Multivariate Gaussian & PCA

Ayush Ramteke 210050030
Tanmay Patil 210050156

October 2022

Contents

1 Question 1

AIM: In this question, we aim to transform a uniformly distributed 2D gaussian distribution to get distributions of our desire.
We are going to generate a gaussian distribution that looks like :-

1. An ellipse
2. A triangle

1.1 Algorithm

Approach : We first select random points uniformly distributed in a unit circle and then transform the points into points of required ellipse. This results in each point being distributed uniformly in the ellipse.
In order to get uniform distribution of points in unit circle, we use radial co-ordinate system.
After transforming the radial co-ordinates to cartesian co-ordinate system, we scale the x and y co-ordinates by length of semi-major and semi-minor axes respectively.

Algorithm 1 Algorithm to generate random points

```
$np \leftarrow numpy$
$r \leftarrow np.sqrt(np.random.random())$   ▷ Proof for using sqrt, check $results/ellipse\_sqrt\_proof.pdf$
$\theta \leftarrow np.random.random() * 2 * np.\pi$
$x \leftarrow r * np.cos(\theta) * 1$
$y \leftarrow r * np.sin(\theta) * 0.5$
return$(x, y)$
```

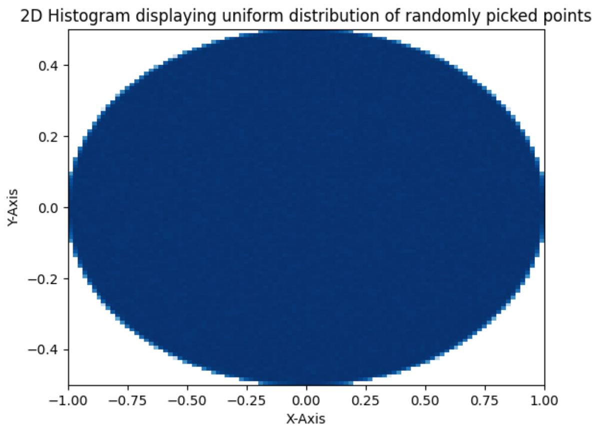
1.2 Plotting the Histogram

1.2.1 Implementation of Algorithm

```python

def uniformRandomPointGenerator_ellipse():
    r = np.sqrt(np.random.random())
    theta = np.random.random()*2*np.pi
    #We have our r and theta, now we convert it to cartesian coordinate system and
    scale the coordinates to
    # that of the ellipse we require(L(major axis)=2, L(minor axis)=1)
    x = r*np.cos(theta)*1
    y = r*np.sin(theta)*0.5
    return (x,y)

sample_ellipse = [] #The list in which our sample data will be stored
for i in range(1, int(1e7)): # Generating 10^7 random points
    point = uniformRandomPointGenerator_ellipse()
    sample_ellipse.append(point)

```

```
sample_ellipse = np.array(sample_ellipse)
```
Listing 1: Code for implementing the algorithm defined in the above subquestion

1.2.2 Histogram



1.3 An algorithm to generate random points uniformly distributed over a triangle

Approach : We first select random points uniformly distributed in a right angled triangle with unit legs, vertices at (0,0), (0,1), (1,1). We then multiply the vector co-ordinates with a transformation matrix to get a uniform distribution of points in the desired triangle.
This transformation matrix actually transforms the right angled unit triangle to the desired triangle, hence all uniformly distributed points get mapped to uniform distribution for the required triangle.

Algorithm 2 Algorithm to generate random points distributed over a triangle

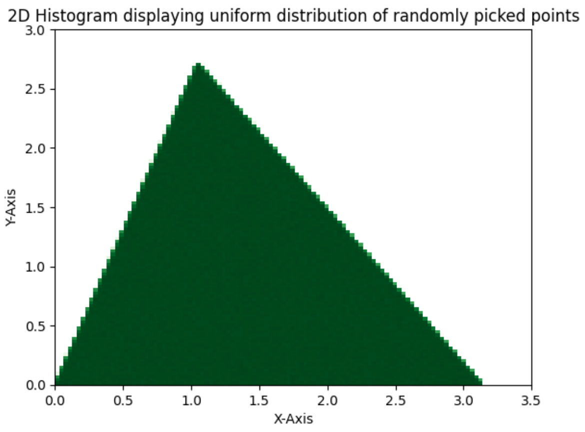
```
$np \leftarrow numpy$
$A \leftarrow np.array([[2 * np.pi/3, np.pi/3], [-np.e, np.e]])$    ▷ Transformation Matrix. How we
                                                                    ▷ got this:-look at $data/Q1\_c\_1$
$x \leftarrow np.random.random()$
$y \leftarrow np.random.random()$
if $x - y > 0$ then
    $x \leftarrow y$                                                ▷ Swapping Values
    $y \leftarrow x$
end if
$point \leftarrow np.array([x, y])$
$point \leftarrow np.matmul(A, point)$
return point
```

1.4 Plotting the Histogram

```python

A = np.array([[2*np.pi/3, np.pi/3],[-np.e, np.e]])

def uniformRandomPointGenerator_triangle():
    x = np.random.random()
    y = np.random.random()
    if x-y >0:
        x, y = y, x
    point = np.array([x, y])
    point = np.matmul(A, point)
    return point

sample_triangle = []
for i in range(1, int(1e7)): # Generating 10^5 random points
    point = uniformRandomPointGenerator_triangle()
    sample_triangle.append(point)

sample_triangle = np.array(sample_triangle)
```

Listing 2: Code for implementing the algorithm defined in the above subquestion

1.4.1 Histogram



2 Question 2

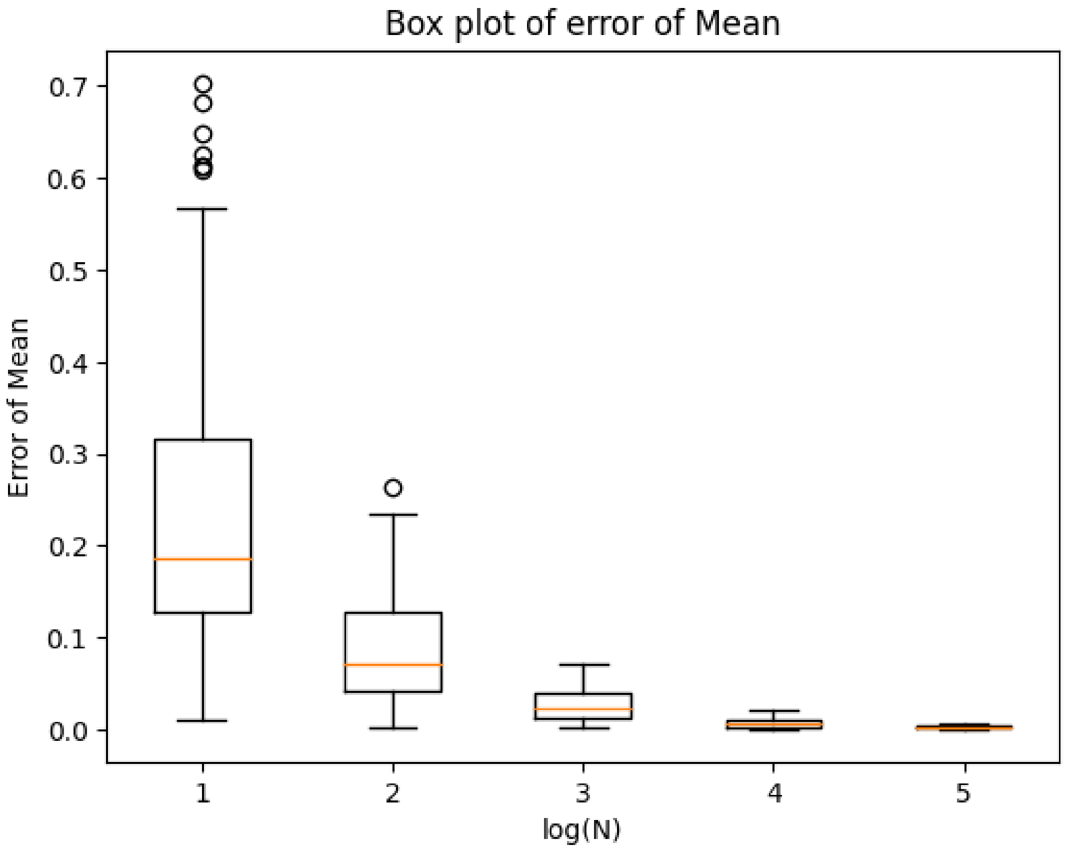
AIM: In this problem, we compute the **maximum-likelihood (ML)** estimates of the mean and the covariance matrix and compare the error of ML estimates and emperical values as a function of **Size of Sample**.

2.1 Generating sample points from 2D Gaussian

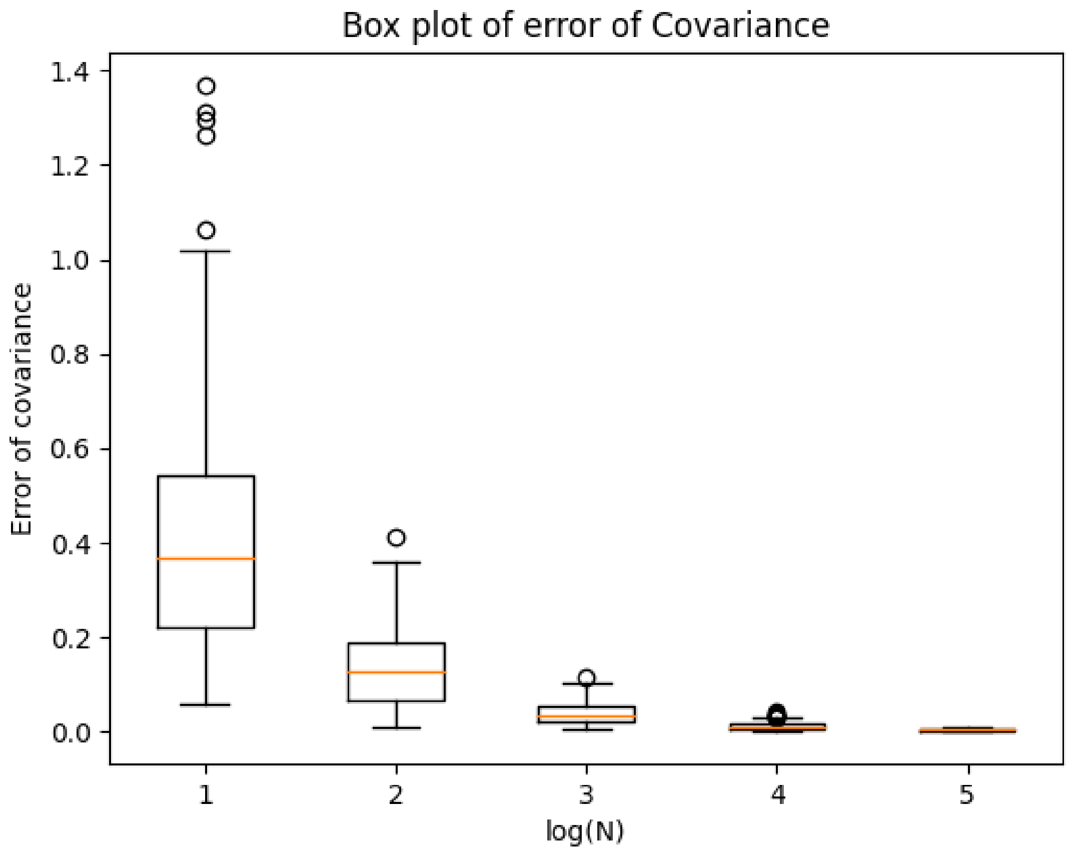
Since we aren't allowed to use in-built functions to generate our 2D Gaussian Distribution, we are going to apply **Transformation of Random Variable** to get it. The steps are:-

1. Generate points of a **Uniform** 2D Gaussian Distribution, and name the random variable as **X**.
2. Multiply the Random Variable with transformation matrix, **A**.
3. Add the mean to shift the transformed Random variable and get desired distribution.

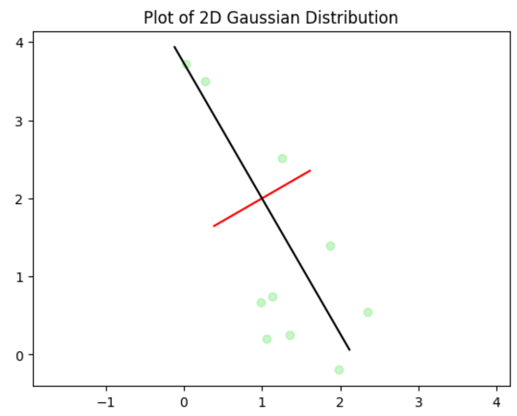
2.2 Boxplot of Error of Mean

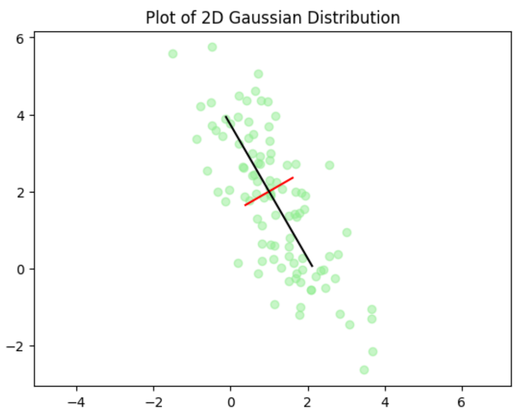


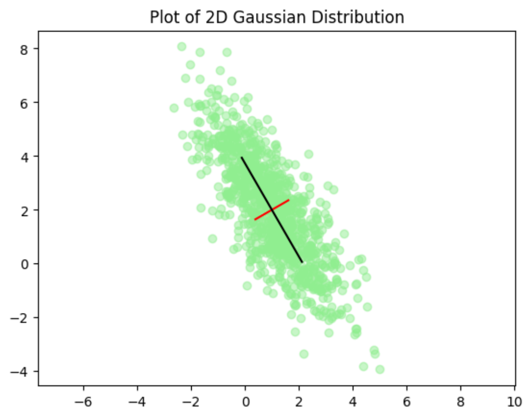
2.3 Boxplot of Error of Covariance

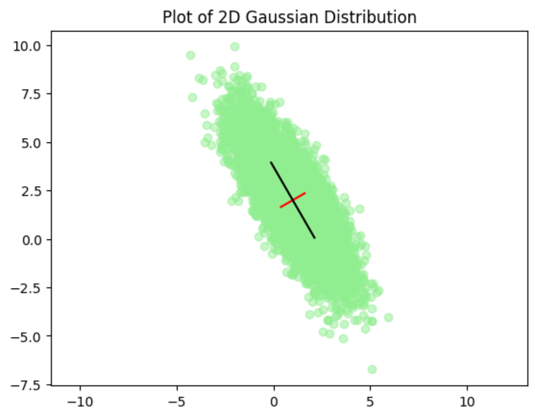


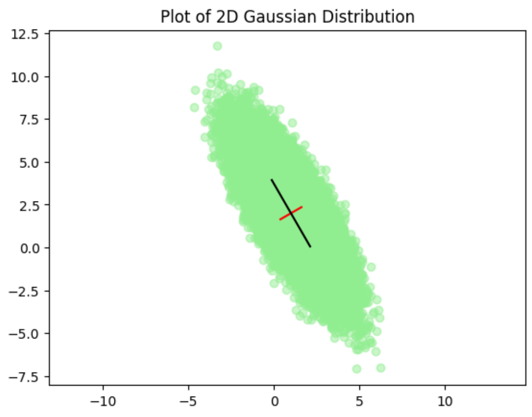
2.4 Plot of 2D Gaussian along with Principal Modes of Variation









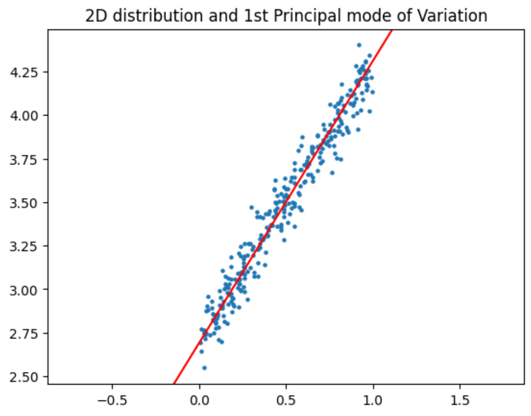


3 Question 3

3.1 Using PCA to best approximate a linear relationship between two random variables

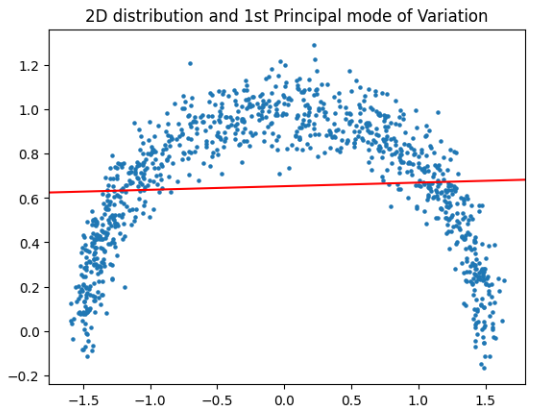
Approach : PCA gives us modes of variation. The principal mode of variation in this case, is a line on which the projection of points gives maximum variance.
Thus, PCA compresses the 2D distribution into 1D distribution with the highest variance. Since the variance is high, more distinct points are observed in 1D and we are able to best approximate a linear relationship between X and Y.

3.2 Overlay of scatter plot of the points and linear relationship line



3.3 Applying the above method for the second set of points

3.3.1 Overlay of scatter plot of the points and linear relationship line



Justification : There is no resemblance between scatter plots and the linear relationship line, as there was with the previous set, because the current set of points do not correspond to a linear relationship between them and the absurd looking line we get was as expected due to the non-linear relationship between the variables.

3.4 Comparing the results

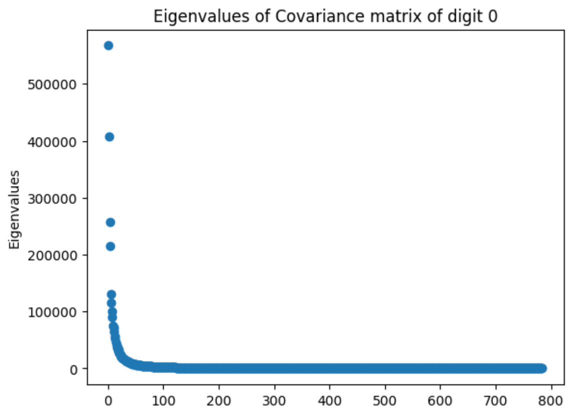
The quality of approximation is better in the first dataset than the second because the first dataset is linear distribution, which makes it easier to obtain a linear relationship between X and Y. On the other hand, the second dataset is a non-linear/curved distribution, which reduces the quality of Linear approximation.

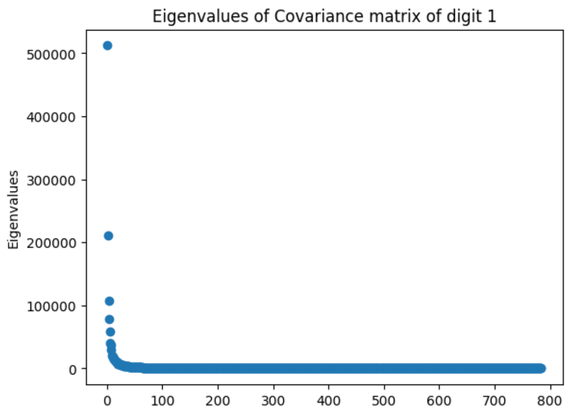
4 Question 4

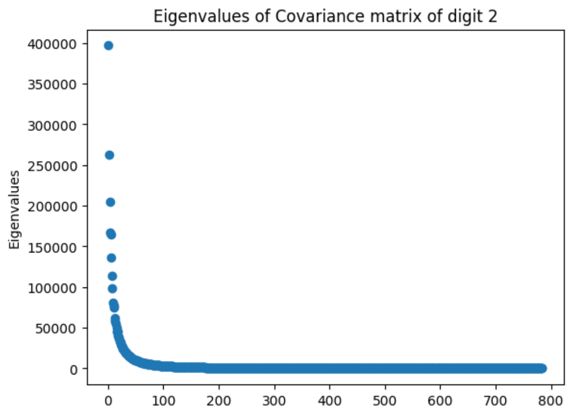
AIM: We took the dataset provided by **MNIST**, which contains hand-drawn images of digits from 0-9, and computed the **mean**, **covariance** and **principal mode of variation** for each of the digits. We also visualised the mean image for each of the digits.

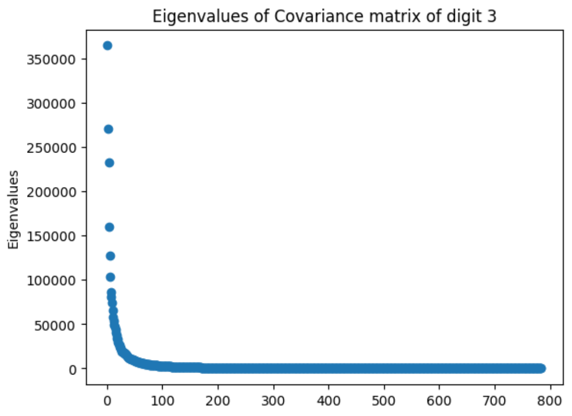
4.1 Significant Eigenvalues

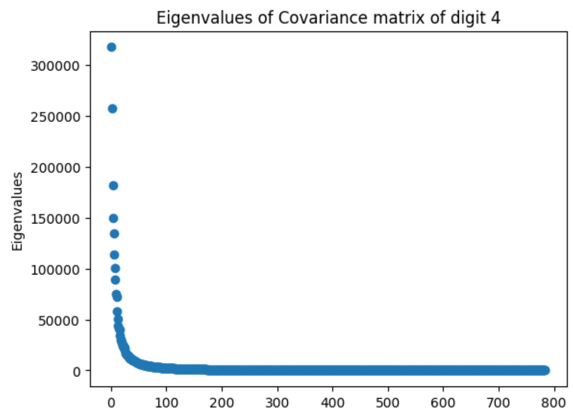
It is observed that there are **near to 100** significant Eigenvalues of covariance matrix of each digit. The reason is that, in the image, the variation occurs almost 0 at many pixels, and only selected (approx 10-20%) pixels have different intensities. The eigenvalues are the degree of variation of values of a pixel, therefore most of the eigenvalues are 0.

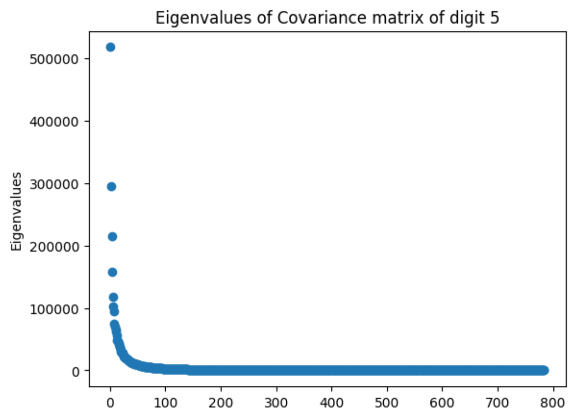


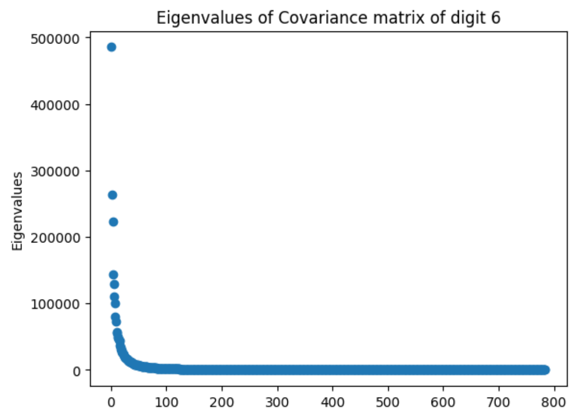


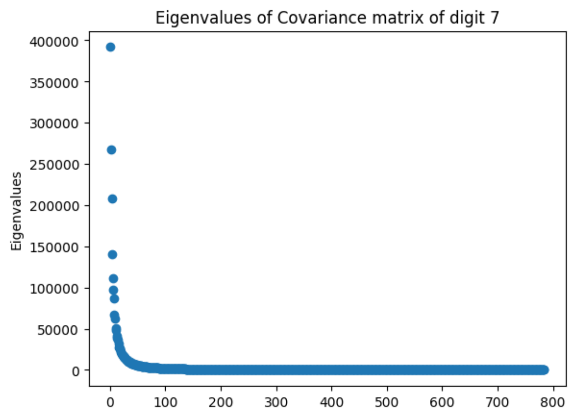


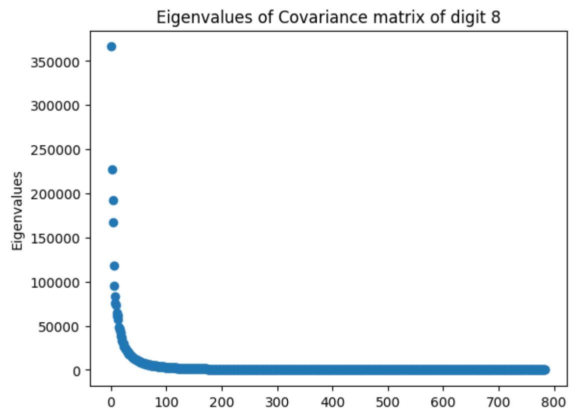


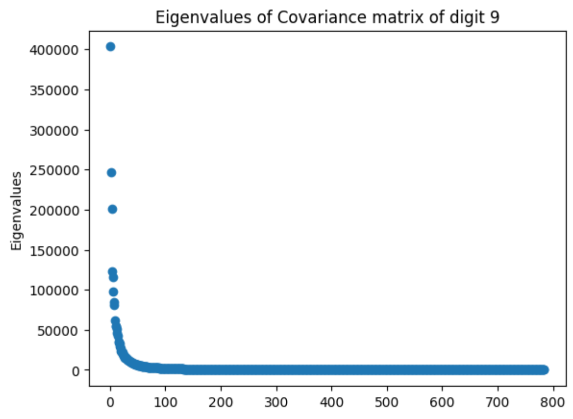






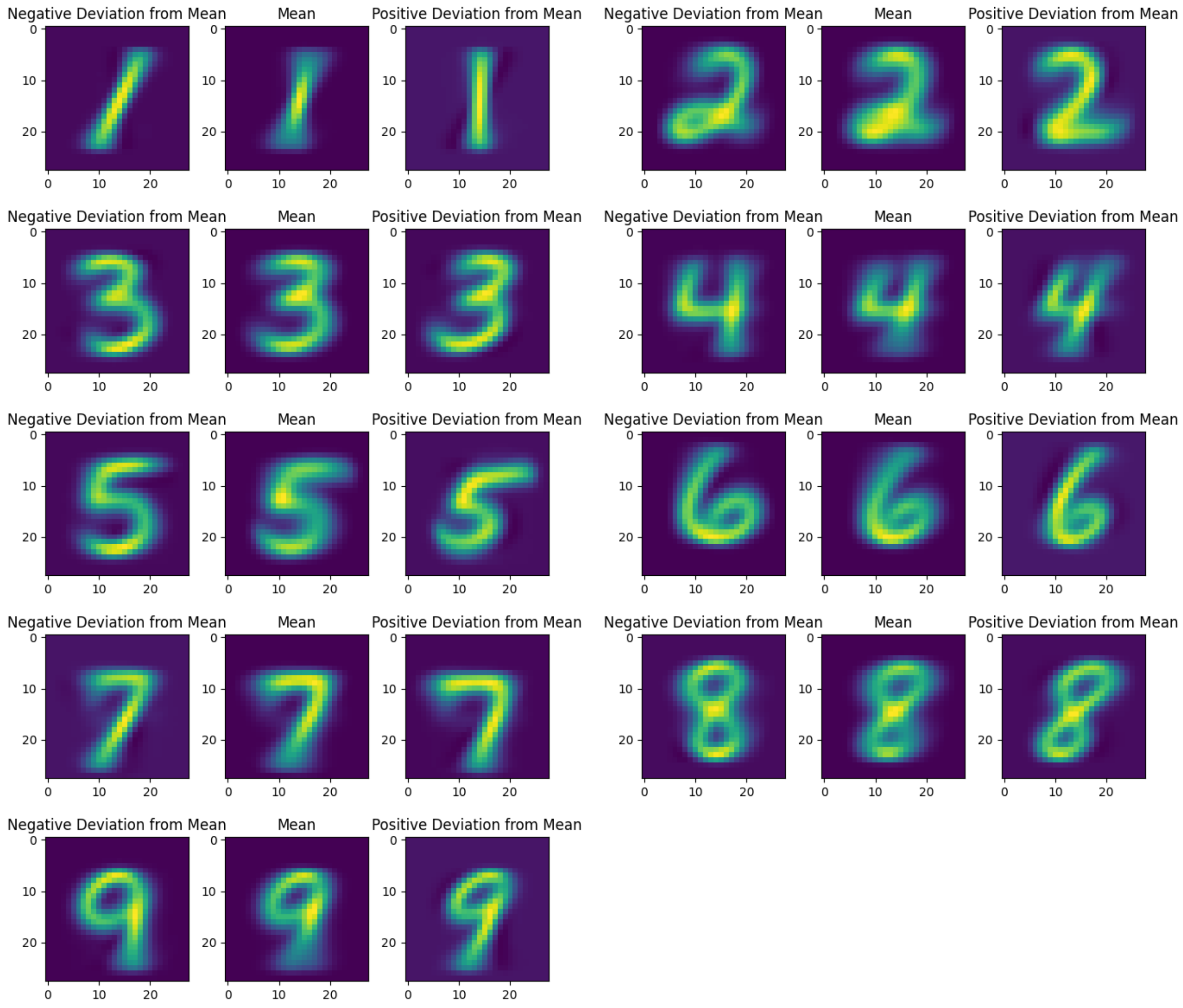






4.2 Deviation from Mean

From the principal mode of variation of the digit 1 tells us that, on an average, people write the digit 1 a little bit slanted. While, less number of people write it even more slanting, and less people write it straight.


Negative Deviation from Mean
Mean
Positive Deviation from Mean
Negative Deviation from Mean
Mean
Positive Deviation from Mean
Negative Deviation from Mean
Mean
Positive Deviation from Mean
Negative Deviation from Mean
Mean
Positive Deviation from Mean
Negative Deviation from Mean
Mean
Positive Deviation from Mean
Negative Deviation from Mean
Mean
Positive Deviation from Mean
Negative Deviation from Mean
Mean
Positive Deviation from Mean
Negative Deviation from Mean
Mean
Positive Deviation from Mean
Negative Deviation from Mean
Mean
Positive Deviation from Mean
0
10
20

5 Question 5: PCA for dimensionality Reduction

5.1 Function to compute 84 co-ordinates

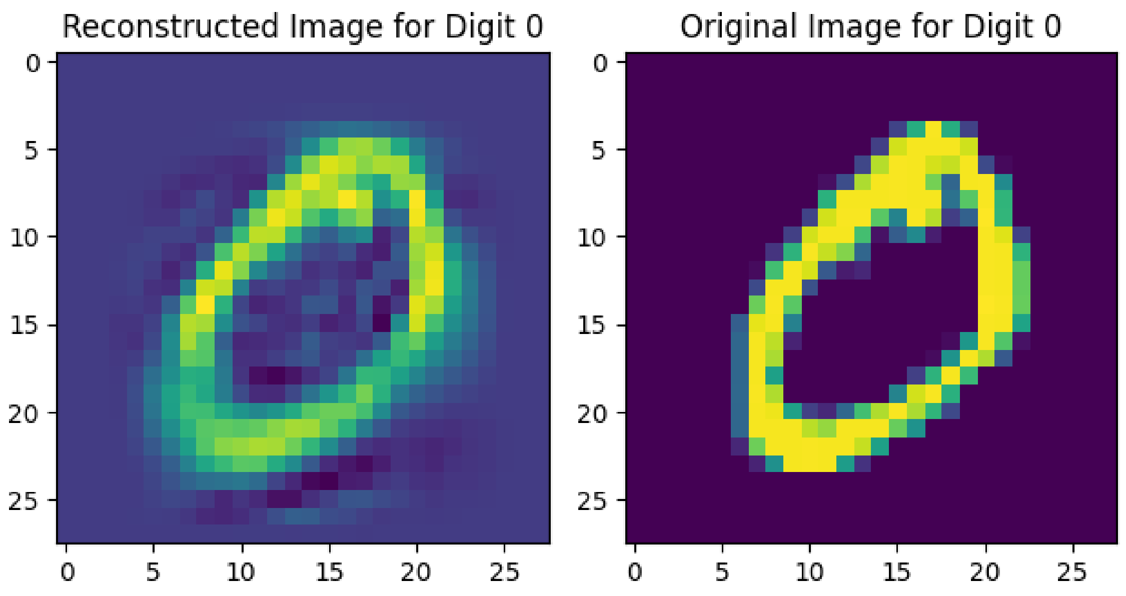
```python
#Some background to be built-up before defining the actual function doing the
    computation

#This function returns the array of images of a particular digit
def classifier(digit):
    arr = []
    count=0
    for i in range(60000):
        if y[i] == digit:
            arr.append(np.array(X[:,i]))
            count +=1
    arr = np.array(arr)
    arr = arr.transpose()
    return arr

#the 2 dimensional array "digits" stores all 60000 images but in arranged order
digits = []
for i in range(10):
    digits.append(classifier(i))

# digits_centralized array stores same info as digits but in a way such that data
    corresponding to a particular digit has mean = 0, hence centralized
means = []
digits_centralized = []
for digit in digits:
    mean = digit.mean(axis=1)
    means.append(mean)
    digit_centralized = digit.copy()
    for i in range(digit.shape[0]):
        digit_centralized[i,:] = digit_centralized[i,:] - mean[i]
    digits_centralized.append(digit_centralized)

#Function to compute the 84 co-ordinates

def computeCoordinates(digit):
    cov_matrix = np.cov(digit_centralized)

    eVals, eVectors = np.linalg.eigh(cov_matrix)
    eVals = np.flip(eVals)
    eVectors = np.flip(eVectors, axis=1)

    X_reduced = eVectors[:,:84]
    X_reduced_transpose = X_reduced.transpose()
    reduced_digit = np.matmul(X_reduced_transpose, digit)
    return reduced_digit, X_reduced
```

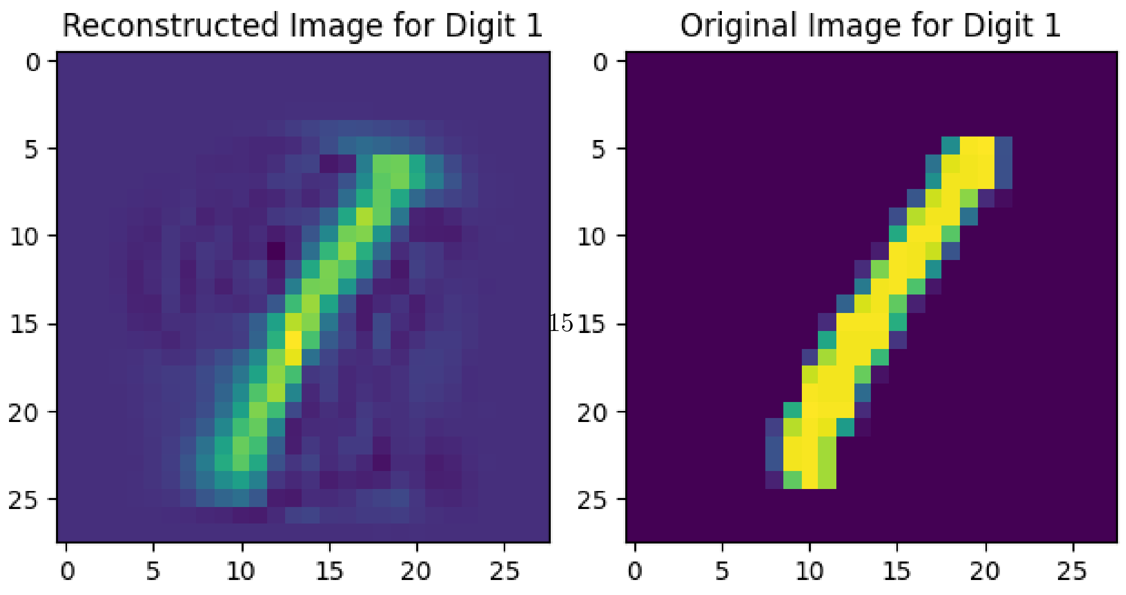
Listing 3: Function to compute 84 co-ordinates

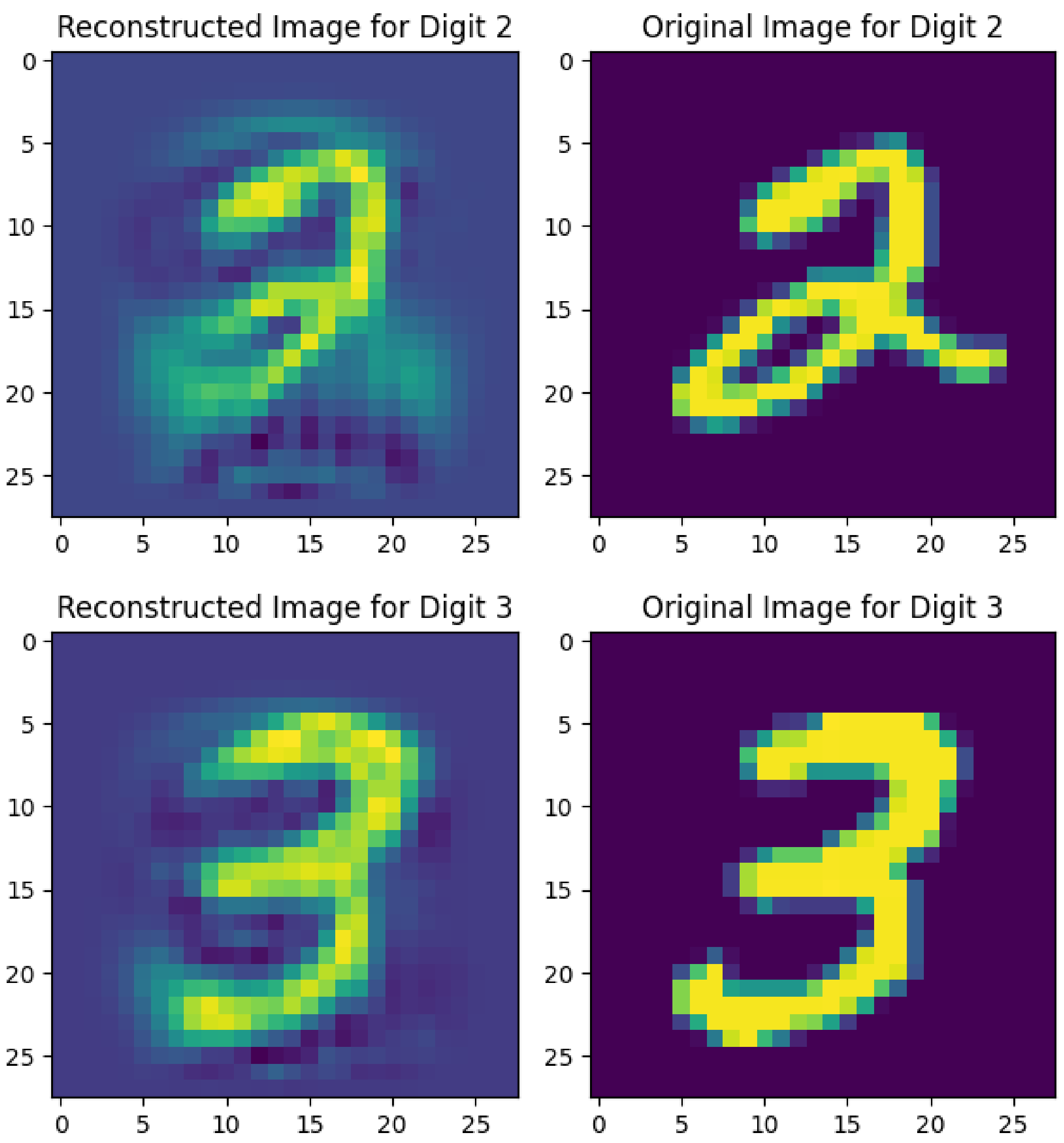
Algorithm 3 Algorithm for regenerating the image

```
$np \leftarrow numpy$
$plt \leftarrow matplotlib.pyplot$
$regenerated\_digit \leftarrow np.matmul(X\_reduced.transpose(), digit\_centralized)$
for $i \leftarrow regenerated\_digit.shape[0]$ do
    $regenerated\_digit[i,:] \leftarrow regenerated\_digit[i,:] + mean[i]$
end for
for $i \leftarrow 0$ to $len(digits\_centralized)$ do
    $reduced\_digit, X\_reduced \leftarrow computeCoordinates(digits\_centralized[i])$
    $regenerated_{d}igit, X\_reduced \leftarrow regenerateImage(reduced\_digit, digits\_centralized[i], X\_reduced, means[i])$
    $plt.subplot(1,2,1)$
    $plt.imshow(regenerated\_digit[:,0].reshape(28,28))$
    $plt.subplot(1,2,2)$
    $plt.imshow(digits[i][:,0].reshape(28,28))$
    $plt.show()$
end for
```

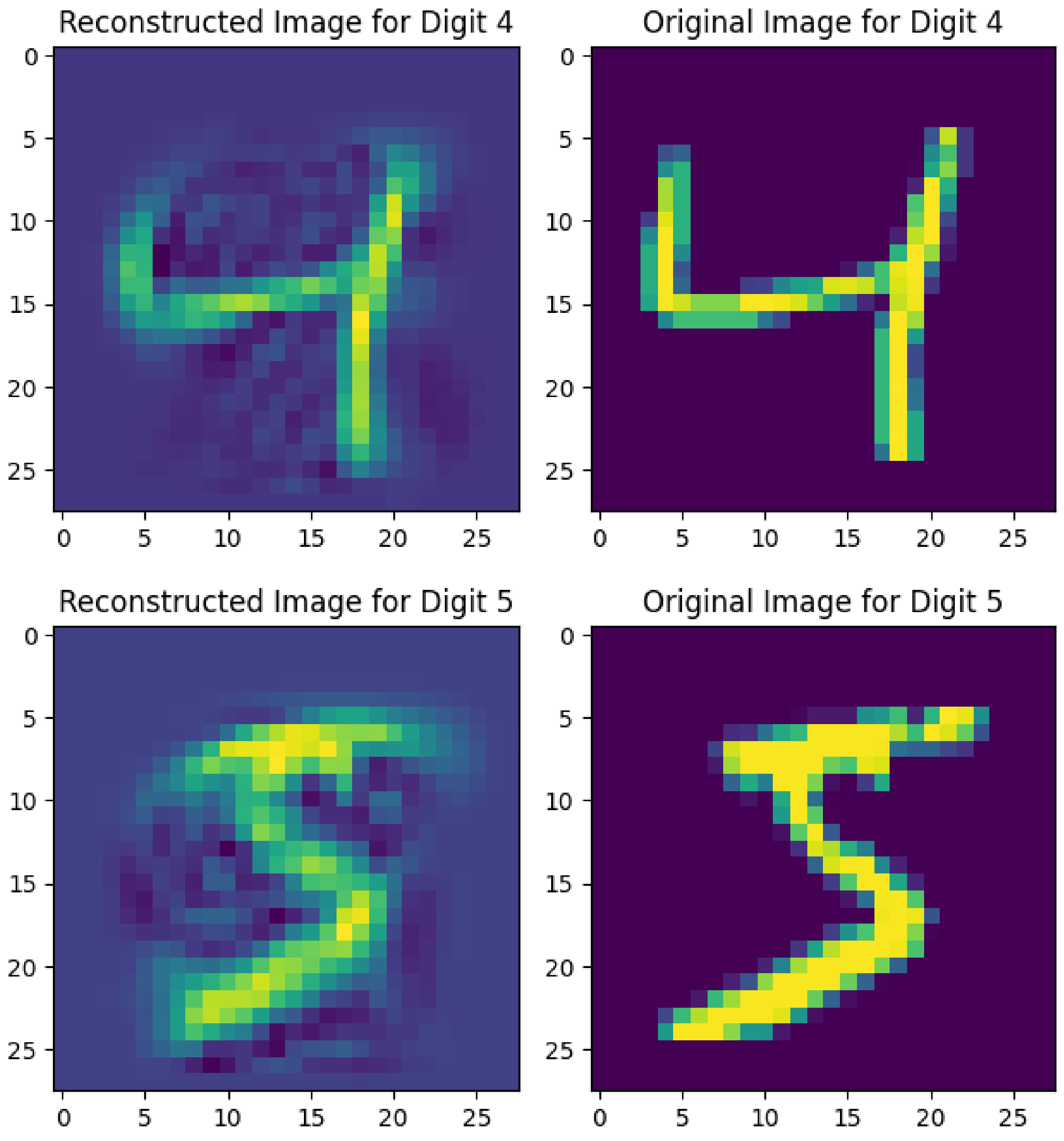
5.2 Algorithm for regenerating the image using 84 co-ordinates

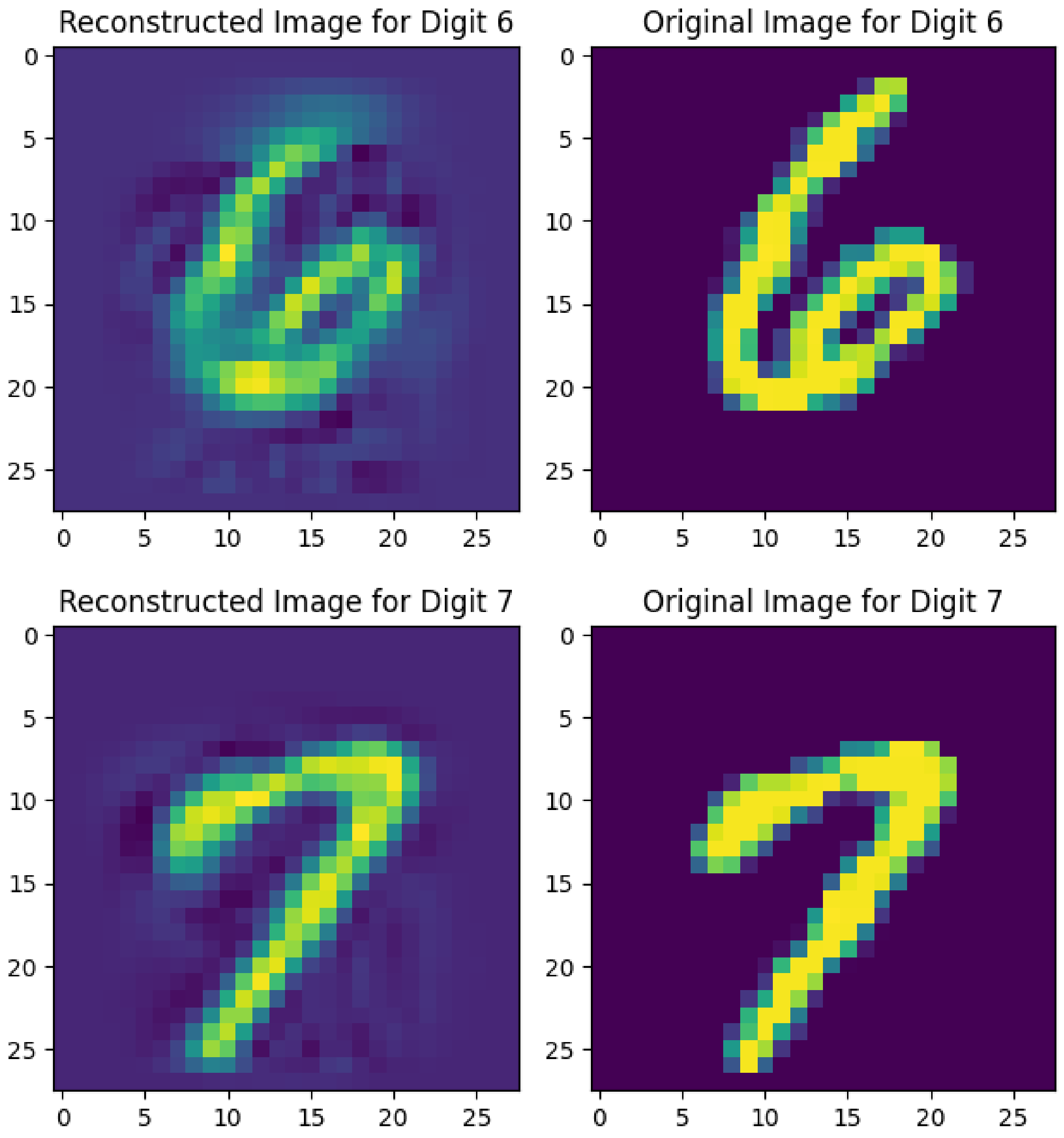


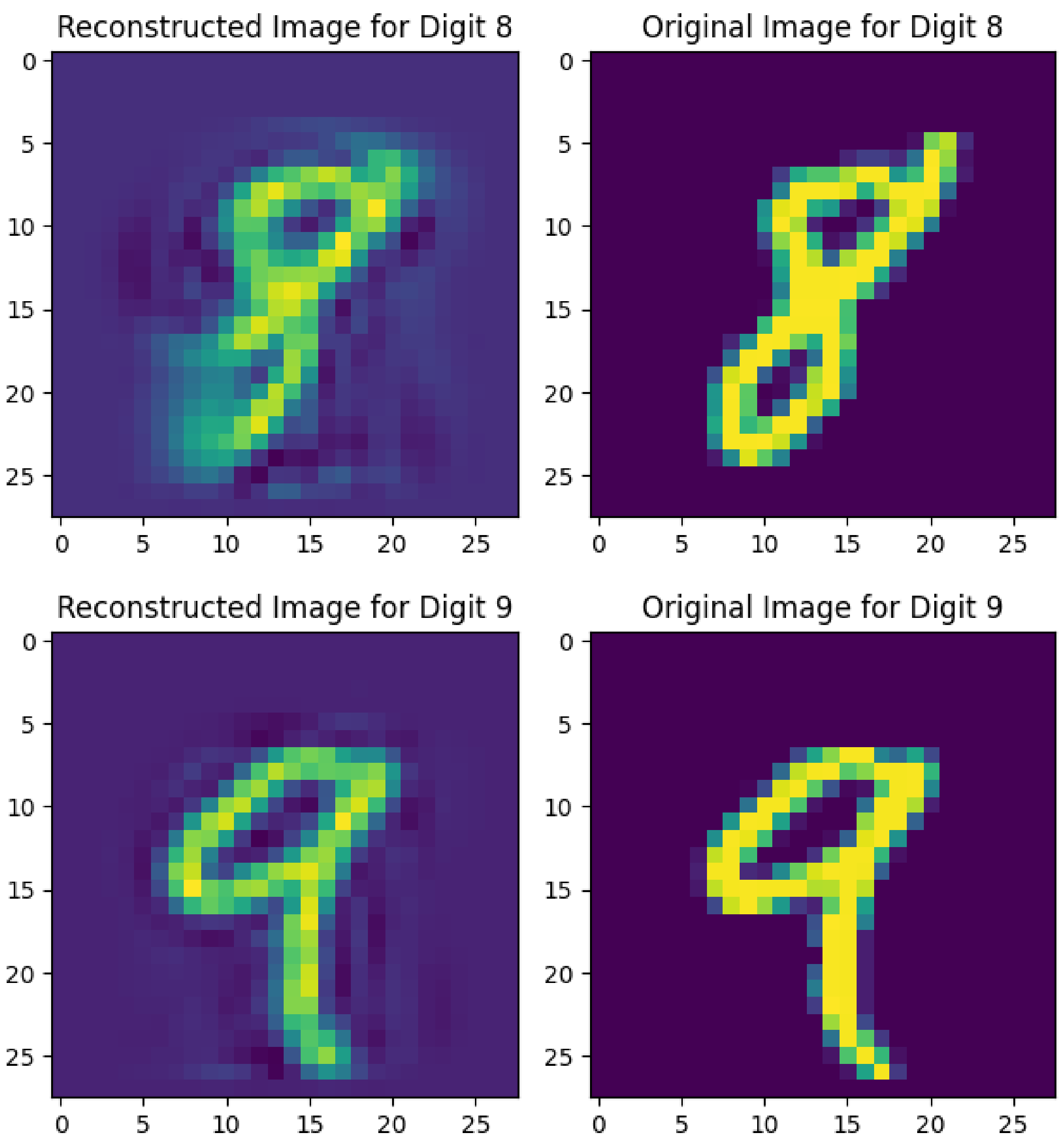




Reconstructed Image for Digit 2
Original Image for Digit 2
Reconstructed Image for Digit 3
Original Image for Digit 3
0
5
10
15
20
25

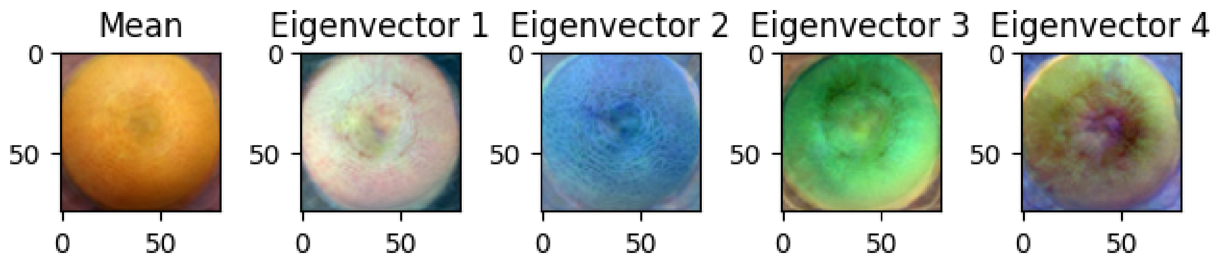

Reconstructed Image for Digit 4
Original Image for Digit 4
Reconstructed Image for Digit 5
Original Image for Digit 5
0
5
10
15
20
25


Reconstructed Image for Digit 6
Original Image for Digit 6
Reconstructed Image for Digit 7
Original Image for Digit 7

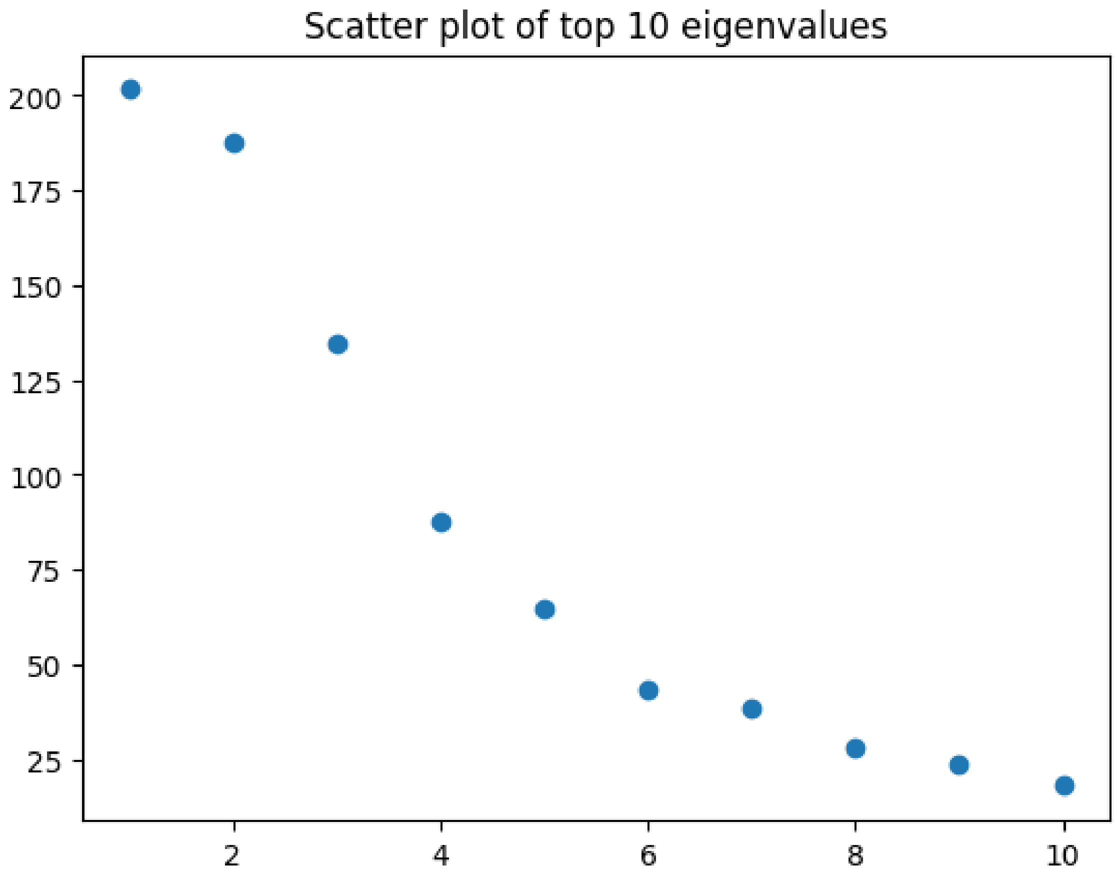

Reconstructed Image for Digit 8
Original Image for Digit 8
Reconstructed Image for Digit 9
Original Image for Digit 9
0
5
10
15
20
25

6 Question 6: PCA for another dataset

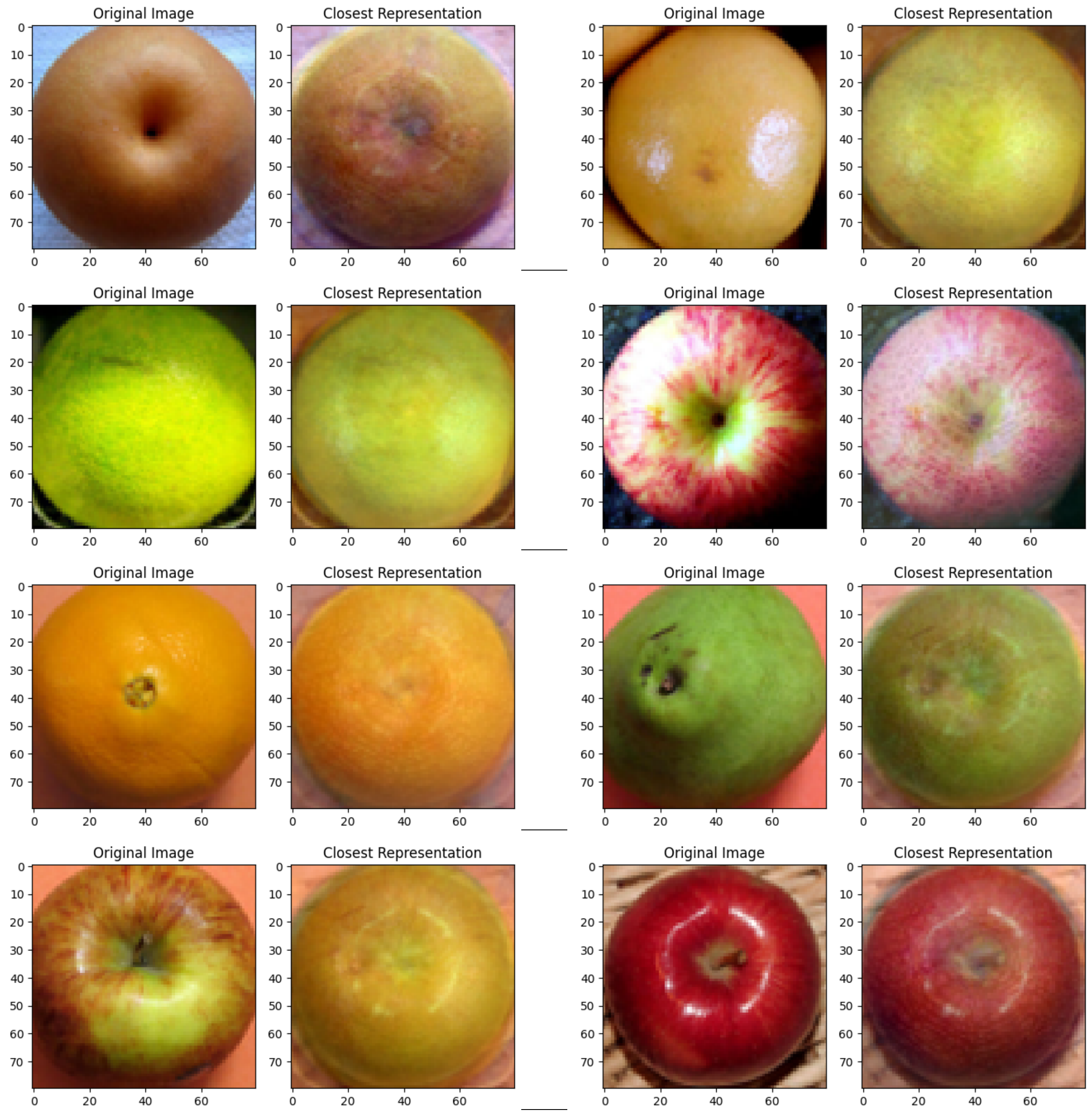
6.1 Part1: Some analysis

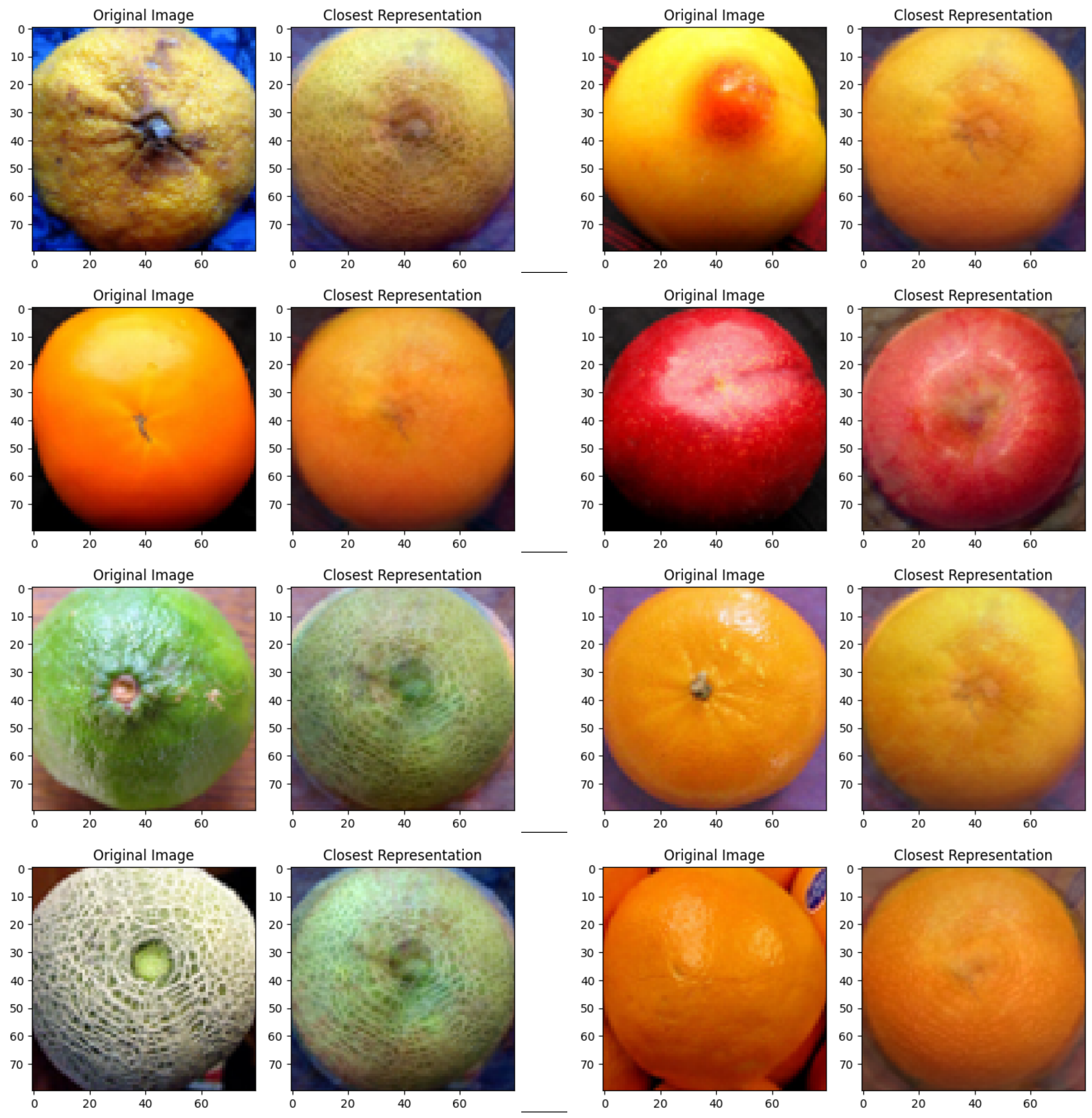


6.1.1 Graph



6.2 Part2: Closest representation




Original Image
Closest Representation
Original Image
Closest Representation
Original Image
Closest Representation
Original Image
Closest Representation
Original Image
Closest Representation
Original Image
Closest Representation
Original Image
Closest Representation
Original Image
Closest Representation

6.3 Part3: Generating new images of fruit