Expected 6	19	908	1	2	0	958
Expected 7	1	0	955	1	23	1028
Expected 8	36	10	10	840	13	974
Expected 9	12	0	21	3	920	1009
Totals	901	965	1014	909	989	10000

Figure 20: Second Half

and here is the error rates and the accuracy of each digit

Error is 0.07030000000000003

	Accuracy
Accuracy for 0	0.980612
Accuracy for 1	0.986784
Accuracy for 2	0.906977
Accuracy for 3	0.916832
Accuracy for 4	0.948065
Accuracy for 5	0.896861
Accuracy for 6	0.947808
Accuracy for 7	0.928988
Accuracy for 8	0.862423
Accuracy for 9	0.911794

Figure 21: Error and Accuracy of One Vs One Multiclass Classifier on testing dataset

Our error has increased slightly from the training to the testing but it makes sense that our training would have lower error when you consider the linear classifier trained specifically on that set.

We still have very high accuracy rates on all the digits, the hardest digit to predict is an 8.

Now lets look at my code that compiles all of the information about the testing along with the error rates the tables and the accuracy

0.4.1 Confusion Matrix for Multiclasses

The code creates a pandas dataframe which will store the information of our prediction and our guess values, it then creates the rows and columns for our dataframe which we have named Predicted (0-9) and Expected (0-9). It also adds an extra row and column to sum up all the values The code then increments through the entire length of our predicted matrix and increments every single prediction versus the expected result. Then its performs some simple math operation like finding the total times we guessed correctly and diving by the total number of guesses and takes 1- that value to find the error rate. The Accuracies for each individual digit are found by taking the times it guessed the digit correctly versus the total number of times that digit appeared

Below is the code for all the steps above

```python

def analyze_multi_class(predicted, expected):
    '''
    Parameters
    ----------
    predicted : TYPE
        DESCRIPTION.
        our input predicted array
    expected : TYPE
        DESCRIPTION.
        the actual value of the digit
    Returns
    -------
    None.
    this code tabulates all the error the matricies and finds the accuracies of all the digits
    '''
    columns = [f'Predicted {i}' for i in range(10)]
    indexes = [f'Expected {i}' for i in range(10)]
    columns.append('Totals')
    indexes.append('Totals')
    confusion_multi_class = pd.DataFrame(data = None, columns = columns, index = indexes)
    confusion_multi_class.loc[:,:] = 0

    for i in range(len(expected)):

        confusion_multi_class.loc[[f"Expected {expected[i][0]}"], [f"Predicted {predicted[i][0]}"]] += 1
    pd.set_option('display.max_columns', None)
    diagonal_sum = 1-(np.trace(confusion_multi_class.values)/len(predicted))
    confusion_multi_class.loc[["Totals"],['Totals']] = confusion_multi_class.sum().sum()
    for i in range(10):
        row_sum = confusion_multi_class.loc[f"Expected {i}"].sum()
        column_sum = confusion_multi_class[f"Predicted {i}"].sum()
        confusion_multi_class.loc[[f"Expected {i}"],['Totals']] = row_sum
        confusion_multi_class.loc[['Totals'],[f"Predicted {i}"]] = column_sum
    print("\n")
    print(confusion_multi_class)
    print("\n")
    print(f"Error is {diagonal_sum}")
    print("\n")
    accuracy_df = pd.DataFrame(data = None, columns = ['Accuracy'], index = [f"Accuracy for {i}" for i in range(10)])
    accuracy_df.loc[:,:] = 0
    for i in range(10):
        denominator = confusion_multi_class.at[f"Expected {i}","Totals"]
        numerator = confusion_multi_class.at[f"Expected {i}",f'Predicted {i}']


        accuracy_df.loc[f"Accuracy for {i}"] = numerator/denominator
    print(accuracy_df)

```

Figure 22: Function: Error and Accuracy of Multiclass classifier

0.5 Feature Space

In this section we will explore the topic of the feature space, here we are converting our 784 dimension input space into an L dimensional feature space. The feature space can be useful to overcome certain issues in the lower dimensional example. The feature space can improve our performance because as we add dimensions we are more likely to find a line that intersects the classes of our images, ie digits. We choose a feature space by first taking a matrix W which is of size 784x1000 and we left matrix multiply our 1x784 image vectors (all 60000/10000 of them) to this W, we then also take a vector b which is of size 1x1000 and add it to each feature vector of size 1x1000, both the b and the W matrix are obtained by creating a random gaussian distribution around 0 with standard deviation 1. Then we run our next testing and training matrices through a function to finalize our feature space. In this project, we will use 4 functions

The identity function, defined as:

$$f(x) = x$$

The sigmoid function, defined as:

$$f(x) = \frac{1}{1+e^{-x}}$$

The sinusoidal function, defined as:

$$f(x) = \sin(x)$$

The RELU function, defined as:

$$f(x) = \max(x, 0)$$

We begin our journey using only L = 1000, later we will vary L and see what happens to our error rate. First lets delve into our code for creating our feature space.

0.5.1 One Vs All Multiclass Classifier

0.5.2 Function: Feature Space

This code implements the aforementioned feature space from the input space.

```
def change_the_set(train_x,test_x,L,function_feature):
    '''


    Parameters
    ----------
    train_x : TYPE np array
        DESCRIPTION.
    test_x : TYPE np array
        DESCRIPTION.
    L : TYPE integer
        DESCRIPTION.
        This is the dimensionality of our feature space, we can reduce or
        increase the dimensionality
    function_feature : TYPE integer
        DESCRIPTION.
        (1-4) this changes which function we use when finalizing our feature space
    Returns
    -------
    new_train_x : TYPE np array
        DESCRIPTION.
        our new feature space training data
    new_test_x : TYPE np array
        DESCRIPTION.
        our new feature space testing data

    '''
    W = np.random.normal(0, 1, (train_x.shape[1],L))
    b = np.random.normal(0, 1, (1,L))

    new_train_x = train_x@W + b
    new_test_x = test_x@W + b

    if function_feature == 1:
        new_train_x = new_train_x
        new_test_x = new_test_x
    elif function_feature == 2:
        new_train_x = 1/(1+np.exp(-new_train_x))
        new_test_x = 1/(1+np.exp(-new_test_x))
    elif function_feature == 3:
        new_train_x = np.sin(np.pi/180*new_train_x)
        new_test_x = np.sin(np.pi/180*new_test_x)
    elif function_feature == 4:
        new_train_x = np.maximum(new_train_x,0)
        new_test_x = np.maximum(new_test_x,0)

    return (new_train_x,new_test_x)
```

Figure 23: Function: Feature Space Implementation

Below we will explore the one vs all multiclass classifier and what the affect is for the feature space with the identity function,the sigmoid function, the sinusoidal function, and the ReLU function with the training set first then the testing set

0.5.3 Identity Function

here is our matrix of our identity function for the training set

	Predicted 0	Predicted 1	Predicted 2	Predicted 3	Predicted 4
Expected 0	5682	7	18	14	24
Expected 1	2	6548	40	15	19
Expected 2	99	264	4792	149	108
Expected 3	42	167	176	5158	32
Expected 4	10	99	42	6	5212
Expected 5	164	95	28	432	105
Expected 6	108	74	61	1	70
Expected 7	55	189	37	47	170
Expected 8	75	493	63	226	105
Expected 9	68	60	20	117	371
Totals	6305	7996	5277	6165	6216

Figure 24: First Half

	Predicted 5	Predicted 6	Predicted 7	Predicted 8	Predicted 9	Totals
Expected 0	43	64	4	61	6	5923
Expected 1	31	14	12	55	6	6742
Expected 2	11	234	91	192	18	5958
Expected 3	125	56	115	135	125	6131
Expected 4	50	39	23	59	302	5842
Expected 5	3991	192	36	235	143	5421
Expected 6	90	5476	0	35	3	5918
Expected 7	9	2	5426	10	320	6265
Expected 8	221	56	20	4412	180	5851
Expected 9	12	4	492	38	4767	5949
Totals	4583	6137	6219	5232	5870	60000

Figure 25: Second Half

next is the error and accuracy

Error is 0.14226666666666665

	Accuracy
Accuracy for 0	0.959311
Accuracy for 1	0.971225
Accuracy for 2	0.804297
Accuracy for 3	0.841298
Accuracy for 4	0.89216
Accuracy for 5	0.736211
Accuracy for 6	0.925313
Accuracy for 7	0.866081
Accuracy for 8	0.754059
Accuracy for 9	0.801311

Figure 26: Errors and Accuracy of training digits in feature space

As we can see when we apply the identity function there exists no difference between training sets in the lower dimensional input space and the higher dimensional output space. Lets see if this is the case with the testing set too Here is the feature space performance on the testing set

	Predicted 0	Predicted 1	Predicted 2	Predicted 3	Predicted 4
Expected 0	944	0	1	2	2
Expected 1	0	1107	2	2	3
Expected 2	18	54	813	26	15
Expected 3	4	17	23	880	5
Expected 4	0	22	6	1	881
Expected 5	23	18	3	72	24
Expected 6	18	10	9	0	22
Expected 7	5	40	16	6	26
Expected 8	14	46	11	30	27
Expected 9	15	11	2	17	80
Totals	1041	1325	886	1036	1085

Figure 27: First Half

	Predicted 5	Predicted 6	Predicted 7	Predicted 8	Predicted 9	Totals
Expected 0	7	14	2	7	1	980
Expected 1	1	5	1	14	0	1135
Expected 2	0	42	22	37	5	1032
Expected 3	17	9	21	22	12	1010
Expected 4	5	10	2	11	44	982
Expected 5	659	23	14	39	17	892
Expected 6	17	875	0	7	0	958
Expected 7	0	1	884	0	50	1028
Expected 8	40	15	12	759	20	974
Expected 9	1	1	77	4	801	1009
Totals	747	995	1035	900	950	10000

Figure 28: Second Half

next is the error and accuracy

Error is 0.13970000000000005

	Accuracy
Accuracy for 0	0.963265
Accuracy for 1	0.97533
Accuracy for 2	0.787791
Accuracy for 3	0.871287
Accuracy for 4	0.897149
Accuracy for 5	0.738789
Accuracy for 6	0.913361
Accuracy for 7	0.859922
Accuracy for 8	0.779261
Accuracy for 9	0.793855

Figure 29: Errors and Accuracy of testing digits in feature space

We see once again our testing set has the same accuracy in the feature space as with the input space, this is because we maintain all the information of the input space and because we arent applying and function to break apart the classes any further we are essentially solving the same problem and hence we have the same errors and accuracies.

Lets see if this extends into our other feature functions, next we will run our sigmoid function on the training data then the testing data.

0.5.4 Sigmoid Function

Here is the feature space performance on the training set

	Predicted 0	Predicted 1	Predicted 2	Predicted 3	Predicted 4
Expected 0	5766	1	12	12	12
Expected 1	1	6605	45	12	16
Expected 2	47	30	5496	58	72
Expected 3	17	17	111	5579	11
Expected 4	4	22	25	1	5522
Expected 5	79	18	25	150	38
Expected 6	40	16	23	4	29
Expected 7	31	65	60	12	90
Expected 8	32	57	68	124	44
Expected 9	40	15	28	80	153
Totals	6057	6846	5893	6032	5987

Figure 30: First Half

	Predicted 5	Predicted 6	Predicted 7	Predicted 8	Predicted 9	Totals
Expected 0	31	42	8	36	3	5923
Expected 1	11	10	13	19	10	6742
Expected 2	13	47	84	97	14	5958
Expected 3	116	27	75	114	64	6131
Expected 4	11	39	8	25	185	5842
Expected 5	4844	110	24	87	46	5421
Expected 6	73	5702	2	29	0	5918
Expected 7	6	3	5860	13	125	6265
Expected 8	137	48	17	5235	89	5851
Expected 9	29	6	152	49	5397	5949
Totals	5271	6034	6243	5704	5933	60000

Figure 31: Second Half

next is the error and accuracy

Error is 0.0665666666666666

	Accuracy
Accuracy for 0	0.973493
Accuracy for 1	0.97968
Accuracy for 2	0.922457
Accuracy for 3	0.909966
Accuracy for 4	0.945224
Accuracy for 5	0.893562
Accuracy for 6	0.963501
Accuracy for 7	0.935355
Accuracy for 8	0.894719
Accuracy for 9	0.907211

Figure 32: Errors and Accuracy of training digits in feature space

We see a drastic increase in the performance along the training set with an error of only 6% and nearly 98% accuracy when predicting 0 or 1, even 8 a digit we had serious trouble predicting in the one vs all case is predicted at almost 90%! the addition of the sigmoid function adds a stark difference in the input space performance versus the feature space, this change in the dimensionality along with the sigmoid function allows for a linear classification at a precision greater than our lower dimensionality input space.

lets test our feature space on the testing data now

	Predicted 0	Predicted 1	Predicted 2	Predicted 3	Predicted 4
Expected 0	955	0	2	2	0
Expected 1	0	1118	2	2	1
Expected 2	15	2	945	11	14
Expected 3	2	0	14	933	1
Expected 4	0	3	4	0	925
Expected 5	13	2	3	27	9
Expected 6	11	4	3	1	6
Expected 7	2	14	23	4	6
Expected 8	7	1	8	24	6
Expected 9	5	5	2	14	28
Totals	1010	1149	1006	1018	996

Figure 33: First Half

	Predicted 5	Predicted 6	Predicted 7	Predicted 8	Predicted 9	Totals
Expected 0	6	11	1	3	0	980
Expected 1	1	4	1	6	0	1135
Expected 2	0	7	9	28	1	1032
Expected 3	16	4	14	19	7	1010
Expected 4	2	11	3	6	28	982
Expected 5	797	14	3	18	6	892
Expected 6	10	923	0	0	0	958
Expected 7	1	1	953	3	21	1028
Expected 8	25	12	6	878	7	974
Expected 9	9	1	22	6	917	1009
Totals	867	988	1012	967	987	10000

Figure 34: Second Half

next is the error and accuracy

Error is 0.06559999999999999

	Accuracy
Accuracy for 0	0.97449
Accuracy for 1	0.985022
Accuracy for 2	0.915698
Accuracy for 3	0.923762
Accuracy for 4	0.941955
Accuracy for 5	0.893498
Accuracy for 6	0.963466
Accuracy for 7	0.927043
Accuracy for 8	0.901437
Accuracy for 9	0.908821

Figure 35: Errors and Accuracy of testing digits in feature space

similar to our training set test we have accomplished a much greater accuracy and much lower error rate, the similarity between the testing and training set error leads me to believe we have not yet reached overfitting in our feature space with the sigmoid function. This function has indeed continued to perform well in the testing data versus the training data, unlike the identity function where both performed quite poorly

0.5.5 Sinusoidal Function

Next let's test the sinusoidal function, here we take our array values and transfer them to radians, if we do not do this our values our values are too large and the sinusoidal function wraps around itself so many times that it is extremely hard to regain any information because we do not know if it turned the circle 5 times or 10 times, so our data scatters and our classifier cannot work.

Here is the feature space performance on the training set

	Predicted 0	Predicted 1	Predicted 2	Predicted 3	Predicted 4
Expected 0	5735	5	13	15	13
Expected 1	0	6604	36	9	14
Expected 2	45	117	5315	77	86
Expected 3	12	101	93	5481	18
Expected 4	8	63	20	3	5432
Expected 5	66	62	11	170	55
Expected 6	42	38	40	5	45
Expected 7	25	124	38	22	101
Expected 8	31	198	43	134	57
Expected 9	31	35	14	91	241
Totals	5995	7347	5623	6007	6062

Figure 36: First Half

	Predicted 5	Predicted 6	Predicted 7	Predicted 8	Predicted 9	Totals
Expected 0	30	58	7	42	5	5923
Expected 1	15	11	15	32	6	6742
Expected 2	9	83	96	113	17	5958
Expected 3	123	23	98	99	83	6131
Expected 4	24	46	10	38	198	5842
Expected 5	4734	128	26	93	76	5421
Expected 6	65	5651	1	31	0	5918
Expected 7	8	1	5767	21	158	6265
Expected 8	149	42	22	5065	110	5851
Expected 9	23	4	203	35	5272	5949
Totals	5180	6047	6245	5569	5925	60000

Figure 37: Second Half

next is the error and accuracy

```
Error is 0.08240000000000003
```

	Accuracy
Accuracy for 0	0.968259
Accuracy for 1	0.979531
Accuracy for 2	0.892078
Accuracy for 3	0.893981
Accuracy for 4	0.929819
Accuracy for 5	0.873271
Accuracy for 6	0.954883
Accuracy for 7	0.920511
Accuracy for 8	0.865664
Accuracy for 9	0.886199

Figure 38: Errors and Accuracy of training digits in feature space

We see a slight increase in the number of errors and a decrease in the accuracy when running our classification on the training set. The sinusoidal function might be a worse candidate for our classification problem

lets see how it performs on the testing data

	Predicted 0	Predicted 1	Predicted 2	Predicted 3	Predicted 4
Expected 0	963	0	1	1	0
Expected 1	0	1118	3	1	2
Expected 2	11	17	912	14	10
Expected 3	2	4	15	930	1
Expected 4	2	11	2	0	911
Expected 5	11	9	1	28	13
Expected 6	12	5	4	1	11
Expected 7	1	27	16	3	17
Expected 8	7	15	13	22	13
Expected 9	7	9	2	17	41
Totals	1016	1215	969	1017	1019

Figure 39: First Half

	Predicted 5	Predicted 6	Predicted 7	Predicted 8	Predicted 9	Totals
Expected 0	4	6	2	3	0	980
Expected 1	1	4	1	5	0	1135
Expected 2	1	15	20	27	5	1032
Expected 3	12	5	17	16	8	1010
Expected 4	1	11	2	5	37	982
Expected 5	767	16	12	24	11	892
Expected 6	14	909	0	1	1	958
Expected 7	1	1	934	1	27	1028
Expected 8	24	11	11	850	8	974
Expected 9	5	1	22	8	897	1009
Totals	830	979	1021	940	994	10000

Figure 40: Second Half

next is the error and accuracy

Error is 0.08089999999999997

	Accuracy
Accuracy for 0	0.982653
Accuracy for 1	0.985022
Accuracy for 2	0.883721
Accuracy for 3	0.920792
Accuracy for 4	0.927699
Accuracy for 5	0.859865
Accuracy for 6	0.948852
Accuracy for 7	0.90856
Accuracy for 8	0.87269
Accuracy for 9	0.888999

Figure 41: Errors and Accuracy of testing digits in feature space

Our classifier runs slightly better on the testing dataset than the training however it is still outclassed by the sigmoid.

Lets look into the ReLU function

0.5.6 ReLU Function

When speaking with TA Kuan-Lin Chen during office hours he spoke on the use of ReLU function in neural networks and explained how widely used it is and that chatGPT uses the ReLU function all the time when parsing your statements. As a result, I expect this feature space to outperform the others

Here is the feature space performance on the training set

	Predicted 0	Predicted 1	Predicted 2	Predicted 3	Predicted 4
Expected 0	5786	2	8	6	4
Expected 1	1	6620	39	15	9
Expected 2	37	47	5551	46	48
Expected 3	11	19	99	5664	6
Expected 4	7	39	13	3	5511
Expected 5	33	18	12	104	28
Expected 6	31	17	12	2	21
Expected 7	16	70	49	16	51
Expected 8	20	67	45	82	28
Expected 9	20	19	14	83	132
Totals	5962	6918	5842	6021	5838

Figure 42: First Half

	Predicted 5	Predicted 6	Predicted 7	Predicted 8	Predicted 9	Totals
Expected 0	13	38	10	50	6	5923
Expected 1	9	7	11	25	6	6742
Expected 2	16	34	67	101	11	5958
Expected 3	107	14	56	95	60	6131
Expected 4	5	46	9	21	188	5842
Expected 5	5044	83	8	52	39	5421
Expected 6	63	5747	2	23	0	5918
Expected 7	7	1	5907	9	139	6265
Expected 8	78	48	22	5395	66	5851
Expected 9	28	4	116	50	5483	5949
Totals	5370	6022	6208	5821	5998	60000

Figure 43: Second Half

next is the error and accuracy

```
Error is 0.0548666666666662


                 Accuracy
Accuracy for 0   0.97687
Accuracy for 1  0.981904
Accuracy for 2  0.931688
Accuracy for 3   0.92383
Accuracy for 4  0.943341
Accuracy for 5  0.930456
Accuracy for 6  0.971105
Accuracy for 7  0.942857
Accuracy for 8  0.922065
Accuracy for 9  0.921668
```

Figure 44: Errors and Accuracy of training digits in feature space

As expected we see the highest accuracy and the lowest error rates for the ReLU function. Our classifier is predicting 8 correctly over 92% of the time now! We also see that we are predicting the digit 1 over 98% of the time!

Lets test our function on the testing set before we jump to conclusion on which function is the best.

Here is the feature space performance on the testing set

	Predicted 0	Predicted 1	Predicted 2	Predicted 3	Predicted 4
Expected 0	961	1	0	1	1
Expected 1	0	1122	2	2	1
Expected 2	7	1	961	10	8
Expected 3	2	3	9	944	0
Expected 4	2	6	4	0	929
Expected 5	8	1	3	18	6
Expected 6	10	3	3	0	3
Expected 7	2	14	15	4	10
Expected 8	7	1	10	16	8
Expected 9	5	7	1	13	30
Totals	1004	1159	1008	1008	996

Figure 45: First Half

	Predicted 5	Predicted 6	Predicted 7	Predicted 8	Predicted 9	Totals
Expected 0	1	8	2	5	0	980
Expected 1	1	5	0	2	0	1135
Expected 2	0	10	10	24	1	1032
Expected 3	23	2	8	13	6	1010
Expected 4	0	7	2	2	30	982
Expected 5	828	10	5	10	3	892
Expected 6	7	929	0	3	0	958
Expected 7	1	1	951	2	28	1028
Expected 8	12	11	6	897	6	974
Expected 9	5	3	10	12	923	1009
Totals	878	986	994	970	997	10000

Figure 46: Second Half

next is the error and accuracy

Error is 0.05549999999999994

	Accuracy
Accuracy for 0	0.980612
Accuracy for 1	0.988546
Accuracy for 2	0.931202
Accuracy for 3	0.934653
Accuracy for 4	0.946029
Accuracy for 5	0.928251
Accuracy for 6	0.969729
Accuracy for 7	0.925097
Accuracy for 8	0.920945
Accuracy for 9	0.914767

Figure 47: Errors and Accuracy of testing digits in feature space

Our classifier is performing equally as good on the testing set as on the training set! This function appears to be the best at performing well on the training and testing datasets
Now its time to move on to the One Vs One classifier for the 4 different functions we apply to our feature space.

0.6 One Vs One Multiclass Classifier

0.6.1 Identity Function

here is the matrix of our identity function for the training set

	Predicted 0	Predicted 1	Predicted 2	Predicted 3	Predicted 4
Expected 0	5806	2	15	8	11
Expected 1	2	6623	36	17	7
Expected 2	50	68	5525	46	57
Expected 3	26	41	119	5579	9
Expected 4	14	18	20	5	5586
Expected 5	43	44	39	137	23
Expected 6	27	15	36	2	31
Expected 7	9	73	53	8	66
Expected 8	35	193	43	107	48
Expected 9	22	13	17	82	155
Totals	6034	7090	5903	5991	5993

Figure 48: First Half

	Predicted 5	Predicted 6	Predicted 7	Predicted 8	Predicted 9	Totals
Expected 0	19	22	6	33	1	5923
Expected 1	16	2	11	21	7	6742
Expected 2	21	42	44	92	13	5958
Expected 3	161	19	48	90	39	6131
Expected 4	11	14	16	8	150	5842
Expected 5	4972	93	10	47	13	5421
Expected 6	83	5692	0	31	1	5918
Expected 7	9	0	5888	5	154	6265
Expected 8	142	37	25	5156	65	5851
Expected 9	31	3	137	31	5458	5949
Totals	5465	5924	6185	5514	5901	60000

Figure 49: Second Half

next is the error and accuracy

Error is 0.06191666666666662

	Accuracy
Accuracy for 0	0.980246
Accuracy for 1	0.982349
Accuracy for 2	0.927325
Accuracy for 3	0.909966
Accuracy for 4	0.956179
Accuracy for 5	0.917174
Accuracy for 6	0.961811
Accuracy for 7	0.939824
Accuracy for 8	0.881217
Accuracy for 9	0.917465

Figure 50: Errors and Accuracy of training digits in feature space

We see a higher accuracy compared to our one vs all classifier as expected, this time we see a slightly better accuracy and lower error rate compared to our input space classifier, the increase in dimension has a very minor affect on the way our one vs one classifier functions, the difference is small to the point of only a few guess were changed. Its interesting to point out the ReLU function one vs all is actually outperforming our one vs one classifier with the identity function!

Lets look at the test data now

	Predicted 0	Predicted 1	Predicted 2	Predicted 3	Predicted 4
Expected 0	961	0	1	1	0
Expected 1	0	1118	4	3	1
Expected 2	9	19	938	11	10
Expected 3	9	2	18	926	2
Expected 4	3	2	7	1	932
Expected 5	6	5	3	30	8
Expected 6	7	5	12	0	5
Expected 7	1	14	18	3	9
Expected 8	7	17	8	23	9
Expected 9	6	5	1	11	29
Totals	1009	1187	1010	1009	1005

Figure 51: First Half

	Predicted 5	Predicted 6	Predicted 7	Predicted 8	Predicted 9	Totals
Expected 0	6	8	3	0	0	980
Expected 1	1	5	1	2	0	1135
Expected 2	4	10	9	22	0	1032
Expected 3	19	1	7	21	5	1010
Expected 4	1	7	4	3	22	982
Expected 5	800	17	2	15	6	892
Expected 6	19	908	0	2	0	958
Expected 7	1	0	957	2	23	1028
Expected 8	36	10	10	841	13	974
Expected 9	12	0	21	4	920	1009
Totals	899	966	1014	912	989	10000

Figure 52: Second Half

next is the error and accuracy

```
Error is 0.06989999999999996


                   Accuracy
Accuracy for 0  0.980612
Accuracy for 1  0.985022
Accuracy for 2  0.908915
Accuracy for 3  0.916832
Accuracy for 4  0.949084
Accuracy for 5  0.896861
Accuracy for 6  0.947808
Accuracy for 7  0.930934
Accuracy for 8   0.86345
Accuracy for 9  0.911794
```

Figure 53: Errors and Accuracy of testing digits in feature space

here we see a slight increase in the errors, this increase is too minor to pin on anything and is most likely due to the testing set being different from the training set.

next lets look at the sigmoid function and how it relates to the one vs one multiclass classifier

0.6.2 Sigmoid Function

Our sigmoid function was the second best function to apply to our feature space, hopefully our error will be reduced from the one vs all multi class classifier

	Predicted 0	Predicted 1	Predicted 2	Predicted 3	Predicted 4
Expected 0	5863	1	9	2	6
Expected 1	1	6667	27	10	11
Expected 2	33	13	5766	23	30
Expected 3	13	6	69	5864	2
Expected 4	7	11	12	0	5693
Expected 5	20	5	19	65	15
Expected 6	21	7	9	1	6
Expected 7	5	21	28	9	35
Expected 8	14	16	22	50	19
Expected 9	17	9	13	54	89
Totals	5994	6756	5974	6078	5906

Figure 54: First Half

	Predicted 5	Predicted 6	Predicted 7	Predicted 8	Predicted 9	Totals
Expected 0	8	12	3	18	1	5923
Expected 1	2	2	7	13	2	6742
Expected 2	3	17	25	40	8	5958
Expected 3	63	5	27	65	17	6131
Expected 4	1	15	11	6	86	5842
Expected 5	5225	30	4	28	10	5421
Expected 6	38	5817	0	19	0	5918
Expected 7	3	0	6083	8	73	6265
Expected 8	53	16	10	5622	29	5851
Expected 9	19	2	63	25	5658	5949
Totals	5415	5916	6233	5844	5884	60000

Figure 55: Second Half

next is the error and accuracy

```
Error is 0.029033333333333355


                Accuracy
Accuracy for 0   0.98987
Accuracy for 1  0.988876
Accuracy for 2  0.967774
Accuracy for 3  0.956451
Accuracy for 4  0.974495
Accuracy for 5  0.963844
Accuracy for 6  0.982933
Accuracy for 7   0.97095
Accuracy for 8  0.960861
Accuracy for 9  0.951084
```

Figure 56: Errors and Accuracy of training digits in feature space

We have exceptional results this time around our classifier is correct over 97% of them time! Lets look at our testing data to see if it performs as well.

	Predicted 0	Predicted 1	Predicted 2	Predicted 3	Predicted 4
Expected 0	971	0	2	0	0
Expected 1	0	1125	2	2	0
Expected 2	11	0	981	5	5
Expected 3	1	0	9	966	0
Expected 4	0	1	3	0	946
Expected 5	7	1	1	24	2
Expected 6	8	3	3	0	7
Expected 7	2	7	23	2	8
Expected 8	4	0	4	15	5
Expected 9	7	5	1	11	20
Totals	1011	1142	1029	1025	993

Figure 57: First Half

	Predicted 5	Predicted 6	Predicted 7	Predicted 8	Predicted 9	Totals
Expected 0	4	0	1	2	0	980
Expected 1	1	2	1	2	0	1135
Expected 2	2	7	8	12	1	1032
Expected 3	13	0	6	13	2	1010
Expected 4	0	6	2	3	21	982
Expected 5	832	9	2	12	2	892
Expected 6	8	924	0	5	0	958
Expected 7	1	0	969	0	16	1028
Expected 8	11	4	3	923	5	974
Expected 9	4	1	16	4	940	1009
Totals	876	953	1008	976	987	10000

Figure 58: Second Half

next is the error and accuracy

Error is 0.042300000000000004

	Accuracy
Accuracy for 0	0.990816
Accuracy for 1	0.991189
Accuracy for 2	0.950581
Accuracy for 3	0.956436
Accuracy for 4	0.96334
Accuracy for 5	0.932735
Accuracy for 6	0.964509
Accuracy for 7	0.942607
Accuracy for 8	0.947639
Accuracy for 9	0.931615

Figure 59: Errors and Accuracy of testing digits in feature space

Our accuracy decreased but is still very good! I believe that the accuracy has decreased because of slight overfitting however I wouldn't consider overfitting to be a major error until we see a bigger difference in the training and testing accuracies/error rates.

Lets look at how our one vs one classifier performs on the sinusoidal function variant

0.6.3 Sinusoidal Function

Lets see if the trend continues of very low error rate and a slight increase in the error as we move from the training data to the testing data!

	Predicted 0	Predicted 1	Predicted 2	Predicted 3	Predicted 4
Expected 0	5861	2	4	4	7
Expected 1	2	6677	27	8	7
Expected 2	23	22	5737	27	35
Expected 3	10	16	80	5827	4
Expected 4	3	10	16	1	5694
Expected 5	22	15	17	68	13
Expected 6	17	8	12	0	12
Expected 7	4	30	36	6	33
Expected 8	10	37	32	60	24
Expected 9	15	11	10	44	86
Totals	5967	6828	5971	6045	5915

Figure 60: First Half

	Predicted 5	Predicted 6	Predicted 7	Predicted 8	Predicted 9	Totals
Expected 0	17	8	2	17	1	5923
Expected 1	3	0	9	3	6	6742
Expected 2	9	27	32	37	9	5958
Expected 3	69	5	40	59	21	6131
Expected 4	2	16	7	5	88	5842
Expected 5	5203	47	3	23	10	5421
Expected 6	38	5819	0	12	0	5918
Expected 7	3	1	6067	6	79	6265
Expected 8	56	22	9	5569	32	5851
Expected 9	16	3	76	18	5670	5949
Totals	5416	5948	6245	5749	5916	60000

Figure 61: Second Half

next is the error and accuracy

```
Error is 0.031266666666666665
```

	Accuracy
Accuracy for 0	0.989532
Accuracy for 1	0.990359
Accuracy for 2	0.962907
Accuracy for 3	0.950416
Accuracy for 4	0.974666
Accuracy for 5	0.959786
Accuracy for 6	0.983271
Accuracy for 7	0.968396
Accuracy for 8	0.951803
Accuracy for 9	0.953101

Figure 62: Errors and Accuracy of training digits in feature space

Our error rate has improved from the one vs all classifier and as with the one vs all we see one of the worse error rates with the sinusoidal function application to the feature space. Even so we have very good performance.

lets see how our function performs on the testing data

	Predicted 0	Predicted 1	Predicted 2	Predicted 3	Predicted 4
Expected 0	966	0	0	1	1
Expected 1	0	1127	1	3	0
Expected 2	8	1	972	8	8
Expected 3	1	0	11	958	2
Expected 4	2	0	6	0	946
Expected 5	7	2	1	19	2
Expected 6	4	3	5	0	4
Expected 7	1	12	17	2	4
Expected 8	5	1	5	15	9
Expected 9	3	5	4	12	24
Totals	997	1151	1022	1018	1000

Figure 63: First Half

	Predicted 5	Predicted 6	Predicted 7	Predicted 8	Predicted 9	Totals
Expected 0	4	4	2	2	0	980
Expected 1	1	1	1	1	0	1135
Expected 2	3	6	11	15	0	1032
Expected 3	10	1	9	15	3	1010
Expected 4	0	5	2	3	18	982
Expected 5	834	12	2	10	3	892
Expected 6	10	930	0	2	0	958
Expected 7	1	0	972	1	18	1028
Expected 8	14	5	9	908	3	974
Expected 9	3	0	15	7	936	1009
Totals	880	964	1023	964	981	10000

Figure 64: Second Half

next is the error and accuracy

```
Error is 0.04510000000000003


                Accuracy
Accuracy for 0  0.985714
Accuracy for 1  0.992952
Accuracy for 2   0.94186
Accuracy for 3  0.948515
Accuracy for 4   0.96334
Accuracy for 5  0.934978
Accuracy for 6  0.970772
Accuracy for 7  0.945525
Accuracy for 8  0.932238
Accuracy for 9  0.927651
```

Figure 65: Errors and Accuracy of testing digits in feature space

We once again see the trend of a slight decrease in the accuracy as we implement our code onto the testing data. The function has definitely improved from one vs one, which continues to support the idea that it is the better classifier between one vs all. we also notice that digits 5,8, and 9 are the most difficult to predict here.

Lets move on to the ReLU function mapping, this will hopefully have very good results!

0.6.4 ReLU Function

Now we finally get to apply our ReLU function to the one vs one classifier, the one vs one classifier outperforms the one vs all and even the ReLU function worked very well on the one vs all in the feature space.

Lets test it on our training data set

	Predicted 0	Predicted 1	Predicted 2	Predicted 3	Predicted 4
Expected 0	5875	3	5	4	5
Expected 1	2	6679	22	6	8
Expected 2	18	14	5816	18	21
Expected 3	5	5	56	5908	2
Expected 4	4	6	12	0	5728
Expected 5	11	3	10	48	6
Expected 6	10	4	7	0	8
Expected 7	3	21	31	7	30
Expected 8	6	18	23	40	12
Expected 9	9	4	9	39	64
Totals	5943	6757	5991	6070	5884

Figure 66: First Half

	Predicted 5	Predicted 6	Predicted 7	Predicted 8	Predicted 9	Totals
Expected 0	6	9	3	10	3	5923
Expected 1	0	2	6	9	8	6742
Expected 2	5	8	26	27	5	5958
Expected 3	57	2	31	47	18	6131
Expected 4	1	10	10	10	61	5842
Expected 5	5283	33	4	14	9	5421
Expected 6	28	5852	0	9	0	5918
Expected 7	3	0	6115	3	52	6265
Expected 8	29	21	13	5669	20	5851
Expected 9	24	0	48	20	5732	5949
Totals	5436	5937	6256	5818	5908	60000

Figure 67: Second Half

next is the error and accuracy

```
Error is 0.022383333333333331


               Accuracy
Accuracy for 0  0.991896
Accuracy for 1  0.990656
Accuracy for 2  0.976166
Accuracy for 3  0.963627
Accuracy for 4  0.980486
Accuracy for 5  0.974543
Accuracy for 6  0.988848
Accuracy for 7  0.976057
Accuracy for 8  0.968894
Accuracy for 9  0.963523
```

Figure 68: Errors and Accuracy of training digits in feature space

We are able to predict the digits 0 and 1 with an accuracy of over 99%! We are worst at predicting the digit 9 now and even then its with an accuracy of over 96%.

Lets see if this holds up with the testing data now

	Predicted 0	Predicted 1	Predicted 2	Predicted 3	Predicted 4
Expected 0	974	0	1	1	0
Expected 1	0	1127	2	4	0
Expected 2	4	1	995	4	4
Expected 3	0	0	11	974	0
Expected 4	1	1	6	0	949
Expected 5	4	1	2	15	4
Expected 6	8	3	4	0	6
Expected 7	1	7	12	2	8
Expected 8	5	0	3	9	7
Expected 9	4	4	4	7	14
Totals	1001	1144	1040	1016	992

Figure 69: First Half

	Predicted 5	Predicted 6	Predicted 7	Predicted 8	Predicted 9	Totals
Expected 0	0	1	1	2	0	980
Expected 1	0	1	0	1	0	1135
Expected 2	1	3	8	12	0	1032
Expected 3	11	0	5	6	3	1010
Expected 4	0	3	0	3	19	982
Expected 5	851	7	1	7	0	892
Expected 6	7	928	0	2	0	958
Expected 7	0	1	983	2	12	1028
Expected 8	9	3	6	927	5	974
Expected 9	1	0	12	6	957	1009
Totals	880	947	1016	968	996	10000

Figure 70: Second Half

next is the error and accuracy

```
Error is 0.033499999999999974


                 Accuracy
Accuracy for 0  0.993878
Accuracy for 1  0.992952
Accuracy for 2  0.964147
Accuracy for 3  0.964356
Accuracy for 4  0.966395
Accuracy for 5  0.954036
Accuracy for 6  0.968685
Accuracy for 7  0.956226
Accuracy for 8  0.951745
Accuracy for 9  0.948464
```

Figure 71: Errors and Accuracy of testing digits in feature space

As expected we continue the trend of slightly worse testing data. The ReLU function is definitely the best generalizer function that we have seen so far, this function has given us the best accuracies. The ReLU function is definitely the best feature mapping of the 4 we are testing

0.6.5 Comments on the L = 1000 Feature Space

From the data we have seen it is clear that the ReLU function is the best function to apply our feature vector to. This function has the lowest error rates and likewise the best accuracy. We also have seen that not touching the feature vector after adding the normal (Gaussian) distribution has the worst rates for the feature space performing either exactly the same or very slightly better.

0.7 Feature Space and Overfitting

0.7.1 Feature Space <= 784

When applying a Feature Space of less than 784 we see some degeneration of the data, for Instance with an L = 10 we get nearly 50% error rate, this is because as we decrease the dimension it is no longer possible to hold all of the information of our images and we truncate alot of useful information until our classifier can no longer see a difference in alot of the images. However the error rate isnt so egregious as we get close to a feature space of dimension 784 the error rate quickly passes back under the 10% range at a feature space of around 200 using the ReLU function. However I would not recommend using the feature space to reduce the dimensionality of the input space unless it can be shown that it wont affect the data (you only remove useless dimensions like the 0's we removed earlier) And using the feature space to increase dimension can improve performance.

0.7.2 Feature Space >= 784

For feature spaces greater than 784 we introduce new ways for our classifier to find lines which separate our classes, this will increase our accuracy on the training set but if we increase the feature space too much we run into a problem called overfitting, this is where our training data can be fitted so perfectly when it reaches testing data the fitting is actually incorrect and the error begins to increase again. This error would be especially apparent if there were many digits in the testing set that were edge cases not seen in the training set. The graph of the accuracy versus the dimensionality added/subtracted should resemble the very rough example below. This graph is not meant to be taken literally I just drew it one office hour and it helped me and some other students understand explain and understand what is happening in our code.

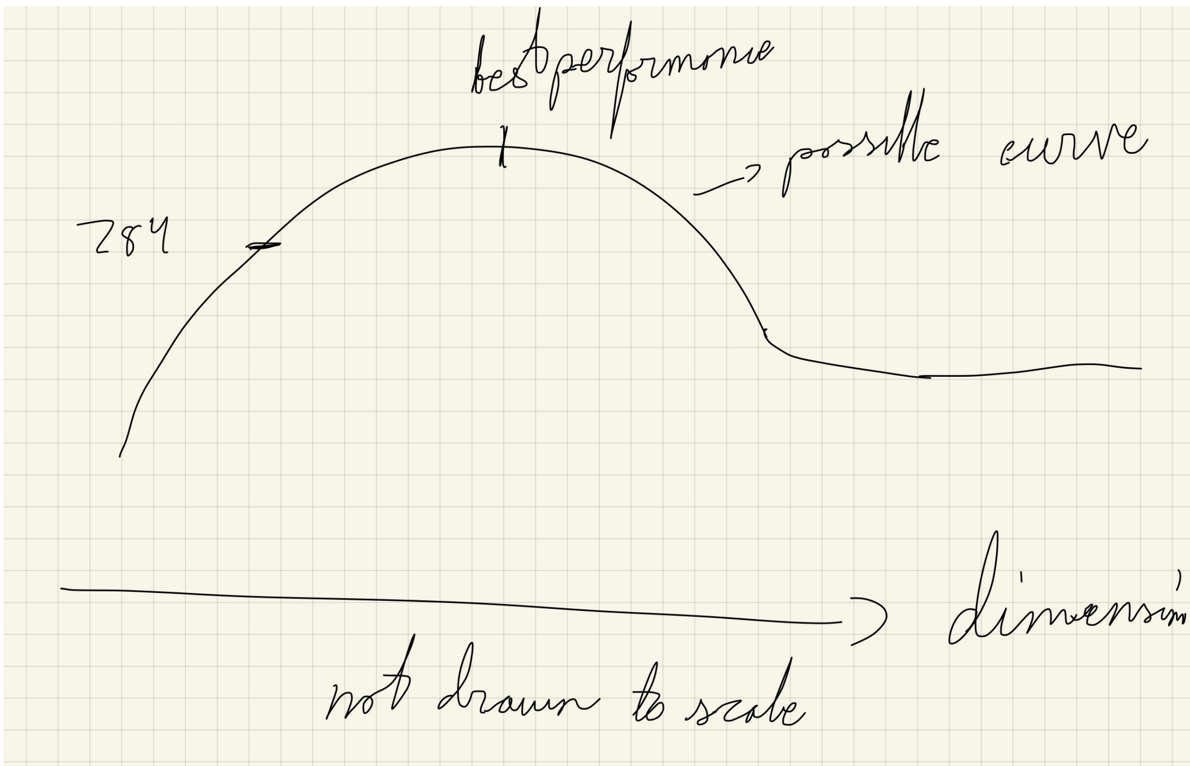

Figure 72: Testing Performance Based on Dimensionality of Feature Space

The figure below illustrates this problem, the trend is quite clear as we move to higher and higher dimensionalities, the training data has lower and lower error while the testing set error doesn't change. The testing set graph appears to drop to 0 after some time, but after about 12 hours of running through my code I needed to turn it off and it hadn't quite finished testing the training and error data up to L = 5000. *Only pay attention to the data before the drop in the test and train data graph.*

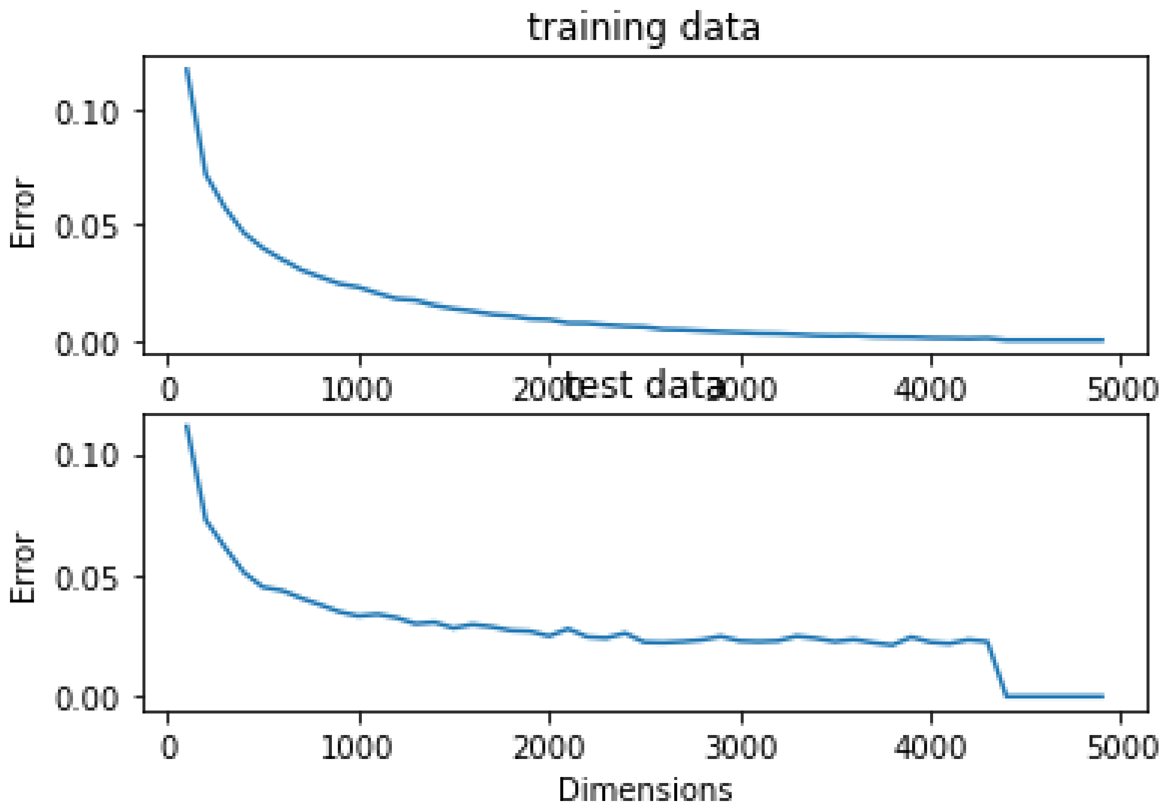

Figure 73: Testing and Training Performance Based on Dimensionality of Feature Space

0.7.3 Changing Feature Space dimensionality

In this section, Ill quickly cover the code which runs the ReLU function and plot a function of error as a function of dimension. We can clearly see that as the dimensionality is low the error is high, as I said in the section Feature Space <= 784 This is because we lose information in our data and certain digits become harder to recognize over others. As we increase the dimensionality past our baseline 784 (718 after removing the 0's column) gets better and better but there becomes a point where the accuracy of the training data and the accuracy of the testing data diverges, I would say alittle before a dimensionality of 1000. At this point I think overfitting becomes an issue and our model is overfit on the training data. In the final test case I ran with a feature space dimension of above 4000, the data was overfit to the point that when the model is run on the training set it has 0 error on 3 digits!

```
Error is 0.0007500000000000284



                 Accuracy
Accuracy for 0        1.0
Accuracy for 1        1.0
Accuracy for 2   0.998993
Accuracy for 3   0.998206
Accuracy for 4   0.998973
Accuracy for 5   0.998524
Accuracy for 6        1.0
Accuracy for 7   0.999521
Accuracy for 8   0.999316
Accuracy for 9   0.998823
```

Figure 74: Training Error above L = 4000

Lets compare this to the testing data in the same dimension feature space.

```
Error is 0.022599999999999953



                  Accuracy
Accuracy for 0   0.992857
Accuracy for 1   0.996476
Accuracy for 2   0.978682
Accuracy for 3   0.974257
Accuracy for 4   0.973523
Accuracy for 5   0.971973
Accuracy for 6   0.987474
Accuracy for 7   0.969844
Accuracy for 8   0.969199
Accuracy for 9   0.957384
```

Figure 75: Training Error above L = 4000

Our error still remains low but the difference between the training and testing dataset is very large, wherever we has 100% accuracy has now dropped to the 99% or 98% mark, it is a clear sign of overfitting if you have a decrease in accuracy from the training to the testing especially when it run perfectly on the training dataset. Its also apparent when you divide the error rates and see that the testing data has 30 times more errors than the testing set, even know 30 times a small number is small, we would be shocked to see this error if we only knew the accuracy on the trainign set beforehand

0.7.4 Function: Dimension Increasing Loop

I modified the code slightly from the original test run, this run was changing the dimensionality (L) of the feature space by 50, starting from 50 up until 5000. I have change it to be from 100 to 2500 I have it commented out in the code if you want to run it, its at the end of the code.

```
    for index,value in enumerate(L_features):

        new_train_x,new_test_x = change_the_set(train_x,test_x,value,function_feature)


        new_test_ones = np.ones((new_test_x.shape[0],1))
        new_train_ones = np.ones((new_train_x.shape[0],1))

        new_train_x = np.append(new_train_x,new_train_ones,axis = 1)

        new_test_x = np.append(new_test_x,new_test_ones,axis = 1)

        guesses_featured_train = run_ovo_all(new_train_x,train_y,new_train_x,train_y)

        error_train[index] = analyze_multi_class(guesses_featured_train, train_y)



        guesses_featured_test = run_ovo_all(new_train_x,train_y,new_test_x,test_y)

        error_test[index] = analyze_multi_class(guesses_featured_test, test_y)
        plt.subplot(2, 1, 1)
        plt.plot(L_features, error_train)
        plt.title("training data")
        plt.xlabel("Dimensions")
        plt.ylabel("Error")
        plt.subplot(2, 1, 2)
        plt.plot(L_features, error_test)
        plt.title("test data")
        plt.xlabel("Dimensions")
        plt.ylabel("Error")
        plt.show()

```

Figure 76: Function: Plot Error Vs Dimension of Feature Space

0.8 Increasing Feature Space for One Vs All Multiclass

next we will cover how increasing the feature space affects our one vs all multiclass classifier with respect to each feature space function. As a reminder our functions are

The identity function, defined as:

$$f(x) = x$$

The sigmoid function, defined as:

$$f(x) = \frac{1}{1 + e^{-x}}$$

The sinusoidal function, defined as:

$$f(x) = \sin(x)$$

The RELU function, defined as:

$$f(x) = \max(x, 0)$$

0.8.1 Identity Function One Vs All

below is our error graph, where error is graphed with respect to the dimension of the feature space.

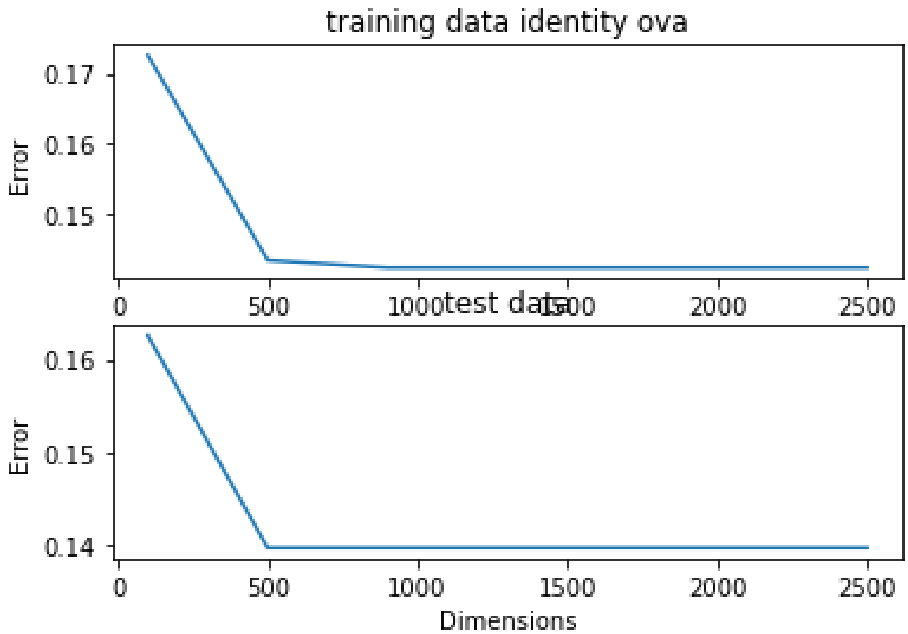

Figure 77: Plot: Identity test and training error vs Dimension

Below L = 784 there is an increase in error, this happens when we try to fit all our data onto a lower dimensional space there is some degeneration of our data, the data cannot be reconstructed as it has lost some of its when mapping to lower dimensions. There is no difference for the identity function above L = 784 however. I believe this is because we do not actually change the function in the feature space through our g(x) = x and there is no new space to draw a line to create a better fit.

Matplotlib for some reason takes the second plots title and overlaps it with the x axis of the first, the top and bottom graphs have the same x axis its just harder to see the top (training) one.

0.8.2 Sigmoid Function One Vs All

As we can see the error rate is much higher for dimensions below the input space, in this case we can understand this as a loss of information when decreasing the dimension. Similarly to the ReLU function we see an increase of both training and testing data but as we take our dimension too high we can see that the training set error and testing set error diverges. This divergence is ”over-fitting” where our training set becomes too accurately fit and it starts to fit only on training and the testing set error doesnt change. This affects both of the next sections, for Sigmoid Sinusoid and ReLU we see the same trend that exists, the trend might be at different error rates but the same trend runs for all of them. The graphs are labelled below.

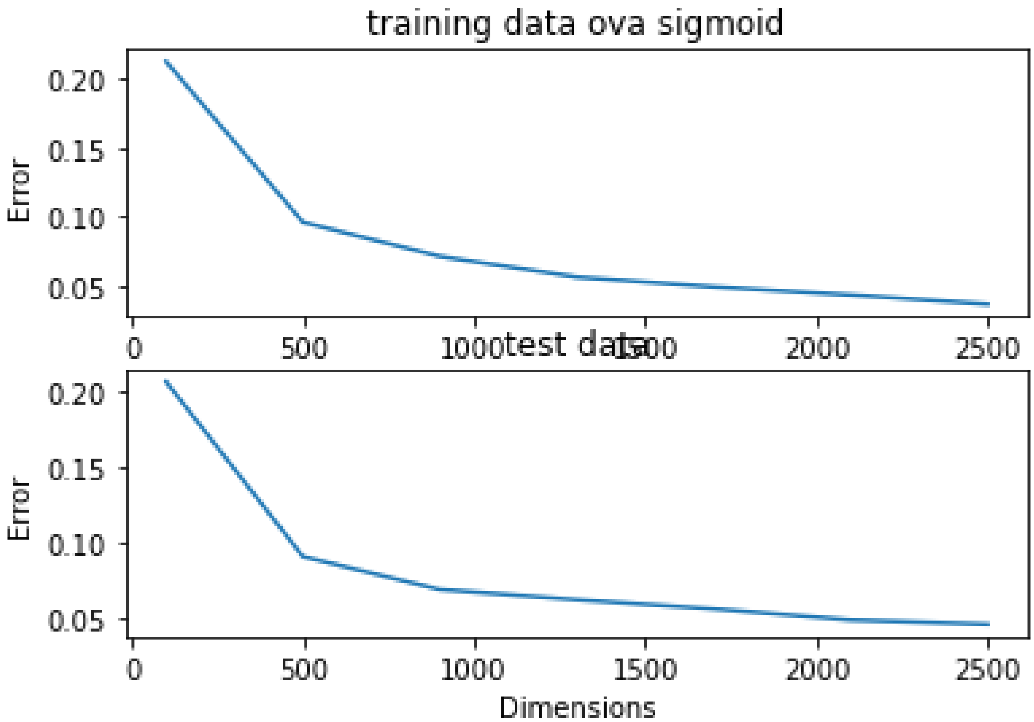

Figure 78: Plot: Error Vs Dimension of Feature Space

0.8.3 Sinusoidal Function One Vs All

Figure 79: Plot: Error Vs Dimension of Feature Space

0.8.4 ReLU Function One Vs All

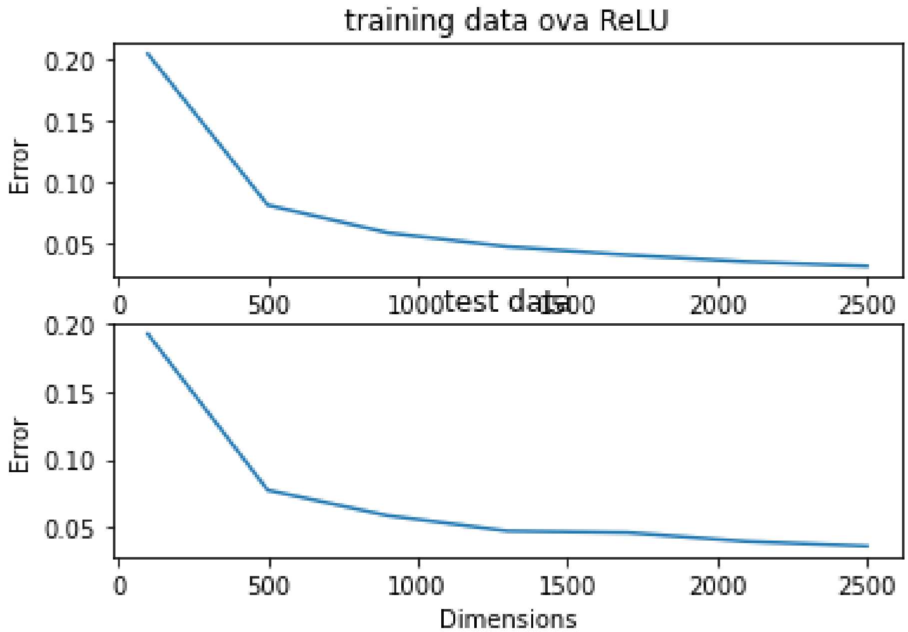

Figure 80: Plot: Error Vs Dimension of Feature Space

As we can see from all of the one vs all classifiers, the error rate between the training and the testing remains essentially the same, I believe we would need to reach a much higher dimension (too computationally expensive for my computer) to notice and overfitting.

The next section we will cover the One Vs One multiclass classifier and the effect of the feature space dimensionality on the traing and test error.

0.9 Increasing Feature Space for One Vs One Multiclass

We move onto the One Vs One multiclass classifier, here I expect the greater accuracy of the One Vs One classifier to begin overfitting on the data, I believe the addition of all 45 of the One Vs One classifiers will give enough flexibility for the model to overfit in a more noticeable way in the higher dimensional feature spaces.

0.9.1 Identity Function One Vs One

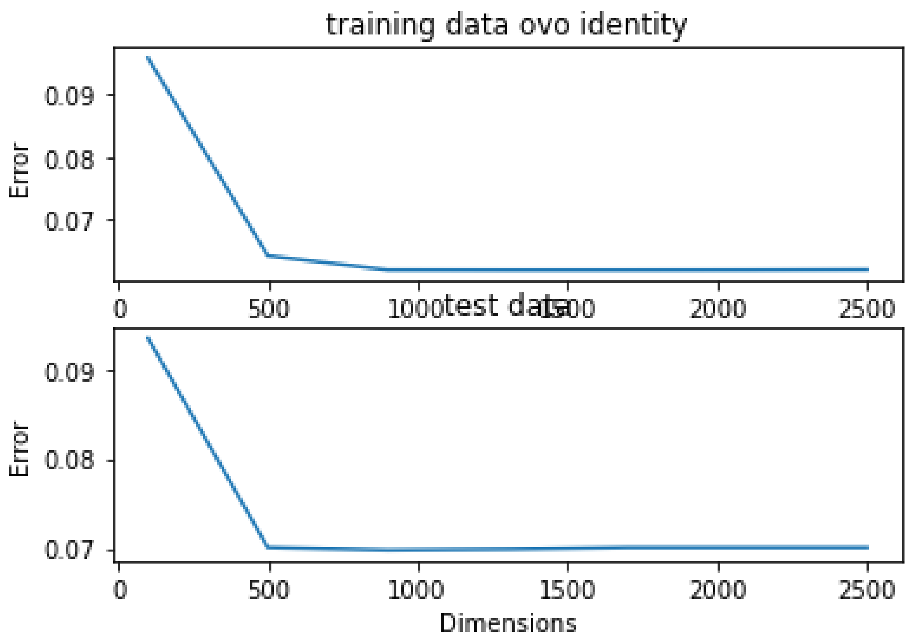

Figure 81: Plot: Error Vs Dimension of Feature Space

The Identity Function in the One vs One multiclass has a very similar plot compared to the One vs All multiclass. We see that there is almost a complete plateau as we increase the dimensionality, as expected the error rate is lower than the One vs All but the plateau happens at around the same dimensionality of the feature space.

0.9.2 Sigmoid Function One Vs One

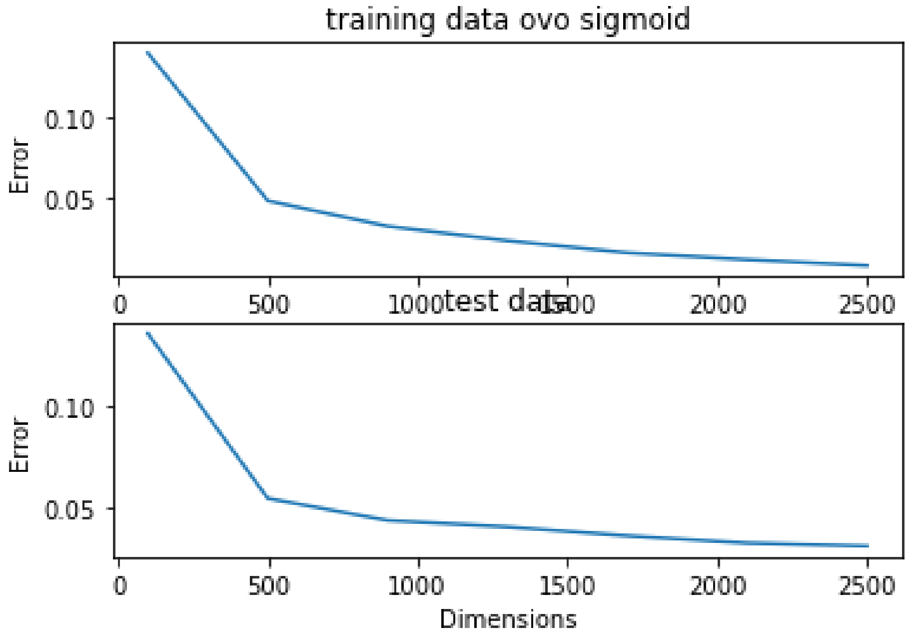

Figure 82: Plot: Plot Error Vs Dimension of Feature Space

Similar to the Multiclass One Vs All classifier we see that all 3 of our last functions have the same trend, when our feature space is lower dimension than the input space the error increases because we lose information in the training set. The sigmoid function doesnt appear to have any serious overfitting below a feature space of dimension 2500 I believe if we increased the dimension high enough we would see overfitting but it would be too computationally expensive to achieve that graph.

However I did run a single test at dimension 4000 to test if there does exist overfitting. As we can see below there is some overfitting when reaching such a high dimensional feature space.

```
Error is 0.0020666666666666661


                  Accuracy
Accuracy for 0  0.999493
Accuracy for 1  0.999703
Accuracy for 2  0.998489
Accuracy for 3  0.994128
Accuracy for 4  0.998802
Accuracy for 5  0.997233
Accuracy for 6  0.998986
Accuracy for 7  0.998244
Accuracy for 8  0.997949
Accuracy for 9  0.996134
```

Figure 83: training data L = 4000 sigmoid

```
Error is 0.028599999999999996


                  Accuracy
Accuracy for 0  0.990816
Accuracy for 1  0.994714
Accuracy for 2  0.968992
Accuracy for 3  0.970297
Accuracy for 4  0.970468
Accuracy for 5  0.954036
Accuracy for 6  0.982255
Accuracy for 7  0.971790
Accuracy for 8  0.950719
Accuracy for 9  0.955401
```

Figure 84: testing data L = 4000 sigmoid

I do not know why it takes so much longer for the sigmoid to overfit than that of the ReLU and the sinusoidal function, I will research into this in the future. **I am leaving the above line to show my mistake and the importance to pay close attention to the data presented to you.** The sigmoid **IS** overfitting if we pay closer attention to the y axis we can see that the lowest point on our graph for the testing data is an error rate near 0, however in the testing set the bottom of our graph is probably closer to .1-.2 This is some strange feature of matplotlib which I do not like, in future project I will either find a way to remedy this or use a new library

0.9.3 Sinusoid Function One Vs One

Figure 85: Plot: Error Vs Dimension of Feature Space

As L (feature space dimension) increases we see overfitting as we increase past our input space around L = 1000 the training set and the testing set diverges and the testing set asymptotes around a non 0 error while the testing set approaches 0 error. We see the same issue with the ReLU function below (Along with the ReLU deep dive simulation up to high dimensions I covered previously)

0.9.4 ReLU Function One Vs One

The matplotlib function makes a weird decision to not reduce the size of the y axis on the top graph so I will also place my ReLU function plot from before with high resolution and dimensionality. The top matplotlib has the bottom set right next to the x axis (0 on y axis) but the bottom one has the bottom at .025 error. Very strange choice of the program.

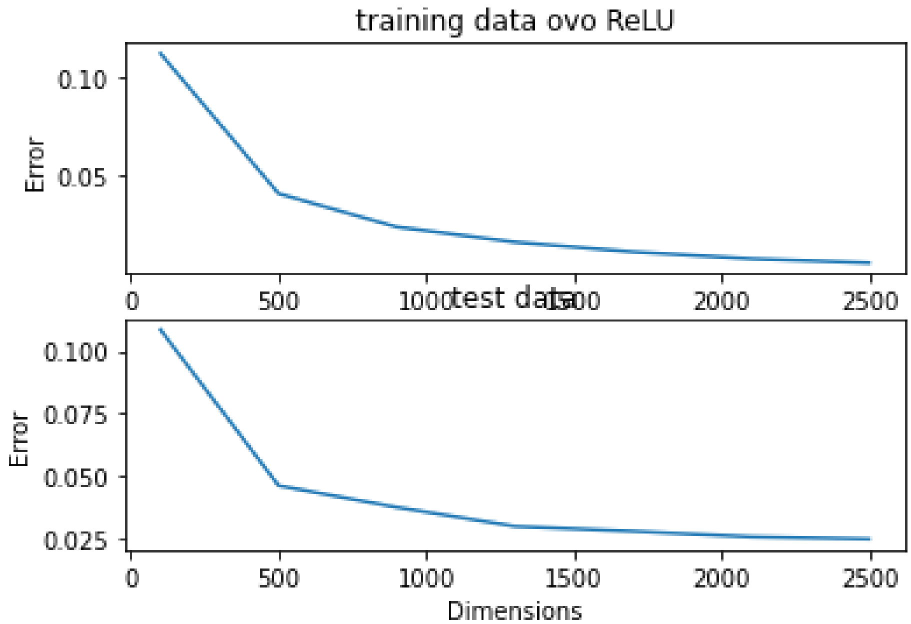

Figure 86: Plot: Error Vs Dimension of Feature Space

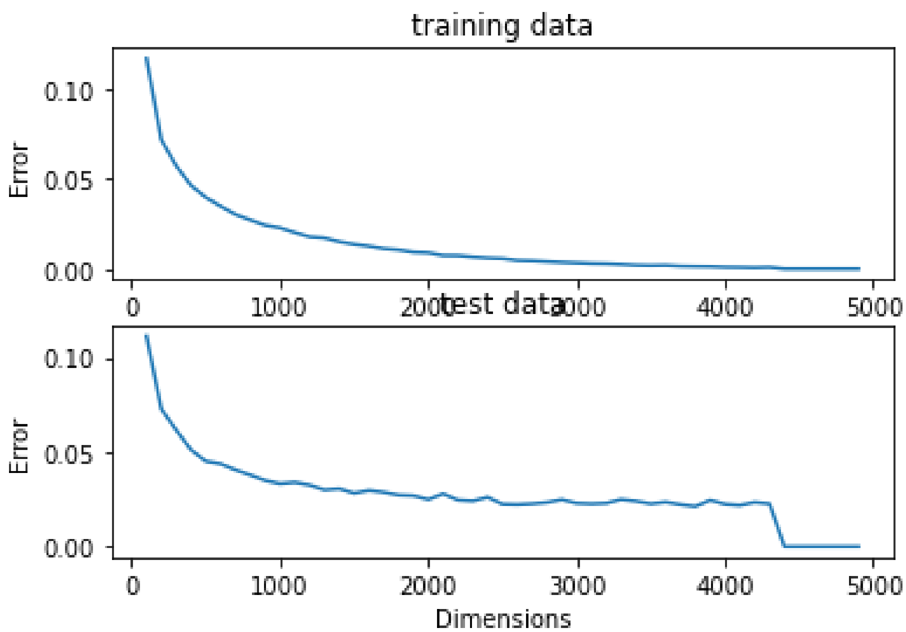

Figure 87: Plot: Error Vs Dimension of Feature Space

Here we can clearly see the issue our training data having near 0 error when the testing has a much more noticeable amount of error.

0.10 Conclusion

In this MiniProject I created a code that takes in handwritten digits from the mnist dataset. These images are all 28x28 images and using Linear Least Squares and the Pseudo Inverse I created a Linear Regression algorithm which can accurately determine a digit (0-9) from the testing dataset by applying a least squares fit to the training dataset. We then explored feature engineering and the feature space where we applied different functions to see if we could still predict handwritten digits only to find that it became easier!(under some conditions)