Module: EE5907 Pattern Recognition

Assignment: CA2

Student: ROGATIYA Mohmad Aspak Arif

Matriculation ID: A0179741U

Email: e0269760@u.nus.edu

Contents

Introduction

This report is written as part of EE5907 assignment CA2. The assignment is to implement linear and polynomial regression to classify images of handwritten digits 0 to 9. The digit images are obtained from popular MNIST database http://yann.lecun.com/exdb/mnist/. It is a good database for people who want to try learning techniques and pattern recognition methods on real-world data while spending minimum time and efforts on collecting, preprocessing and formatting training and test data.

a) The MNIST Database

Four files are available on MNIST database.

train-images-idx3-ubyte.gz: training set images (9912422 bytes)
train-labels-idx1-ubyte.gz: training set labels (28881 bytes)
t10k-images-idx3-ubyte.gz: test set images (1648877 bytes)
t10k-labels-idx1-ubyte.gz: test set labels (4542 bytes)

The training and test dataset contains 60000 and 10000 handwritten digit images respectively. Each image has 784 pixels (28x28). In other words, the feature data length for each test or training sampleis 784. Each pixel is an 8-bits value, i.e. 0 to 255; where 0 means white and 255 means black.

Load MNIST datasets

The downloaded datasets are in compressed *.gz format. I have uncompressed these files and put under MNIST_Data folder.

Following constants are defined for the files and folder names.

```
FILES_DIR = '.\\MNIST_Data\\'
TRAIN_FILE = 'train-images.idx3-ubyte'
TRAIN_LABEL = 'train-labels-idx1-ubyte'
TEST_FILE = 't10k-images-idx3-ubyte'
TEST_LABEL = 't10k-labels-idx1-ubyte'
```

The Python's numpy module provides np.fromfile to read from *ubyte file as shown below. The datasets "shapes" are highlighted.

```
with open(FILES_DIR + TRAIN_FILE,'rb') as ftemp:
    datatemp = np.fromfile(ftemp, dtype = np.ubyte)
    trainingData=datatemp[16::].reshape(60000,784)
    print('Size of the training set: ',trainingData.shape)

with open(FILES_DIR + TRAIN_LABEL,'rb') as ftemp:
    datatemp = np.fromfile(ftemp, dtype = np.ubyte)
    trainingLabels=datatemp[8::]
    print('Size of the training labels: ',trainingLabels.shape)

with open(FILES_DIR + TEST_FILE) as ftemp:
    datatemp = np.fromfile(ftemp, dtype = np.ubyte)
    testData=datatemp[16::].reshape(10000,784)
    print('Size of the test set: ',testData.shape)

with open(FILES_DIR + TEST_LABEL,'rb') as ftemp:
    datatemp = np.fromfile(ftemp, dtype = np.ubyte)
    testLabels=datatemp[8::]
    print('Size of the test labels: ',testLabels.shape)
```

```
Size of the training set:  (60000, 784)
Size of the training labels:  (60000,)
Size of the test set:  (10000, 784)
Size of the test labels:  (10000,)
```

b) Calculating Eigenvalues and Eigenvectors for Features Extraction

The necessary steps to visualize the topmost 10 eigenvectors are as below.

Step-1: Calculate the mean of training data.

The formula to calculate the **mean Ψ** is as shown below.

$$\Psi = \frac{1}{m}\sum_{i=1}^{m}\Gamma_i$$

Where, m = number of training samples, Γi = training vector

Implementation

- The StandardScaler method from the preprocessing class of the sklearn module is used to standardize the data by removing the mean and scaling to unit variance.
- The mean method from numpy library is used to calculate the mean matrix. The mean matrix shape is 784x1 (same as training samples).

```
trainingStd = StandardScaler().fit_transform(trainingData)
trainingMean = np.mean(trainingStd, axis=0)
print (trainingMean.shape)
```

```
(784,)
```

Step-2: Normalize the matrix and calculate covariance matrix

Normalized matrix can be calculated by subtracting mean from each training sample.

$$\Phi_i = \Gamma_i - \Psi$$

The StandardScaler method from the preprocessing class of the sklearn module is used to normalize the data by removing the mean and scaling to unit variance to generate trainingStd matrix.

$$C = \frac{1}{m}\sum_{i=1}^{m}\Phi_i\Phi_i^T = AA^T \quad (N^2 \times N^2 \text{ matrix})$$

$$\text{where } A = [\Phi_1, \Phi_2, \cdots, \Phi_m] \quad (N^2 \times m \text{ matrix})$$

We can use numpy module's "cov" method on the normalized trainingStd matrix to calculate the covariance matrix. The shape of covariance matrix is 784x784 (N^2xN^2).

```
covMatx = np.cov(trainingStd.T)
print (covMatx.shape)
```

```
(784, 784)
```

Step-3: Calculate eigenvalues and eigenvectors.

I have used the linalg.eig from numpy library to calculate eigenvalues and eigenvectors.

```
eigenVals, eigenVecs = np.linalg.eig(covMatx)
print ("Eigenvals shape: " + str(eigenVals.shape))
print ("Eigenvecs shape: " + str(eigenVecs.shape))
```

```
Eigenvals shape: (784,)
Eigenvecs shape: (784, 784)
```

Step-4: Visualize eigenvectors corresponding to topmost 10 eigenvalues.

First, I have to sort the eigenvectors based on eigenvalues.

```
eigenValVec = [(np.abs(eigenVals[i]), eigenVecs[:,i]) for i in range(len(eigenVals))]
eigenValVec = sorted(eigenValVec, reverse=True, key=lambda x:x[0])
```

Then, I used “imshow” method from matplotlib library to display top 10 eigenvectors.

```
f,ax = plt.subplots(1,10,figsize=(50,50))
for i in range(10):
    ax[i].imshow(eigenValVec[i][1].reshape(28,28))
plt.show()
```

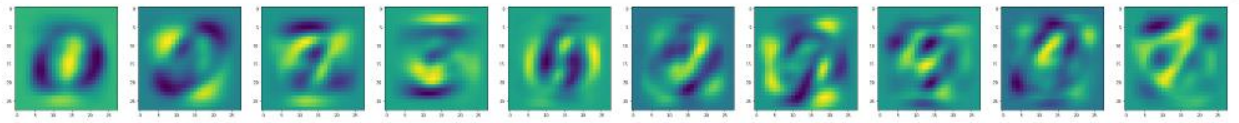


Image Reconstruction:

This section describes the method of displaying selected images from test and training data. It also describes the method to reconstruct the test images using the eigenvectors calculated in the previous section.

(1) First Row: Display selected images from original training set.

I have displayed following columns of the training data: 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 in the first row of the plot. I have prepared a list of these indices.

```
trainDigitList = np.arange(2,21,2)
print (trainDigitList)
```

```
[ 2  4  6  8 10 12 14 16 18 20]
```

(2) Second Row: Display reconstructed images from the test set.

I have displayed reconstructed images corresponding to the following columns of the test data: 4, 3, 2, 19, 5, 9, 12, 1, 62, 8 in second row.

```
testDigitList = np.asarray([4, 3, 2, 19, 5, 9, 12, 1, 62, 8])
print (testDigitList)
```

```
[ 4  3  2 19  5  9 12  1 62  8]
```

Step-1: The λ_i of the test sample is calculated using the following formula**.**

$$\lambda_i = \boldsymbol{x}^T \boldsymbol{v}_i$$

Step-2: The reconstructed test digits was then calculated by the sum of the product between the λ_i and the eigenvectors using following formula.

$$\boldsymbol{x}_p = \sum_{i=1}^{30} \lambda_i \boldsymbol{v}_i$$

Implementation:
Two steps listed above are implemented as following.

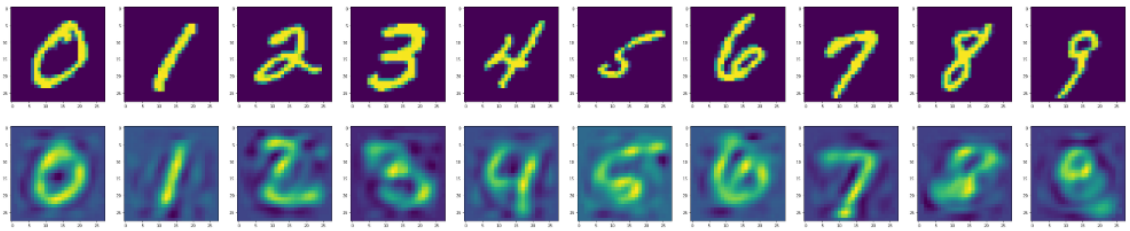
```
reconstructedTestDigits = []
```

```
reconstructedTestDigits = []
for idx, digit in enumerate(testDigitList):
    lambdas = []
    for i in range(TOP_EIGENVECTORS):
        lambdas.append(testData[digit-1].T.dot(eigenValVec[i][1]))
    digitNew = np.zeros(784)
    for i in range(TOP_EIGENVECTORS):
        digitNew += lambdas[i]*eigenValVec[i][1]
    reconstructedTestDigits.append(digitNew)
```

Display First and Second Row:
The selected training digits (first row) and reconstructed selected test digits are plotted using "imshow" method from matplotlib library.

```
f,ax = plt.subplots(1,10,figsize=(50,50))
for idx,digit in enumerate(trainDigitList):
    ax[idx].imshow(trainingData[digit-1].reshape(28,28))
plt.show()

f,ax = plt.subplots(1,10,figsize=(50,50))
for idx in range(10):
    ax[idx].imshow(reconstructedTestDigits[idx].reshape(28,28))
plt.show()
```



c) Using Eigen-digits to Classify each digit image

Project data to 30-dimensional space

- Step-1: Prepare an array of top 30 eigenvectors. This will be used in next steps.

```
top30EigenVecs = []
for i in range(TOP_EIGENVECTORS):
    top30EigenVecs.append(eigenValVec[i][1])
top30EigenVecs = np.asarray(top30EigenVecs)
top30EigenVecs.shape
```

- Step-2: Normalize training and test datasets by subtracting training data mean from each sample.

```
# Standardize training dataset
trainingStd = StandardScaler().fit_transform(trainingData)
# Standardize test dataset
testStd = StandardScaler().fit_transform(testData)

# Calculate mean of the training dataset
trainingMean = np.mean(trainingStd, axis=0)

# Normalize train and test datasets by subtracting training mean.
normedTrain = np.subtract(trainingStd,trainingMean)
normedTest = np.subtract(testStd,trainingMean)
```

- Step-3: Project each training datapoints into the subspace spanned by top 30 eigenvectors.

```
# Project training datapoints into the subspace spanned by top 30 eigenvectors
for idx, digit in enumerate(normedTrain):
    lambdas = []
    for i in range(TOP_EIGENVECTORS):
        lambdas.append(normedTrain[idx].T.dot(eigenValVec[i][1]))
    digitNew = np.zeros(784)
    for i in range(TOP_EIGENVECTORS):
        digitNew += lambdas[i]*eigenValVec[i][1]
    normedTrain[idx] = digitNew

normedTrain = np.dot(normedTrain,top30EigenVecs.T)
print (normedTrain.shape)
```

```
(60000, 30)
```

- Step-4: Project each test datapoints into the subspace spanned by top 30 eigenvectors.

```
# Project test datapoints into the subspace spanned by top 30 eigenvectors
for idx, digit in enumerate(normedTest):
    lambdas = []
    for i in range(TOP_EIGENVECTORS):
        lambdas.append(normedTest[idx].T.dot(eigenValVec[i][1]))
    digitNew = np.zeros(784)
    for i in range(TOP_EIGENVECTORS):
        digitNew += lambdas[i]*eigenValVec[i][1]
    normedTest[idx] = digitNew

normedTest = np.dot(normedTest,top30EigenVecs.T)
print (normedTest.shape)
```

```
(10000, 30)
```

(1) Linear Regression

We will now apply the Linear Regression on training and test datasets and measure accuracy. Linear Regression classifier is implemented by following formula.

$$g(\mathbf{x}) = w_0 + \sum_{i=1}^{d} w_i x_i.$$

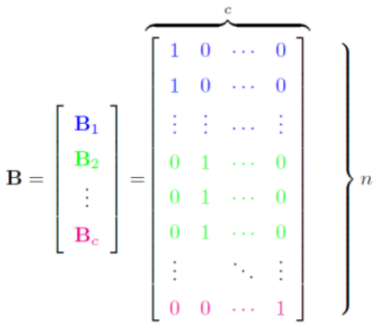
We will first generate a weight matrix A of sized $(d + 1) \times c$, where d is feature dimension and c is number of classes. A matrix is calculated by following formula.

$$\mathbf{A} = \mathbf{Y}^{\dagger}\mathbf{B} = (\mathbf{Y}^t\mathbf{Y})^{-1}\mathbf{Y}^t\mathbf{B}$$

Where, Y matrix looks like below.

$$\mathbf{Y} = \begin{bmatrix} \mathbf{Y}_1 \\ \mathbf{Y}_2 \\ \vdots \\ \mathbf{Y}_c \end{bmatrix} = \begin{bmatrix} y_{11}(\mathbf{x}) & y_{12}(\mathbf{x}) & \cdots & y_{1\hat{d}}(\mathbf{x}) \\ y_{21}(\mathbf{x}) & y_{22}(\mathbf{x}) & \cdots & y_{2\hat{d}}(\mathbf{x}) \\ \vdots & \vdots & \cdots & \vdots \\ y_{31}(\mathbf{x}) & y_{32}(\mathbf{x}) & \cdots & y_{3\hat{d}}(\mathbf{x}) \\ y_{41}(\mathbf{x}) & y_{42}(\mathbf{x}) & \cdots & y_{4\hat{d}}(\mathbf{x}) \\ y_{51}(\mathbf{x}) & y_{52}(\mathbf{x}) & \cdots & y_{5\hat{d}}(\mathbf{x}) \\ \vdots & & \ddots & \vdots \\ y_{n1}(\mathbf{x}) & y_{n2}(\mathbf{x}) & \cdots & y_{n\hat{d}}(\mathbf{x}) \end{bmatrix}$$

B is a one-hot encoded matrix as shown below.



Linear Regression steps can be summarized as below.

Step-1: Calculate Y matrix.

1. We have to sort the training datasets in the order of training labels.

```
sortedNormedTrain = []
for i in range(0,60000,1):
    sortedNormedTrain += (normedTrain[trainingLabels==i].tolist())

sortedNormedTrain = np.asarray(sortedNormedTrain)
print (sortedNormedTrain.shape)
```

```
(60000, 30)
```

2. We will sort the training labels as well.

```
sortedTrainLabels = np.sort(trainingLabels)
print (sortedTrainLabels)
print (sortedTrainLabels.shape)
```

```
[0 0 0 ... 9 9 9]
(60000,)
```

3. Append a bias column at the end of the sorted normalized training data to generate Y matrix.

```
Y = np.append(sortedNormedTrain,np.ones(60000,dtype=int).reshape(60000,1),axis=1)
print (Y.shape)
```

```
(60000, 31)
```

Step-2: Generate B matrix.

1. Generate a zero matrix of size NxC.
2. Put a 1 in each row's class "label" position to get one-hot encoded B matrix.

```
# Create a zero matrix of size NxC (N = # of training data, C = # of class labels in training data)
B = np.zeros((sortedTrainLabels.size, sortedTrainLabels.max()+1))
# Put a 1 in class position in each row, to create a one-hot encoded B matrix
B[np.arange(sortedTrainLabels.size),sortedTrainLabels] = 1
# print B's shape
print ("************************\n B's shape: ", B.shape, "\n************************")
print ("B Matrix:\n",B)
```

Step-3: Generate A matrix.

1. Generate a pseudo inverse of Y matrix using "pinv" method from numpy library.
2. Do a dot multiplication with B to generate A matrix.

```
# Generate a pseudo inverse matrix of Y and then multiply with B to generate A matrix.
A = np.linalg.pinv(Y).dot(B)
print ("************************\n A's shape: ", A.shape, "\n************************")
```

```
************************
 A's shape:  (31, 10)
************************
```

Step-4: Transpose the A matrix.

```
# Get A's transpose matrix, it will be useful in next steps.
AT = A.T
print ("**********************\n AT's shape: ", AT.shape, "\n**********************")
```

Predict training labels using linear regression

Step-1: Run Linear Regression classifier on each training sample for each label and select the class label with highest probability. Prepare a list of predicted labels. We will compare this list against actual training labels to calculate accuracy.

```
# prepare a list of predicted labels. We will compare this list against actual training labels to calculate accuracy.
predictedLabel = []

# Run Linear Regression on each training sample.
for j in range(60000):
    # prepare a list of results for each digit from 0 to 9.
    result = []

    # Loop through each digit label and run Linear Regression. Append the output to "result" list.
    for i in range(10):
        result.append(AT[i].T.dot(Y[j]))

    # Choose the class with highest probability
    predictedLabel.append(result.index(max(result)))

# Covert it to an nparray
predictedLabels = np.asarray(predictedLabel)

print ("**********************************")
print ("predictedLabels's shape: ", predictedLabels.shape)
print ("**********************************")
```

```
**********************************
predictedLabels's shape:  (60000,)
**********************************
```

Step-2: Find total number of accurate results and then calculate accuracy percentage.

```
# Find total number of accurate results
accuratePredictions = np.sum(sortedTrainLabels==predictedLabels)

# Calculate accuracy %
trainAccuracy = (accuratePredictions/60000)*100

# print accuracy %
print ("**************************")
print ("Training accuracy: ", trainAccuracy.round(2))
print ("**************************")
```

```
**************************
Training accuracy:  80.67
**************************
```

Predict test labels using linear regression

Step-1: Generate a Y matrix for test data.

- Similar to Y matrix described for training data.

Step-2: Run Linear Regression classifier on each test sample for each label and select the class label with highest probability. Prepare a list of predicted labels. We will compare this list against actual test labels to calculate accuracy.

- Similar to Linear Regression on training data.

Step-3: Find total number of accurate results and then calculate accuracy percentage.

- Similar to accuracy calculation for training data.

(2) Polynomial Regression

We will now apply the Polynomial Regression on training and test datasets and measure accuracy. The Polynomial Regression classifier is implemented by following formula.

$$g(\mathbf{x}) = w_0 + \sum_{i=1}^{d} w_i x_i + \sum_{i=1}^{d} \sum_{j=1}^{d} w_{ij} x_i x_j$$

Step-1: Generate Y matrix.

Each row of Y matrix has all the terms in the polynomial discriminant function. We can get all these terms (combinations) using combinations_with_replacement method in the itertools module.

```
# Create a list for Y matrix used in Polynomial Regression
poly_Y = []

for y in (Y):
    # Generate combinations of polynomial terms based on polynomial order.
    combinations = np.asarray(list(combinations_with_replacement(y,POLYNOMIAL_ORDER)))
    # Get a product of each combination
    prodCombinations = np.prod(combinations,axis=1)
    # Append each product to polynomial Y matrix
    poly_Y.append(prodCombinations)

# Convert it to nparray type.
poly_Y = np.asarray(poly_Y)

print ("**************************")
print ("Y's shape: ", poly_Y.shape)
print ("**************************")
```

```
**************************
Y's shape:  (60000, 496)
**************************
```

Step-2: Generate B matrix.

Similar to B matrix described in Linear Regression.

Step-3: Generate A matrix.

Similar to A matrix described in Linear Regression.

Step-4: Transpose A matrix.

Similar to A matrix transpose described in Linear Regression.

Predict training labels using Polynomial Regression

Step-1: Run Polynomial Regression classifier on each training sample for each label and select the class label with highest probability. Prepare a list of predicted labels. We will compare this list against actual training labels to calculate accuracy.

```
# prepare a list of predicted labels. We will compare this list against actual training labels to calculate accuracy.
predictedLabel = []

# Run Polynomial Regression on each training sample.
for j in range(60000):
    # prepare a list of results for each digit from 0 to 9.
    result = []

    # Loop through each digit label and run Polynomial Regression. Append the output to "result" list.
    for i in range(10):
        result.append(poly_AT[i].T.dot(poly_Y[j]))

    # Choose the class with highest probability
    predictedLabel.append(result.index(max(result)))

# Covert it to an nparray
predictedLabels = np.asarray(predictedLabel)

print ("**********************************")
print ("predictedLabels's shape: ", predictedLabels.shape)
print ("**********************************")
```

```
**********************************
predictedLabels's shape:  (60000,)
**********************************
```

Step-2: Find the total number of accurate results and then calculate accuracy percentage.

```
# Find total number of accurate results
accuratePredictions = np.sum(sortedTrainLabels==predictedLabels)

# Calculate accuracy %
trainAccuracy = (accuratePredictions/60000)*100

# print accuracy %
print ("***************************")
print ("Training accuracy: ", trainAccuracy.round(2))
print ("***************************")
```

```
***************************
Training accuracy:  94.0
***************************
```

Predict test labels using Polynomial Regression

Step-1: Generate a Y matrix for test data.

- Similar to Y matrix we described for training data

Step-2: Run Polynomial Regression classifier on each test sample for each label and select the class label with highest probability. Prepare a list of predicted labels. We will compare this list against actual test labels to calculate accuracy.

- Similar to Polynomial Regression on training data.

Step-3: Find total number of accurate results and then calculate accuracy percentage.

- Similar to accuracy calculations on training data.

d) Possible Techniques to Improve the Test Accuracy

In general, Polynomial regression has better accuracy and higher execution time compared to Linear Regression.

We can also increase the test accuracy using following techniques.

1) Increase number of Eigenvectors

We can simply modify TOP_EIGENVECTORS constant to modify the number of top eigenvectors.

```
# Define constants for polynomial order and top eigenvectors
POLYNOMIAL_ORDER = 2
TOP_EIGENVECTORS = 30
```

The accuracy improves as we use more than 30 eigenvectors. Following table shows the test accuracy with different eigenvectors. The execution time also increases drastically with increasing number of eigen vectors. Hence, we have to find a balance between accuracy and execution time.

	Linear Regression		Polynomial Regression	
TOP_EIGENVECTORS	Training accuracy	Test accuracy	Training accuracy	Test accuracy
30	80.67%	81.44%	94.00%	93.90%
50	83.25%	84.05%	95.99%	95.50%
70	83.81%	84.71%	96.92%	95.85%
100	84.50%	85.42%	97.88%	96.17%

2) Increase Polynomial Order

We can increase the test accuracy by increasing the Polynomial Order. The execution time also increases dramatically with increasing Polynomial Order. This is because the number of combinations of polynomial terms in the discriminant function increases exponentially with increasing polynomial order. Hence, we have to find a balance between test accuracy and execution time.

We can modify the polynomial order by simply modifying POLYNOMIAL_ORDER constant.

```
# Define constants for polynomial order and top eigenvectors
POLYNOMIAL_ORDER = 2
TOP_EIGENVECTORS = 30
```

Following table shows the test accuracy with different Polynomial orders.

	Polynomial Regression	
POLYNOMIAL_ORDER	Training accuracy	Test accuracy
2	94.00%	93.90%
3	98.09%	96.71%

3) Including Bias term

We can include or remove bias term by simply modifying **INCLUDE_BIAS** constant.

```
# Include or exclude bias term
INCLUDE_BIAS = True
```

It was observed that including Bias term improves the test accuracy slightly.

	Polynomial Regression	
INCLUDE_BIAS	Training accuracy	Test accuracy
With bias	94.00%	93.90%
Without bias	91.80%	91.73%