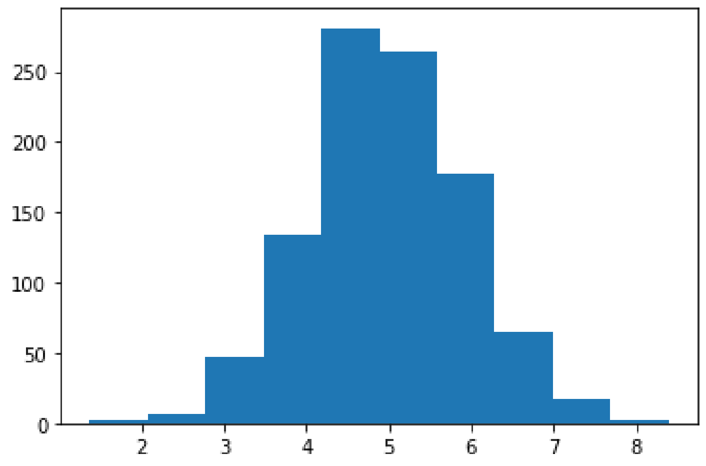
Machine Learning (CSE 6363) ASSIGNMENT 3

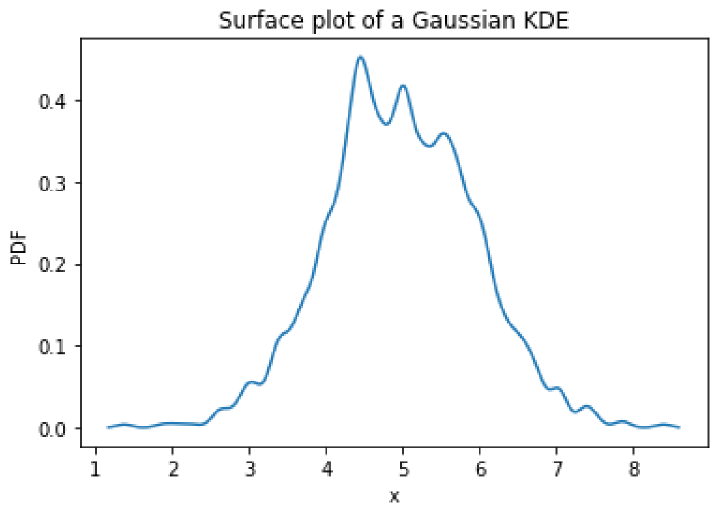
Name: Ipsa Mishra
Student ID: 1001759616

In the first problem we have computed the KDE (Kernel Density Estimation) using Python. It is a non-parametric way to estimate the probability density function of a variable. Then we have proceeded with the question and listed them in this report according to the instructions given in the problem.

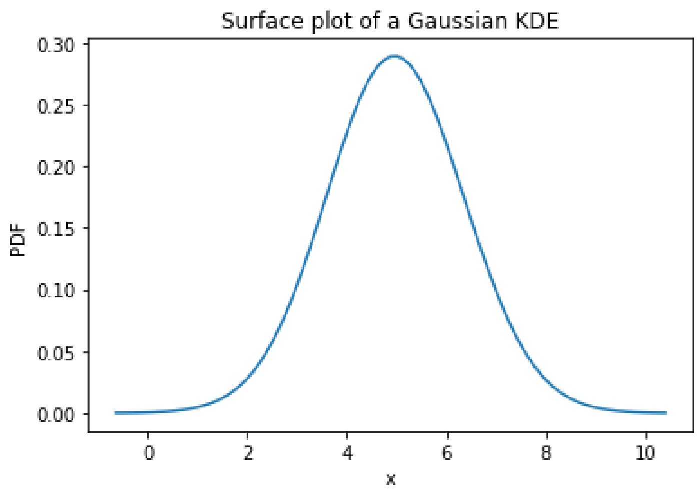
1.1a) Dataset with N = 1000, mean = 5 and std = 1



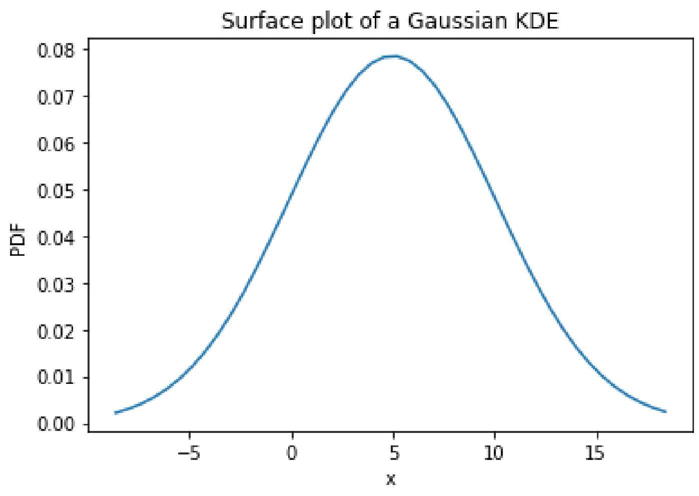
The domain of x-axis = [1.1720208128777057, 8.59691613820467] The bandwidth = 0.1



The domain of x-axis = [-0.6279791871222944, 10.39691613820467] The bandwidth = 1

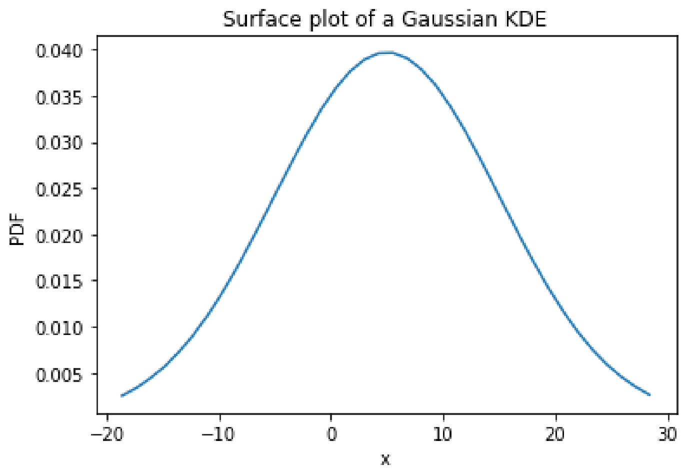


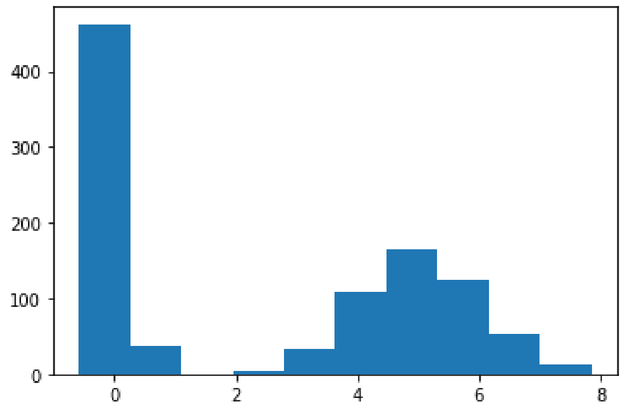
The domain of x-axis = [-8.627979187122294, 18.39691613820467] The bandwidth = 5



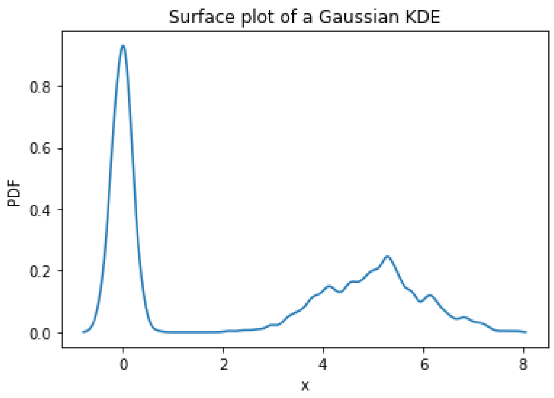
The domain of x-axis = [-18.627979187122293, 28.39691613820467] The bandwidth = 10

b) Dataset with N = 1000, mean_1 = 5 and std_1 = 1 & mean_2 = 0 and std_2 = 0.2

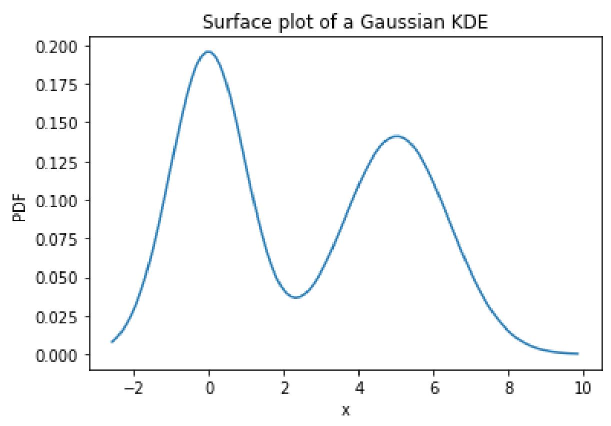




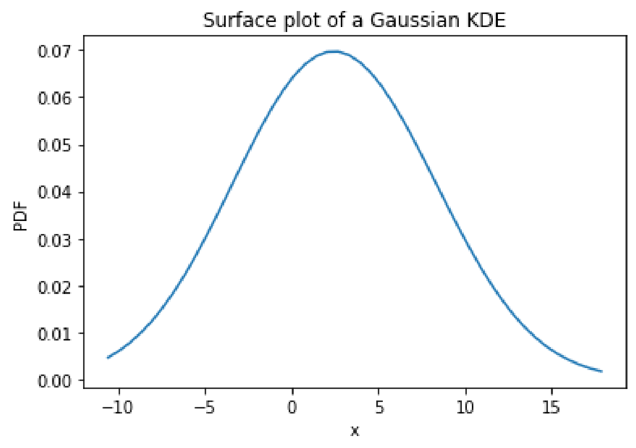
The domain of x-axis = [-0.7851858019277349, 8.050344644359088] The bandwidth = 0.1



The domain of x-axis = [-2.585185801927735, 9.850344644359089] The bandwidth = 1

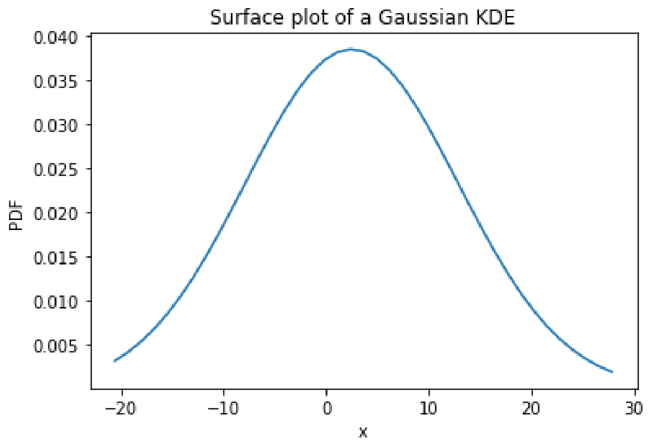


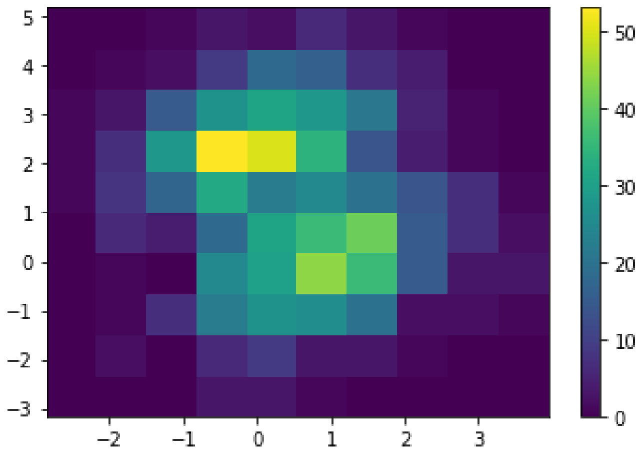
The domain of x-axis = [-10.585185801927734, 17.85034464435909] The bandwidth = 5



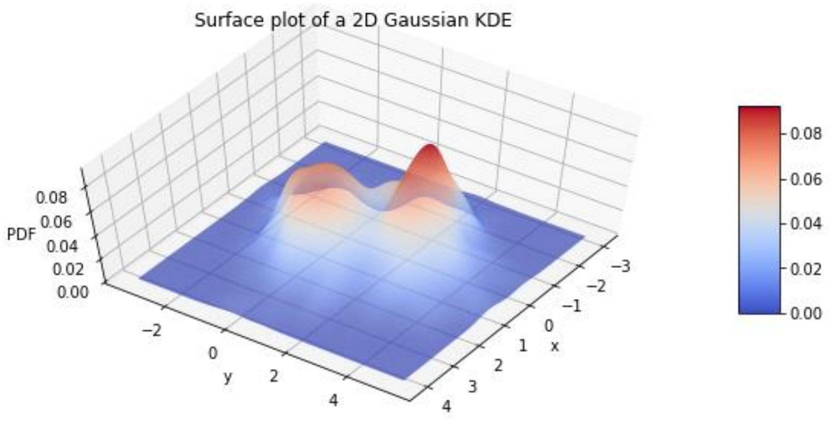
The domain of x-axis = [-20.585185801927736, 27.85034464435909] The bandwidth = 10

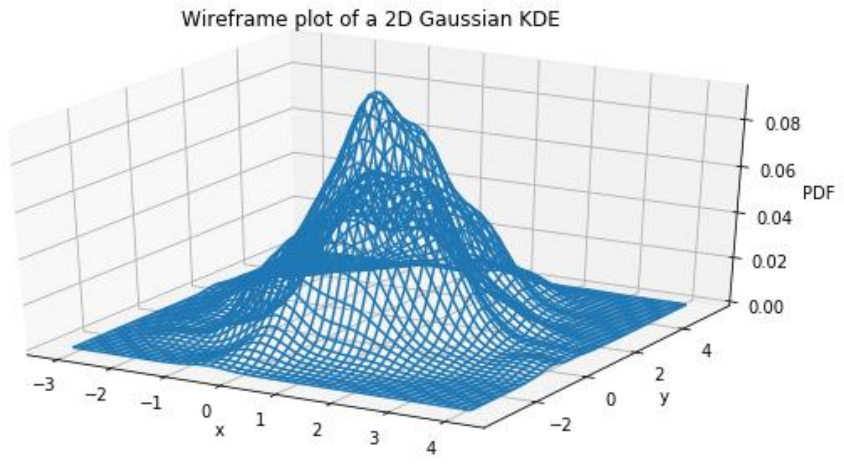
1.2 Test function mykde on this data with h = {0.1, 1, 5, 10}



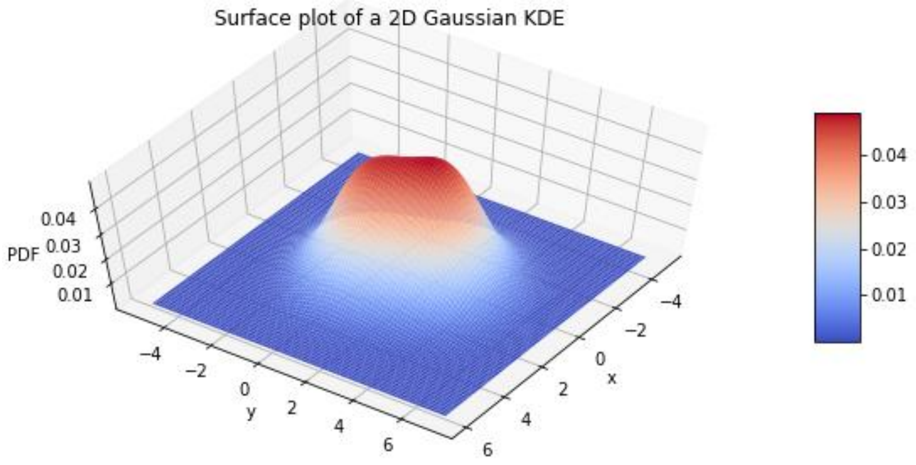


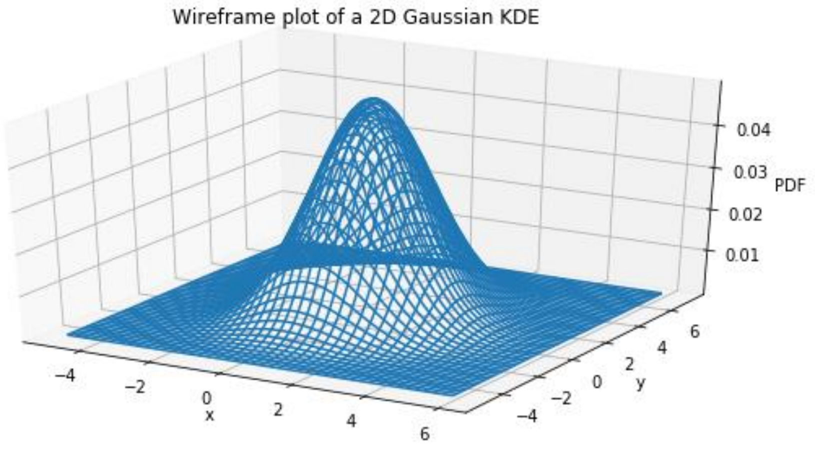
The bandwidth = 0.1
The domain of x-axis = [-3.0483257061090194, 4.148851425509042] The domain of y-axis = [-3.3592207863752193, 5.38799887039534]



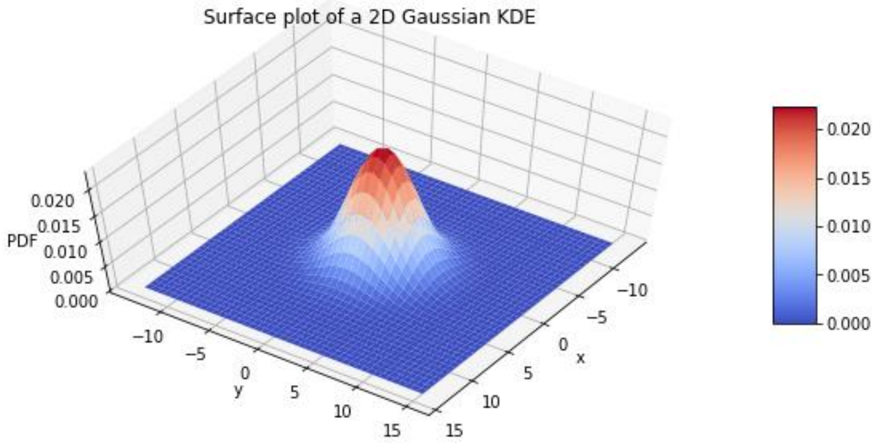


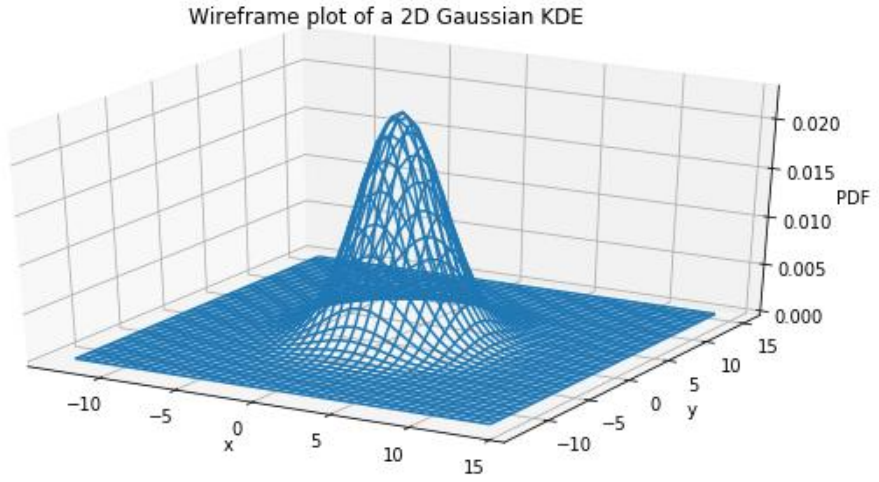
The bandwidth = 1
The domain of x-axis = [-4.848325706109019, 5.948851425509042] The domain of y-axis = [-5.159220786375219, 7.18799887039534]





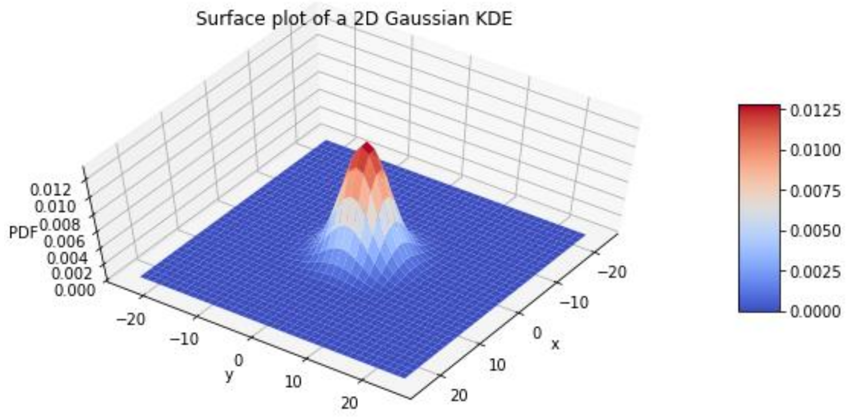
The bandwidth = 5
The domain of x-axis = [-12.84832570610902, 13.94885142550904] The domain of y-axis = [-13.159220786375219, 15.18799887039534]

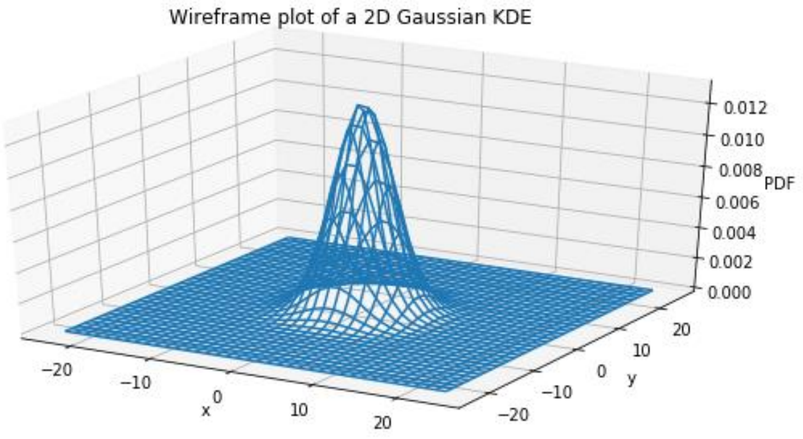




The bandwidth = 10

The domain of x-axis = [-22.84832570610902, 23.94885142550904] The domain of y-axis = [-23.15922078637522, 25.18799887039534]





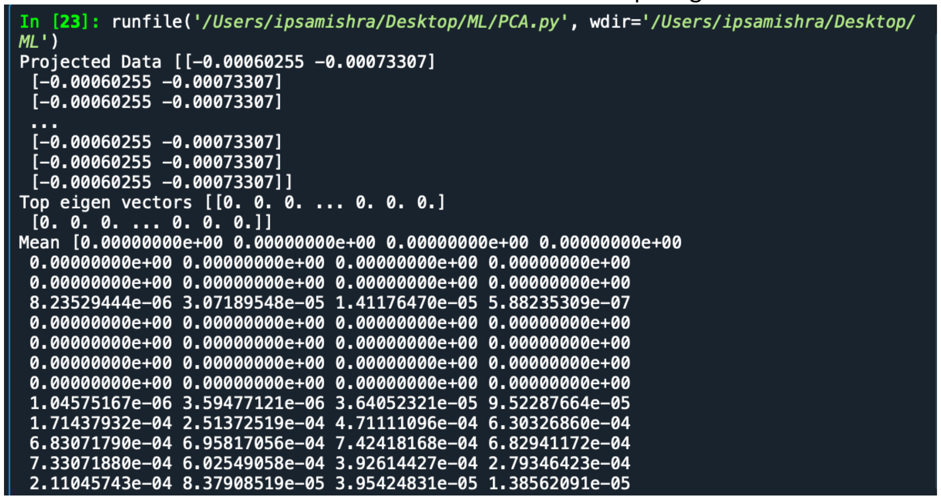
2.1 Write your own function [PC] = myPCA(X, k) that returns k principle components (PCs)

```
def myPCA(X ,k):
    M = np.mean(X , axis = 0)                          # Calculate Mean
    N = X - M                                          # Normalize data
    C = np.cov(N , rowvar = False)                     # Calculate covariance (trick)
    eig_vals, eig_vecs = np.linalg.eigh(C)             # Eigen computation for symmetric matrix
    ind = np.argsort(eig_vals)[::-1]                   # Descending order for eigen-values
    eig_vecs = eig_vecs[ind, :]                        # Sort eigen-vectors
    top_eig = eig_vecs[0:k, :]                         # Choose top k eigen-vectors
    X_kPC = np.matmul(N, np.transpose(top_eig))        # Projected data
    return X_kPC, top_eig, M
```

2.2 Run your PCA on MNIST dataset

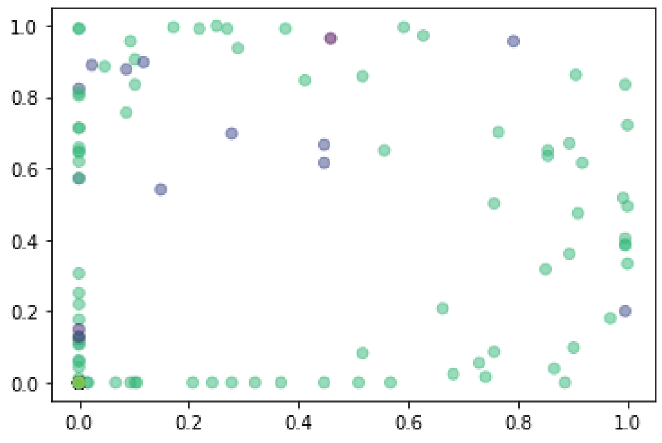
Time taken to perform PCA on MNIST train data with 2 dimensions is 1.4940829277038574 seconds (as shown by the output)

I am able to run it. I have attached a screenshot of the output I got.



```
In [23]: runfile('/Users/ipsamishra/Desktop/ML/PCA.py', wdir='/Users/ipsamishra/Desktop/
ML')
Projected Data [[-0.00060255 -0.00073307]
 [-0.00060255 -0.00073307]
 [-0.00060255 -0.00073307]
 ...
 [-0.00060255 -0.00073307]
 [-0.00060255 -0.00073307]
 [-0.00060255 -0.00073307]]
Top eigen vectors [[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]]
Mean [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00
 0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00
 0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00
 8.23529444e-06 3.07189548e-05 1.41176470e-05 5.88235309e-07
 0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00
 0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00
 0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00
 0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00
 1.04575167e-06 3.59477121e-06 3.64052321e-05 9.52287664e-05
 1.71437932e-04 2.51372519e-04 4.71111096e-04 6.30326860e-04
 6.83071790e-04 6.95817056e-04 7.42418168e-04 6.82941172e-04
 7.33071880e-04 6.02549058e-04 3.92614427e-04 2.79346423e-04
 2.11045743e-04 8.37908519e-05 3.95424831e-05 1.38562091e-05
```

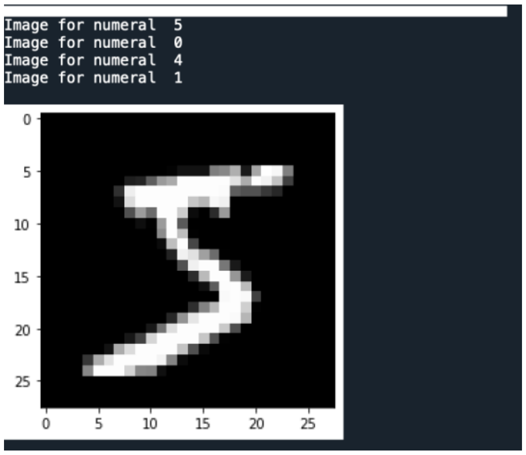
2.3 Visualize the data using the first 2 PC, i.e., plot each sample as scatter plot by projecting them onto a 2D spaces whose each axis is the PC.



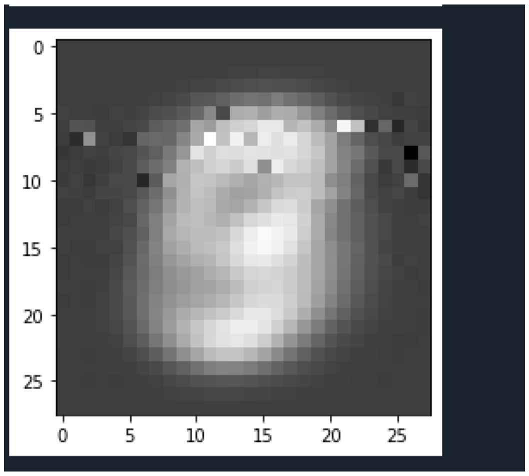
The plot doesn't look good as the data from different classes are all jumbled up.

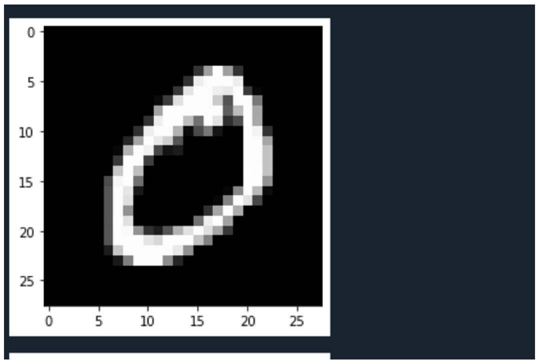
2.4 Visualize 10 PC as images.

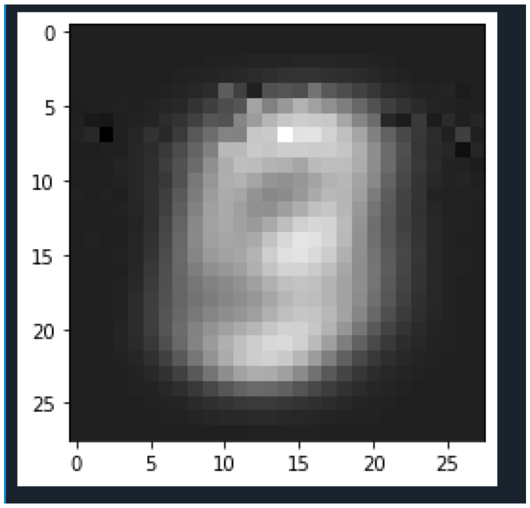
The outputs are shown over here.

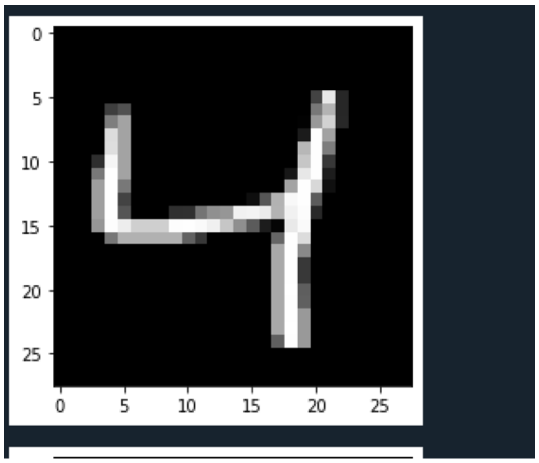


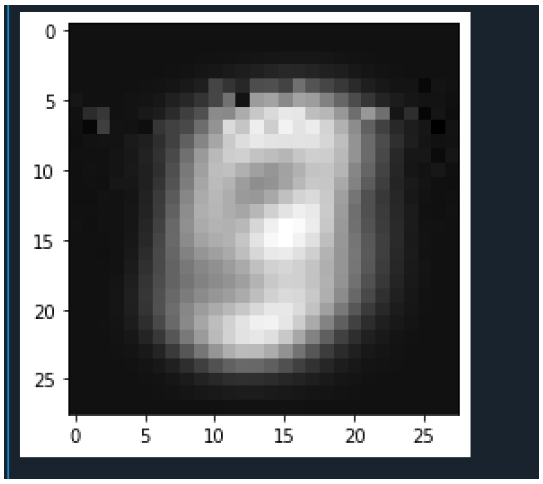
```
Image for numeral  5
Image for numeral  0
Image for numeral  4
Image for numeral  1
```

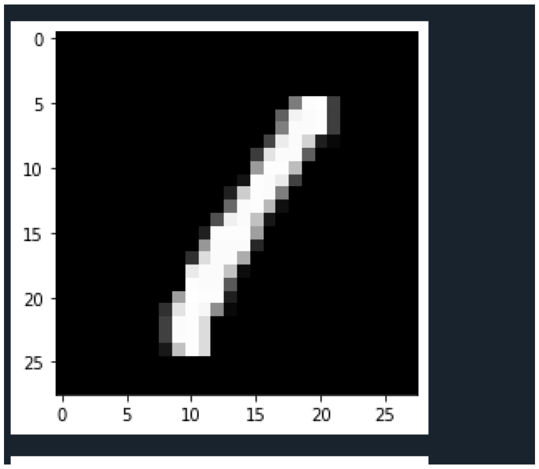


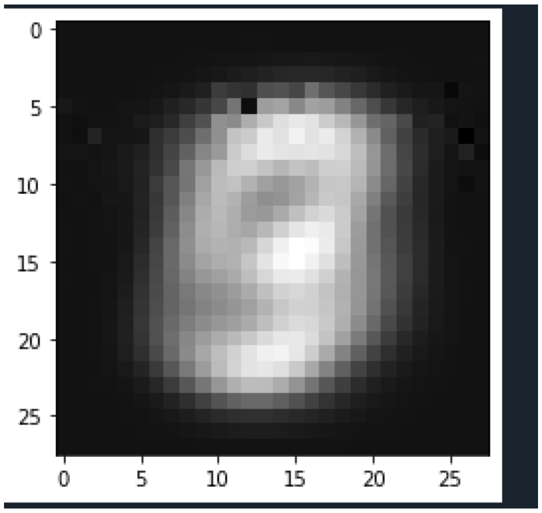



0
5
10
15
20
25


0
5
10
15
20
25


0
5
10
15
20
25





As we can see in here, the images are not very clear using 10 principal components but it is somewhat similar. These shapes look like slanted version of the digits.

2.5 Use Logistic Regression with 10 output nodes with softmax function as the classifier.

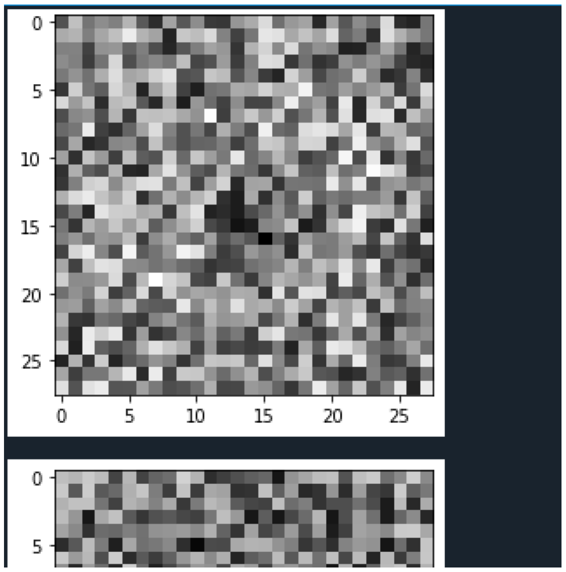
```
 With raw images -->
 Epoch ==> 0 Loss = 2.959013521285601 , Absolute Gradient Sum = 0.009740950140537993
 Epoch ==> 1 Loss = 2.3350999166289013 , Absolute Gradient Sum = 0.009329103350078657
 Epoch ==> 2 Loss = 1.990375663827106 , Absolute Gradient Sum = 0.006896596401695835
 Epoch ==> 3 Loss = 1.3852903835035268 , Absolute Gradient Sum = 0.0052126877325688195
 Epoch ==> 4 Loss = 1.2863363041420053 , Absolute Gradient Sum = 0.005409346316074241
 Epoch ==> 5 Loss = 1.2883063738236238 , Absolute Gradient Sum = 0.0053981791070089715
 Epoch ==> 6 Loss = 1.1765676665874436 , Absolute Gradient Sum = 0.0051653121552358025
 Epoch ==> 7 Loss = 1.124152655221771 , Absolute Gradient Sum = 0.005093456597256774
 Epoch ==> 8 Loss = 1.0762090879748238 , Absolute Gradient Sum = 0.004940342595808559
 Epoch ==> 9 Loss = 1.0245492069982505 , Absolute Gradient Sum = 0.004520201479822152
Number of epochs:  10
Accuracy for raw images on train set =  0.6925
Accuracy for raw images on test set =  0.7039
Training time for raw images =  66.52255296707153  seconds
```

```
With projected images -->
Epoch ==>  0 Loss =  2.2999052751121645 , Absolute Gradient Sum =  0.006158072568541135
Epoch ==>  1 Loss =  2.272903417745532 , Absolute Gradient Sum =  0.005996761219110199
Epoch ==>  2 Loss =  2.247390261522414 , Absolute Gradient Sum =  0.00584031194656885
Epoch ==>  3 Loss =  2.2232716156409222 , Absolute Gradient Sum =  0.005688787483043459
Epoch ==>  4 Loss =  2.200458062765024 , Absolute Gradient Sum =  0.005543337710011503
Epoch ==>  5 Loss =  2.1788650096543867 , Absolute Gradient Sum =  0.005403309807968939
Epoch ==>  6 Loss =  2.158412681810948 , Absolute Gradient Sum =  0.005268310860965197
Epoch ==>  7 Loss =  2.139026068360716 , Absolute Gradient Sum =  0.005138709187375465
Epoch ==>  8 Loss =  2.120634823895938 , Absolute Gradient Sum =  0.005013518061238266
Epoch ==>  9 Loss =  2.103173134254224 , Absolute Gradient Sum =  0.004892643037155932
Number of epochs:  10
Accuracy for raw images on train set =  0.2861
Accuracy for projected images on test set =  0.0675
Training time for projected images =  27.528839826583862  seconds
```

2.6 What do Neural Networks learn?

It generally performs supervised learning tasks.

I have attached one of the outputs over here.


0
5
10
15
20
25
0
5
10
15
20
25
0
5