ECE 174 Mini Project Report.

Rahman Hajiyev A17439304

Introduction

This report presents the implementation and analysis of two multi-class classifiers based on the Least Squares method, applied to the MNIST dataset of handwritten digits. The focus is on the design and evaluation of both one-versus-one and one-versus-all classifiers.

Dataset

The MNIST dataset, consisting of 60,000 training images and 10,000 test images of handwritten digits, was used. Each grayscale image of 28x28 pixels is represented as a 784-dimensional vector. Prior to analysis, the images were normalized to the [0, 1] range

Problem 1.

In this problem we are tasked with creating two types of classifiers out of binary classifiers, one-versus-one classifier and one-versus-all classifier.

Binary class:

This is fundamental building block of this project, we use these to create complex classifiers. This operate in this way: the main idea is it decides whether the input data/image is more likely be part of class a or b, this is really useful to build multi-object classifiers.

One versus all:

In this classifier we use binary classifier to compare a number with rest of numbers for each possible number meaning we only have K classifiers. The final prediction is gained by votes

One versus one

Here, we design binary classifiers for every pair of classes, resulting in

$$K(K-1)/2$$

$K(K-1)/2$ classifiers. The final prediction for a test vector x is based on a voting scheme among these classifiers.

Implementation and Code Integration

Custom Python code was developed to implement both one-versus-one and one-versus-all classifiers. Key strategies and optimizations in the code included:

- Normalizating of data by dividing it by 255.
- Add bias to the data
- Use pseudo inversewith dot product to find correct values for parameters of each binary classifier
- Create a voting mechanism for both of the classifiers

Code explanations:

```
import numpy as np

from scipy.io import loadmat
```

The part above is used to import libraries.

\

```
#Loading data

def load_mat_file(file_path):

   data = loadmat(file_path)

   return data



file_path = 'mnist.mat'

mat_data = load_mat_file(file_path)
```

This is where we load our data to our kernel using spicy.io

Designing one versus one

Problem 1.2 1. (i) <- one-versus-one classifier

```
def all_pairs():
    return [[j, i] for i in range(1, 10) for j in range(0, i)]

def data_for_each_pair(pair, mat_data):
    print(pair)
    smaller_data_x = []
    smaller_data_y = []
    for x, y in zip(mat_data['trainX'], mat_data['trainY'][0]):
        # Check if the current label y is one of the pair elements
        if y in pair:
            normalized_x = x / 255

            x_with_bias = np.append(normalized_x, 1)

            # Add the processed feature vector to the list
            smaller_data_x.append(x_with_bias)

            # Determine the label for this pair and add to the list
            # Label is 1 if y matches the first element of the pair, else -1
            label = 1 if y == pair[0] else -1
            smaller_data_y.append(label)

    # Return the processed feature vectors and corresponding labels
    return smaller_data_x, smaller_data_y

def compute_theta(X, Y):
    X, Y = np.array(X), np.array(Y)
    X_transpose = X.T
    return np.dot(np.linalg.pinv(np.dot(X_transpose, X)), np.dot(X_transpose, Y))

def one_versus_one():
    return {tuple(pair): compute_theta(*data_for_each_pair(pair, mat_data)) for pair in all_pairs()}
```

This is code for creating function to get parameters for onetoone classifier.

Here: all_pairs output all the possible number pairs whose length determines our number of binary classifiers for this model.

Data for each pair look at each pari and output labeled data according to it, this add bias as well like:

$$f = \text{sign} \circ g$$

$$: \mathbb{R} \longrightarrow \{-1, +1\} \text{ is a function defined b}$$

$$\text{sign}(x) = \begin{cases} 1, & \text{if } x \geq 0 \\ -1, & \text{otherwise} \end{cases}$$

$$\tilde{\mathbf{x}} = \begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix}, \text{ and } \quad \boldsymbol{\theta} = \begin{bmatrix} \boldsymbol{\beta} \\ \alpha \end{bmatrix}.$$

Compute theta uses pseudo inverse using equations as provided:

$$\mathbf{X}^T\mathbf{X}\theta = \mathbf{X}^T\mathbf{y}.$$

One versus oen function on the other hadn wraps everything up and outputs dictionary of parameters.

Designini one versus one

```
def data_for_one_vs_all(number):
    data_x, data_y = [], []
    for x, y in zip(mat_data['trainX'], mat_data['trainY'][0]):
        x = x/ 255
        data_x.append(np.append(x, 1))  # Adding bias here
        data_y.append(1 if y == number else -1)
    return data_x, data_y

def one_versus_all():
    parameters = {}
    for number in range(0, 10):
        X, Y = data_for_one_vs_all(number)
        theta = compute_theta(np.array(X), np.array(Y))
        parameters[number] = theta
    return parameters
#################################################################
```

0.0s

Function: `data_for_one_vs_all(number)`

This function prepares the training data for a one-vs-all classifier for a specific digit (denoted by `number`) in the MNIST dataset.

Function: `one_versus_all()`

This function orchestrates the creation of one-vs-all classifiers for each digit in the MNIST dataset.

The main idea and difference from the othe classifier is that in this one while labeling instead of looking at pairs of number, we label 1 is number is likely and -1 if other set is likely which consists of all other digits besides this number.
Here we also use previously defined compute_theta

Evaluating data:

v

```
def error_rate_one_to_one_classifier(datax, datay, parameter_pair_pairs):
    def one_versus_one_classifies(x): …

    num_of_error = sum(1 for x, y in zip(datax, datay[0]) if one_versus_one_classifies(x) != y)
    return num_of_error / len(datay[0])
```

✓ 0.0s

```
def label_error_one_to_one_classifier(datax, datay, parameter_pair_pairs):
    def one_versus_one_classifies(x):
        normalized_x = x / 255

        normalized_x = np.append(normalized_x, 1)
        output = {i: 0 for i in range(10)}
        max_class, max_count = None, -1

        for (smaller, higher), params in parameter_pair_pairs.items():
            predicted = np.dot(params, normalized_x)
            chosen_class = smaller if predicted > 0 else higher
            output[chosen_class] += 1

            if output[chosen_class] > max_count:
                max_class, max_count = chosen_class, output[chosen_class]

        return max_class
    predicted_labels = [one_versus_one_classifies(x) for x in datax]
    # Create a dictionary to store counts of errors and total counts for each label
    label_errors = {}
    net_count = {}

    for real, predicted in zip(datay, predicted_labels):
        if real != predicted:
            label_errors[real] = label_errors.get(real, 0) + 1
        net_count[real] = net_count.get(real, 0) + 1

    # Calculate error rate for each label
    error_rate_per_label = {label: (label_errors.get(label, 0) / max(net_count.get(label, 1), 1)) for label in set(datay)}


    return error_rate_per_label
```

This function calculates the error rate of a one-to-one classifier. It iterates over the dataset, classifying each data point using the one-to-one approach, and compares the predicted labels with the actual labels to count the number of misclassifications. The error rate, a crucial metric for evaluating classifier performance, is then computed as the ratio of misclassified data points to the total number of data points. This function is essential for understanding how well the classifier performs overall on a given dataset.

`error_rate_one_to_one_classifier(datax, datay, parameter_pair_pairs)`

This function calculates the error rate of a one-to-one classifier. It iterates over the dataset, classifying each data point using the one-to-one approach, and compares the predicted labels with the actual labels to count the number of misclassifications. The error rate, a crucial metric for evaluating classifier performance, is then computed as the ratio of misclassified data points to the total number of data points. This function is essential for understanding how well the classifier performs overall on a given dataset.

`label_error_one_to_one_classifier(datax, datay, parameter_pair_pairs)`

This function specifically calculates the error rate for each individual label in a one-to-one classification scenario. It predicts the label for each data point and then compares these predictions with the actual labels to track misclassifications per label. This granular level of error analysis is valuable for identifying which specific digits (or classes) the classifier struggles with, allowing for more targeted improvements in the classifier's design or training process.

```
def confusion_matrix_for_one_to_one(datax, datay, parameter_pair_pairs):
    def one_versus_one_classifies(x, parameter_pair_pairs):
        normalized_x = x / 255
        x = np.append(normalized_x, 1)  # Add bias term
        output = {i: 0 for i in range(10)}

        for (smaller, higher), params in parameter_pair_pairs.items():
            predicted = np.dot(params, x)
            chosen_class = smaller if predicted > 0 else higher
            output[chosen_class] += 1

        return max(output, key=output.get)  # Class with the maximum votes

    con_matrix = np.zeros((10, 10))

    for i, y in enumerate(datay[0]):
        predict = one_versus_one_classifies(datax[i], parameter_pair_pairs)
        con_matrix[y, predict] += 1
    return con_matrix
```

`confusion_matrix_for_one_to_one(datax, datay, parameter_pair_pairs)`

The function constructs a confusion matrix for the one-to-one classifier. It classifies each data point and populates the confusion matrix, which is a 10x10 matrix in the case of the MNIST dataset, with counts of true positives, false positives, true negatives, and false negatives for each class. The confusion matrix is a vital tool for visualizing the classifier's performance, revealing not just the errors but also providing insights into the types of errors (like which digits are often confused with each other).

One to all classifiier

```
def one_versus_all_classify(x, parameters):
    x = x/255
    x = np.append(x, 1)  # Add bias
    predictions = {digit: np.dot(params, x) for digit, params in parameters.items()} # get prediction
    return max(predictions, key=predictions.get) #get max
def error_rate_one_to_all_classifier(datax, datay, parameters):
    num_of_error = 0

    for x, y in zip(datax, datay[0]):
        predict = one_versus_all_classify(x, parameters) # get predict
        if predict != y:
            num_of_error += 1
    return num_of_error / len(datay[0])

##########################################################################################################
```

✓ 0.0s

```
def label_error_rate_one_to_all_classifier(datax, datay, parameters):
    errors = { 0: 0, 1: 0, 2 : 0 , 3: 0, 4: 0, 5: 0, 6 : 0, 7 : 0, 8: 0, 9 : 0}
    total = { 0: 0, 1: 0, 2 : 0 , 3: 0, 4: 0, 5: 0, 6 : 0, 7 : 0, 8: 0, 9 : 0}
    for i, y in enumerate(datay[0]):
        predict = one_versus_all_classify(datax[i], parameters)
        total[y] += 1
        if predict != y:
            errors[y] += 1
    output = { 0: 0, 1: 0, 2 : 0 , 3: 0, 4: 0, 5: 0, 6 : 0, 7 : 0, 8: 0, 9 : 0}
    for i in range(10):
        output[i] = errors[i]/total[i]
    return output
```

`data_for_one_vs_all(number)`

- **Purpose**: This function is designed to prepare the data for training a specific binary classifier within the one-vs-all framework. In a one-vs-all setting, you have multiple classifiers, each trained to recognize one specific class (in this case, a digit from 0 to 9) against all other classes.
- **Functionality**: It takes the digit (`number`) for which the classifier is to be trained and processes the training data. Each image in the MNIST dataset (represented as a vector of pixel values) is normalized (scaled to a range between 0 and 1). A bias term is added to this vector, which is a common practice in machine learning to help the learning algorithm perform better. The labels are converted to a binary format where the label is 1 if it matches the `number` and -1 otherwise. This binary labeling is key to training a classifier that focuses on distinguishing one specific digit from all others.

`one_versus_all()`

- **Purpose**: This function implements the overarching one-vs-all strategy by creating and storing a series of binary classifiers, each trained to recognize a different digit.
- **Functionality**: It iterates over each digit from 0 to 9. For each digit, it calls `data_for_one_vs_all` to prepare the corresponding dataset. It then computes the parameters (theta) for the binary classifier for that digit using a learning algorithm (presumably in `compute_theta`). These parameters are stored in a dictionary, with each entry corresponding to a different digit. The result is a collection of classifiers, each tuned to identify one specific digit from the MNIST dataset.

RESULT

One-Versus-One Classifier

- **Training Data:**
  - Low error rates for labels 0, 1, 6 (around 2.01%, 1.79%, 3.85% respectively).
  - High error rates for labels 2, 5, 8 (around 7.36%, 8.41%, 12.12% respectively).
  - Confusion matrix shows misinterpretations, particularly between digits like 3 and 5, or 8 and 9.
- **Testing Data:**
  - Overall error rate is 7.03%.
  - Confusion matrix on test data shows similar confusion patterns as training data.

One-Versus-All Classifier

- **Training Data:**
  - Low error rates for digits 0, 1, 6 (around 4.07%, 2.88%, 7.47% respectively).

    - High error rates for digits 2, 5, 8 (around 19.57%, 26.38%, 24.59% respectively).
    - Confusion matrix indicates difficulties in differentiating digits, especially 2, 5, and 8.
- **Testing Data:**
    - Overall error rate is 13.97%.
    - Confusion matrix reflects similar challenges in classification as in training data.

General Observations

- **Performance Consistency:** Both classifiers show consistency in performance from training to testing datasets.
- **Better Accuracy:** The One-Versus-One classifier generally shows better accuracy and lower error rates compared to the One-Versus-All classifier.
- **Challenging Digits:** Digits 0, 1, and 6 are classified more accurately, while 2, 5, and 8 are more challenging for both classifiers.

Data: `error_rate of one to one classifier is0.062233333333333335`

```
label data rate that show for each digit how much the model is confused on training
data for onetoone model->{0: 0.020091170015195003, 1: 0.017947196677543756, 2:
0.07368244377307821, 3: 0.09036046321970315, 4: 0.044505306401917154, 5:
0.08411732152739347, 6: 0.03852652923284894, 7: 0.05778132482043097, 8:
0.12117586737309861, 9: 0.08169440242057488}
```

```
confusion marix of onetoone on train[[5.806e+03 3.000e+00 1.500e+01 8.000e+00
1.100e+01 1.900e+01 2.200e+01

  6.000e+00 3.200e+01 1.000e+00]

 [2.000e+00 6.623e+03 3.600e+01 1.700e+01 7.000e+00 1.600e+01 2.000e+00

  1.100e+01 2.100e+01 7.000e+00]

 [5.100e+01 6.800e+01 5.522e+03 4.900e+01 5.700e+01 2.000e+01 4.200e+01
```

```
  4.400e+01 9.200e+01 1.300e+01]

 [2.600e+01 4.200e+01 1.190e+02 5.579e+03 9.000e+00 1.610e+02 1.800e+01

  4.800e+01 9.000e+01 3.900e+01]

 [1.400e+01 1.800e+01 2.000e+01 5.000e+00 5.586e+03 1.100e+01 1.400e+01

  1.600e+01 8.000e+00 1.500e+02]

 [4.400e+01 4.800e+01 3.900e+01 1.380e+02 2.200e+01 4.968e+03 9.300e+01

  1.000e+01 4.600e+01 1.300e+01]

 [2.700e+01 1.600e+01 3.600e+01 2.000e+00 3.200e+01 8.400e+01 5.690e+03

  0.000e+00 3.000e+01 1.000e+00]

 [1.000e+01 7.600e+01 5.300e+01 7.000e+00 6.900e+01 9.000e+00 0.000e+00

  5.881e+03 5.000e+00 1.550e+02]

 [3.500e+01 1.950e+02 4.200e+01 1.070e+02 4.800e+01 1.420e+02 3.700e+01

  2.500e+01 5.155e+03 6.500e+01]

 [2.200e+01 1.400e+01 1.700e+01 8.200e+01 1.550e+02 3.000e+01 3.000e+00

  1.370e+02 3.100e+01 5.458e+03]]



Overall error rate of one to all on train0.14226666666666668



confusion marix of onetoall on train[[5.682e+03 7.000e+00 1.800e+01 1.400e+01
2.400e+01 4.300e+01 6.400e+01

  4.000e+00 6.100e+01 6.000e+00]

 [2.000e+00 6.548e+03 4.000e+01 1.500e+01 1.900e+01 3.100e+01 1.400e+01

  1.200e+01 5.500e+01 6.000e+00]
```

[9.900e+01 2.640e+02 4.792e+03 1.490e+02 1.080e+02 1.100e+01 2.340e+02

 9.100e+01 1.920e+02 1.800e+01]

[4.200e+01 1.670e+02 1.760e+02 5.158e+03 3.200e+01 1.250e+02 5.600e+01

 1.150e+02 1.350e+02 1.250e+02]

[1.000e+01 9.900e+01 4.200e+01 6.000e+00 5.212e+03 5.000e+01 3.900e+01

 2.300e+01 5.900e+01 3.020e+02]

[1.640e+02 9.500e+01 2.800e+01 4.320e+02 1.050e+02 3.991e+03 1.920e+02

 3.600e+01 2.350e+02 1.430e+02]

[1.080e+02 7.400e+01 6.100e+01 1.000e+00 7.000e+01 9.000e+01 5.476e+03

 0.000e+00 3.500e+01 3.000e+00]

[5.500e+01 1.890e+02 3.700e+01 4.700e+01 1.700e+02 9.000e+00 2.000e+00

 5.426e+03 1.000e+01 3.200e+02]

[7.500e+01 4.930e+02 6.300e+01 2.260e+02 1.050e+02 2.210e+02 5.600e+01

 2.000e+01 4.412e+03 1.800e+02]

[6.800e+01 6.000e+01 2.000e+01 1.170e+02 3.710e+02 1.200e+01 4.000e+00

 4.920e+02 3.800e+01 4.767e+03]]

label data rate that show for each digit how much the model is confused on training data for onetoall model {0: 0.04068884011480669, 1: 0.02877484425986354, 2: 0.19570325612621686, 3: 0.15870167998695156, 4: 0.10783978089695309, 5: 0.2637889688249401, 6: 0.07468739438999662, 7: 0.13391859537110934, 8: 0.24594086480943428, 9: 0.19868885526979324}

Overall error rate of one to one on test0.0703

label data rate that show for each digit how much the model is confused on test data for onetoone model{0: 0.019387755102040816, 1: 0.013215859030837005, 2: 0.09302325581395349, 3: 0.08316831683168317, 4: 0.0539714867617108, 5: 0.1031390134529148, 6: 0.05219206680584551, 7: 0.06809338521400778, 8: 0.13963039014373715, 9: 0.08721506442021804}

confusion marix of onetoone on test[[9.61e+02 0.00e+00 1.00e+00 1.00e+00 0.00e+00 6.00e+00 8.00e+00 3.00e+00

  0.00e+00 0.00e+00]

 [0.00e+00 1.12e+03 3.00e+00 3.00e+00 1.00e+00 1.00e+00 4.00e+00 1.00e+00

  2.00e+00 0.00e+00]

 [9.00e+00 1.80e+01 9.36e+02 1.20e+01 1.00e+01 5.00e+00 1.00e+01 1.00e+01

  2.20e+01 0.00e+00]

 [9.00e+00 1.00e+00 1.80e+01 9.26e+02 2.00e+00 2.00e+01 1.00e+00 7.00e+00

  2.10e+01 5.00e+00]

 [2.00e+00 4.00e+00 6.00e+00 1.00e+00 9.31e+02 1.00e+00 7.00e+00 4.00e+00

  3.00e+00 2.30e+01]

 [7.00e+00 5.00e+00 3.00e+00 3.00e+01 8.00e+00 8.00e+02 1.70e+01 2.00e+00

  1.50e+01 5.00e+00]

 [6.00e+00 5.00e+00 1.20e+01 0.00e+00 5.00e+00 1.90e+01 9.08e+02 1.00e+00

  2.00e+00 0.00e+00]

 [1.00e+00 1.60e+01 1.70e+01 3.00e+00 1.10e+01 1.00e+00 0.00e+00 9.55e+02

  1.00e+00 2.30e+01]

 [7.00e+00 1.70e+01 8.00e+00 2.30e+01 1.00e+01 3.60e+01 1.00e+01 1.00e+01

  8.40e+02 1.30e+01]

[6.00e+00 5.00e+00 1.00e+00 1.10e+01 3.00e+01 1.20e+01 0.00e+00 2.10e+01

3.00e+00 9.20e+02]]

Overall error rate of one to all on test0.1397

coonfusion matrix of one to all on test[[9.440e+02 0.000e+00 1.000e+00 2.000e+00
2.000e+00 7.000e+00 1.400e+01

2.000e+00 7.000e+00 1.000e+00]

[0.000e+00 1.107e+03 2.000e+00 2.000e+00 3.000e+00 1.000e+00 5.000e+00

1.000e+00 1.400e+01 0.000e+00]

[1.800e+01 5.400e+01 8.130e+02 2.600e+01 1.500e+01 0.000e+00 4.200e+01

2.200e+01 3.700e+01 5.000e+00]

[4.000e+00 1.700e+01 2.300e+01 8.800e+02 5.000e+00 1.700e+01 9.000e+00

2.100e+01 2.200e+01 1.200e+01]

[0.000e+00 2.200e+01 6.000e+00 1.000e+00 8.810e+02 5.000e+00 1.000e+01

2.000e+00 1.100e+01 4.400e+01]

[2.300e+01 1.800e+01 3.000e+00 7.200e+01 2.400e+01 6.590e+02 2.300e+01

1.400e+01 3.900e+01 1.700e+01]

[1.800e+01 1.000e+01 9.000e+00 0.000e+00 2.200e+01 1.700e+01 8.750e+02

0.000e+00 7.000e+00 0.000e+00]

[5.000e+00 4.000e+01 1.600e+01 6.000e+00 2.600e+01 0.000e+00 1.000e+00

8.840e+02 0.000e+00 5.000e+01]

[1.400e+01 4.600e+01 1.100e+01 3.000e+01 2.700e+01 4.000e+01 1.500e+01

1.200e+01 7.590e+02 2.000e+01]

[1.500e+01 1.100e+01 2.000e+00 1.700e+01 8.000e+01 1.000e+00 1.000e+00

7.700e+01 4.000e+00 8.010e+02]]

label data rate that show for each digit how much the model is confused on test data for onetoone model{0: 0.036734693877551024, 1: 0.024669603524229075, 2: 0.21220930232558138, 3: 0.12871287128712872, 4: 0.10285132382892057, 5: 0.26121076233183854, 6: 0.08663883089770355, 7: 0.14007782101167315, 8: 0.22073921971252566, 9: 0.20614469772051536}

Problem 2:

In this problem , the process is really similar to the first problem but instead we you mappings to create features and train them.

RESULTS:

- **Error Rate Trends:** All transformations show a steep decrease in error rate as the dimension L increases, particularly noticeable up to L=200. Beyond this point, the curves start to plateau, indicating diminishing returns on error reduction with increased dimensionality.
- **Performance of Feature Mappings:**
    - **Identity Function:** This function is comparable good model but after 200, this starts to plateau meaning that adding more features does not add value.
    - **Sigmoid Function:** The sigmoid function demonstrates a significant reduction in error rate early on and then plateaus. Its performance is better than the identity function, which may be due to its ability to handle non-linearities in the data.
    - **Sinusoidal Function:** Somehow sinisoidial function performs really bad in this data with highest error rate.
    - **ReLU Function:** The ReLU function exhibits the lowest error rate among the transformations, especially after L=200. This suggests its effectiveness in creating a feature space where the multi-class classifier can better separate the classes.
- **Comparison with Classifier from Problem 1:**
    - **On Training Data:**. ReLU shows the lowest error rates here, meaning that the ReLU-based feature mapping outperforms the classifier from Problem 1 on the training data, if we increase the value of L, because for lower values of L like 200 the error rate for any function is really bad compared to Problem 1, but when we increase it to 1000 L thne Relu or even sigmoid is much better than Problem 1.

- **On Testing Data:** Similarly, if the trend follows from the training data, the ReLU-based feature mapping might generalize better on the testing data, indicated by its lowest error rate in the graph.
- Relu function generalizes best followed by sigmoid identity and sin.
- Adding more features are important helps a lot if L is so low, but adding extensive features overfits data meaning it performs bad on testing data.