Question 1

a) Compute the global mean and covariance of the data

The **global mean** for the training data (after normalising by dividing by 255) comes out to be **0.13066**

The covariance matrix (784*784) has a total of **514089 non-zero values**. I found that using np.nonzero method.

```
covar = np.nonzero(np.cov(train_images_vectorize.T))
np.shape(covar)
```

b) Implement PCA and FDA from scratch

i) PCA

I created two variety of PCA functions, one gives encoded data for top ‘p’ components having total eigen energy greater than energy_boundary.

```
def pca(train_images_vectorize, energy_boundary):
    # Eigenvectors of dataset
    train_images_vectorize = train_images_vectorize - mean_vector(train_images_vectorize)
    covariance_matrix = np.matmul(train_images_vectorize.T, train_images_vectorize)
    eigenvalues, eigenvectors = np.linalg.eigh(covariance_matrix)
    print("eigenvalues shape", np.shape(eigenvalues), "eigenvectors shape", np.shape(eigenvectors))

    index_for_sort = eigenvalues.argsort()[::-1]
    eigenvalues = eigenvalues[index_for_sort]
    eigenvectors = eigenvectors[:,index_for_sort]

    last_important_eigen_value_index = eigen_energy(eigenvalues, energy_boundary)

    # Encode Training Data

    encoded_data =  np.matmul(train_images_vectorize, eigenvectors[:,:last_important_eigen_value_index])

    print("encoded data shape ", np.shape(encoded_data))

    return encoded_data
```

Below is the function to send ‘p’ value for given energy_boundary

```
def eigen_energy(eigenvalues, energy_boundary):
    eigenvalues = np.real(eigenvalues)
    total_energy = eigenvalues.sum()
    ratio_val = 0
    i = 0

    while (i<np.shape(eigenvalues)[0]) and ratio_val<energy_boundary:
        ratio_val += eigenvalues[i]/total_energy
        i+=1

    print("Ratio", ratio_val)

    return i
```

The second PCA function I created produces top ‘p’ components where p is the number of dimensions passed as paramaters.

```
def pca_ndim(train_images_vectorize, pca_ndim):
    # Eigenvectors of dataset
    train_images_vectorize = train_images_vectorize - mean_vector(train_images_vectorize)
    covariance_matrix = np.matmul(train_images_vectorize.T, train_images_vectorize)
    #print(np.shape(train_images_vectorize[0]))
    eigenvalues, eigenvectors = np.linalg.eigh(covariance_matrix)
    #print("eigenvectors", eigenvectors)
    index_for_sort = eigenvalues.argsort()[::-1]
    eigenvalues = eigenvalues[index_for_sort]
    eigenvectors = eigenvectors[:,index_for_sort]
    #print("eigenvalues ",(eigenvalues), "eigenvectors shape", (eigenvectors[:,:pca_ndim].shape))

    last_important_eigen_value_index = pca_ndim

    # Encode Training Data

    encoded_data = np.matmul(train_images_vectorize, eigenvectors[:,:last_important_eigen_value_index])
    #print(eigenvalues[0:10])

    #print("Shape of encoded data", np.shape(encoded_data))

    return encoded_data
```

ii) FDA

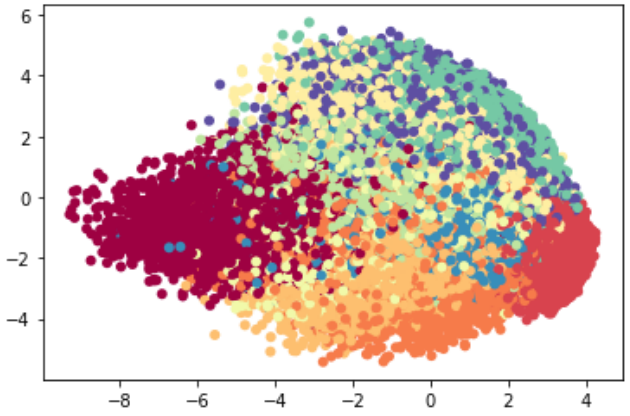
Below is my implementation of FDA, I created seperate functions to compute scatter variance and between class matrix (using total variance).

```python
def fda(train_images_vectorize, train_labels):
    train_images_vectorize -= mean_vector(train_images_vectorize)
    within_class_covariance_val = within_class_covariance(train_images_vectorize, train_labels)
    total_covariance = np.cov(train_images_vectorize.T)
    between_class_covariance_val = between_class_covariance(within_class_covariance_val, total_covariance)

    eigenvalues, eigenvectors = np.linalg.eig(np.matmul(np.linalg.pinv(within_class_covariance_val), between_class_covariance_val ))

    index_for_sort = eigenvalues.argsort()[::-1]
    eigenvalues = eigenvalues[index_for_sort]
    eigenvectors = eigenvectors[:,index_for_sort]
    print(eigenvalues[:5])
    last_important_eigen_value_index = 9

    encoded_data = np.matmul(train_images_vectorize, eigenvectors[:,:last_important_eigen_value_index])

    print("eigenvalues shape", np.shape(eigenvalues), "eigenvectors shape", np.shape(eigenvectors[:,:last_important_eigen_value_index]))

    #print(np.shape(encoded_data))

    return encoded_data
```

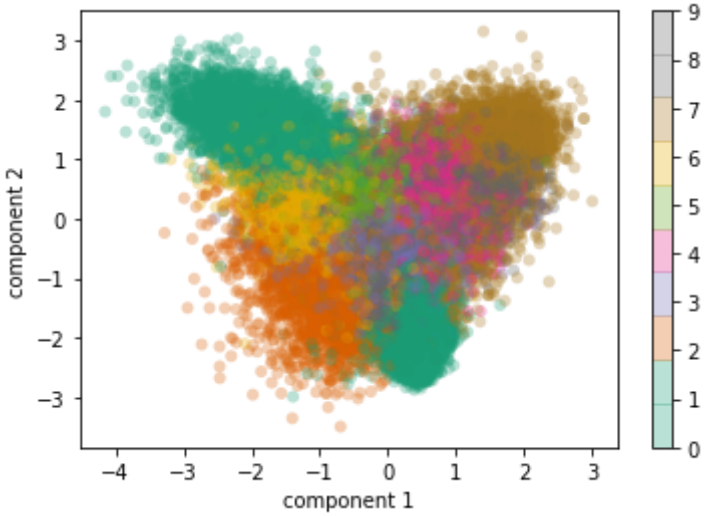
```python
def within_class_covariance(train_images_vectorize, train_labels):
    dict_fda = {}
    covar_i = []

    for i, j in zip(train_labels, train_images_vectorize):
        if(i in dict_fda):
            dict_fda[i].append(j)

        else:
            dict_fda[i] = [j]

    for label in np.unique(train_labels):
        dict_fda[label] = np.asarray(dict_fda[label])
        dict_fda[label] -= np.mean(dict_fda[label], axis=0)

    for i in dict_fda:
        covar_i.append(np.matmul(dict_fda[i].T, dict_fda[i]))

    print(np.shape((np.sum(covar_i, axis=0))))
    return np.sum(covar_i, axis=0)

def between_class_covariance(within_class_covariance, total_covariance):
    return total_covariance - within_class_covariance
```

c) Visualize data using a scatter plot after applying PCA & FDA.

I applied **PCA** on the training data set for eigen energy boundary 0.95 and got 154 components out of the 784 dimensions, Here is the plot of the first two dimensions of the transformed (encoded data).



I applied **FDA** on the data with 154 dimensions and plotted the first two dimensions of the newly constructed data.



d. Implement the LDA discriminant function from scratch.

In the LDA function I am attempting to fit a gaussian to each class by computing mean and covariance class-wise. Prior probabilities are also calculated as would be required by the discriminant function.

```
# Ans 4
def lda(train_images_vectorize, train_labels):
    dict_lda = {}
    prior_probabilities = {}
    mu_vals = {}
    covar = {}
    cov_det = {}
    cov_inv = {}

    for i, j in zip(train_labels, train_images_vectorize):
        if(i in dict_lda):
            dict_lda[i].append(j)

        else:
            dict_lda[i] = [j]

    for label in np.unique(train_labels):
        dict_lda[label] = np.asarray(dict_lda[label])

    for i in dict_lda:
        prior_probabilities[i] = np.shape(dict_lda[i])[0]/np.shape(train_images_vectorize)[0]

    for i in dict_lda:
        mu_vals[i] = np.mean(dict_lda[i], axis = 0)

    for i in dict_lda:
        covar[i] = np.cov(dict_lda[i].T)
        cov_det[i] = np.linalg.slogdet(covar[i])
        cov_inv[i] = np.linalg.pinv(covar[i])
    return prior_probabilities, mu_vals, covar, cov_det, cov_inv
```

LDA predict function uses the values returned by lda to compute the discriminant value for any test case.

```
def lda_predict(prior_probab, mu_vals, covar, cov_det, cov_inv, test):
    discriminant_max = -100000
    predict_class = -1


    for i in range(10):
        #print(test.shape)
        #disc = (-1/2)*np.log(cov_det[i]) - (1/2) * np.matmul(np.matmul((test - mu_vals[i]).T, cov_inv[i]), test - mu_vals[i]) + np.log(prior_probab[i])
        disc = (-1/2)*cov_det[i][1] - (1/2) * np.matmul(np.matmul((test - mu_vals[i]).T, cov_inv[i]), test - mu_vals[i]) + np.log(prior_probab[i])

        if (disc>discriminant_max):
            discriminant_max = disc
            predict_class = i

    return predict_class
```

e. Apply PCA with 95% eigen energy on MNIST and then LDA for classification and report the accuracy on test data.

After applying PCA with 95% eigen energy on train data and running the discriminant functions over test data **94.84**.

```
data_after_pca, eigenvectors = pca(train_images_vectorize, 0.95)
prior_probab, mu_vals, covar, cov_det, cov_inv = lda(data_after_pca, train_labels)

test_images_vectorize = n_vectorize(test_images)
test_images_vectorize = test_images_vectorize - mean_vector(test_images_vectorize)
test_images_vectorize_projected = np.matmul(test_images_vectorize, eigenvectors)

correct = 0

n_test = np.shape(test_images_vectorize_projected)[0]

for i in range(n_test):
    if(lda_predict(prior_probab, mu_vals, covar, cov_det, cov_inv, test_images_vectorize_projected[i]) == test_labels[i]):
        correct+=1
    #print(train_labels[i], lda_predict(prior_probab, mu_vals, train_images_vectorize[i]))

print("Accuracy on test data after 0.95 eigen PCA on test data", (((correct)/n_test)*100))
```

f) Visualize and analyze the eigenvectors obtained using PCA (only for eigenvectors obtained in part(e). I.e., Display eigenvectors by converting them into image form).

Eigenvectors would represent the new subspace of the data after PCA gets performed over it.

g. Perform step(e) with different eigen energy mentioned below and show the comparisons and analysis on accuracy.

- 70% eigen energy

I got **95.83** accuracy on 0.70 eigen-energy PCA.

- 90% eigen energy

I got **95.89** accuracy on 0.70 eigen-energy PCA.

- 99% eigen energy

I got **72.84** accuracy on 0.99 eigen-energy PCA.

The increasing accuracy with respect to the eigen energy maybe due to the case that though the computation power required to label the test data increases, more parameters help fit the data better. On the other hand it decreased in case of 99% eigen energy as the discriminant function may be approaching overfitting with a huge train set available.

```
Accuracy on test data after 0.95 eigen PCA on test data 94.84
eigenvalues shape (784,) eigenvectors shape (784, 784)
Ratio 0.700198100207202
encoded data shape  (60000, 26)
Accuracy on test data after 0.70 eigen PCA on test data 95.83
eigenvalues shape (784,) eigenvectors shape (784, 784)
Ratio 0.9001062226425082
encoded data shape  (60000, 87)
Accuracy on test data after 0.90 eigen PCA on test data 95.89
eigenvalues shape (784,) eigenvectors shape (784, 784)
Ratio 0.9900129426354092
encoded data shape  (60000, 331)
Accuracy on test data after 0.99 eigen PCA on test data 72.84
```

h) Apply FDA on MNIST and then LDA for classification and report the accuracy on test data.

I got a low accuracy of 9.8 percent which may be due to underflowing of values in the scatter matrix which may be producing the same predict value for each test case.

```
####FDA

data_after_fda, eigenvectors = fda(train_images_vectorize, train_labels)
prior_probab, mu_vals, covar, cov_det, cov_inv = lda(data_after_fda, train_labels)

test_images_vectorize = n_vectorize(test_images)
test_images_vectorize = test_images_vectorize - mean_vector(test_images_vectorize)
test_images_vectorize_projected = np.matmul(test_images_vectorize, eigenvectors)

correct = 0

n_test = np.shape(test_images_vectorize_projected)[0]

for i in range(n_test):
    if(lda_predict(prior_probab, mu_vals, covar, cov_det, cov_inv, test_images_vectorize_projected[i]) == test_labels[i]):
        correct+=1
    #print(train_labels[i], lda_predict(prior_probab, mu_vals, train_images_vectorize[i]))

print("Accuracy on test data after FDA on test data", (((correct)/n_test)*100))
```

```
Accuracy on test data after FDA on test data 9.8
(331, 331)
[-0.99992042+0.j -0.99992964+0.j -0.99993634+0.j -0.99995565+0.j
 -0.99995864+0.j]
eigenvalues shape (331,) eigenvectors shape (331, 9)
Ratio 0.9501960192613034
eigenvalues shape (784,) eigenvectors shape (784, 154)
encoded data shape  (60000, 154)
Accuracy on test data after FDA on test data 94.84
```

i.) Perform PCA then FDA. Classify the transformed datasets using LDA. Analyze the results on Accuracy.

Accuracy on test data after FDA on test data 94.84. In my case as the FDA is not contributing much the the FDA+PCA is working similar to the way it was working for only PCA.

```
####FDA then PCA

data_after_fda, eigenvectors = fda(data_after_pca, train_labels)
data_after_pca, eigenvectors = pca(train_images_vectorize, 0.95)
prior_probab, mu_vals, covar, cov_det, cov_inv = lda(data_after_pca, train_labels)

test_images_vectorize = n_vectorize(test_images)
test_images_vectorize = test_images_vectorize - mean_vector(test_images_vectorize)
test_images_vectorize_projected = np.matmul(test_images_vectorize, eigenvectors)

correct = 0

n_test = np.shape(test_images_vectorize_projected)[0]

for i in range(n_test):
    if(lda_predict(prior_probab, mu_vals, covar, cov_det, cov_inv, test_images_vectorize_projected[i]) == test_labels[i]):
        correct+=1
    #print(train_labels[i], lda_predict(prior_probab, mu_vals, train_images_vectorize[i]))

print("Accuracy on test data after applying FDA then PCA", (((correct)/n_test)*100))
```

Question 2

a. Add Gaussian noise to the dataset.

Am adding the guassian noise to the image dataset(mnist-train) after vectorizing the later from 60000x28x28 to 60000x784 dimensions.

Below you can see the gauss_noise function I have created to add noise to the dataset.

```
def gauss_noise(dataset, sigma=0.2, mu=0):
    noise = np.random.normal(mu, sigma, (dataset.shape))
    dataset = dataset + noise
    return dataset
```

Here is the function I used to flatten the dataset.

```
# Flatten data. ie convert Nxdxd image to Nx(d^2)x1 vector
def n_vectorize(dataset):
    flat_data = dataset.reshape(dataset.shape[0], -1)
    #print(np.shape(flat_data))
    return flat_data
```

b. Perform PCA on the noisy dataset for Noise Reduction.

Below is the PCA function I created which takes train dataset and energy boundary as parameters.

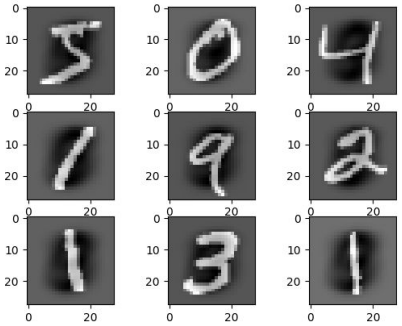
```python
def pca(train_images_vectorize, energy_boundary):
# Eigenvectors of dataset
    train_images_vectorize = train_images_vectorize - mean_vector(train_images_vectorize)
    covariance_matrix = np.matmul(train_images_vectorize.T, train_images_vectorize)
    eigenvalues, eigenvectors = np.linalg.eigh(covariance_matrix)
    print("eigenvalues shape", np.shape(eigenvalues), "eigenvectors shape", np.shape(eigenvectors))

    index_for_sort = eigenvalues.argsort()[::-1]
    eigenvalues = eigenvalues[index_for_sort]
    eigenvectors = eigenvectors[:,index_for_sort]

    last_important_eigen_value_index = eigen_energy(eigenvalues, energy_boundary)

    # Encode Training Data

    encoded_data =  np.matmul(train_images_vectorize, eigenvectors[:,:last_important_eigen_value_index])

    print("encoded data shape ", np.shape(encoded_data))

    return encoded_data, eigenvectors[:,:last_important_eigen_value_index]
    #return eigenvectors
```

```python
def eigen_energy(eigenvalues, energy_boundary):
    total_energy = eigenvalues.sum()
    ratio_val = 0
    i = 0

    while (i<np.shape(eigenvalues)[0]) and ratio_val<energy_boundary:
        ratio_val += eigenvalues[i]/total_energy
        i+=1

    print("Ratio", ratio_val)

    return i
```

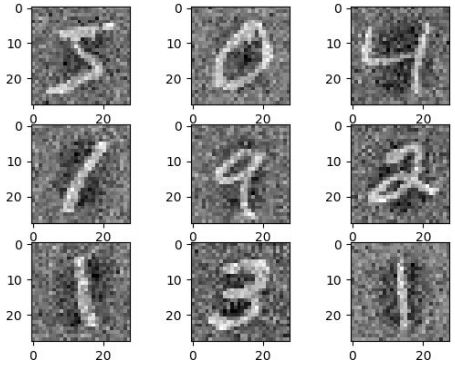
Here is the eigen energy function which returns the number of top k eigenvalues to perform dimensional reduction with.

c. Visualize the dataset before & after noise reduction. (Report the images as shown below. Linear PCA in the below image refers to normal PCA only.).

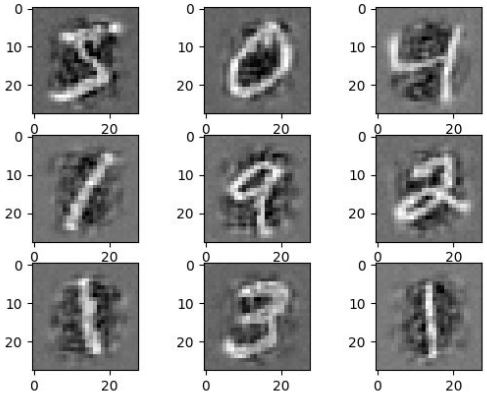
(Centralized) Dataset before gauss noise was added:



Dataset after Gauss noise was added:



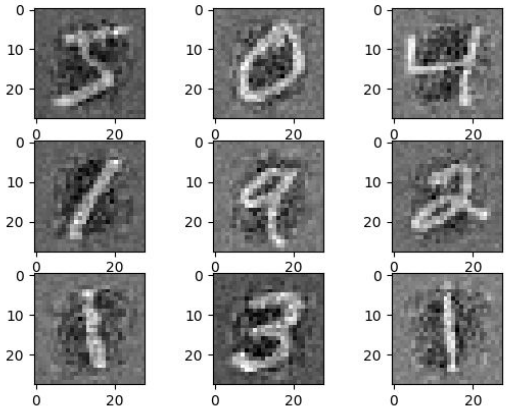
Dataset after PCA done for eigen energy(0.70) (195 component)



d. Report the number of components for which PCA works the best in Noise Reduction.

While trying for different combinations, eigen energy boundary = 0.70 (number of components is 195) gave the best visualisation. There is no concrete right answer I suppose in this case. When we introduce noise, the pca doesn't know about the increment it will try its best to remove the dimensions according to the new set. If the noise is equally distributed then we can expect PCA to perform some denoising.

For eg for eigenenergy 0.80 (PCA components 362) here is the visualisation


0
10
20
0
20