Question 1

Part (i)

We used Matplotlib to generate Figure 1. We can see that, the countours of the plot are approximately elliptical in nature, so we can roughly conclude that a bivariate normal distribiution generated the data.

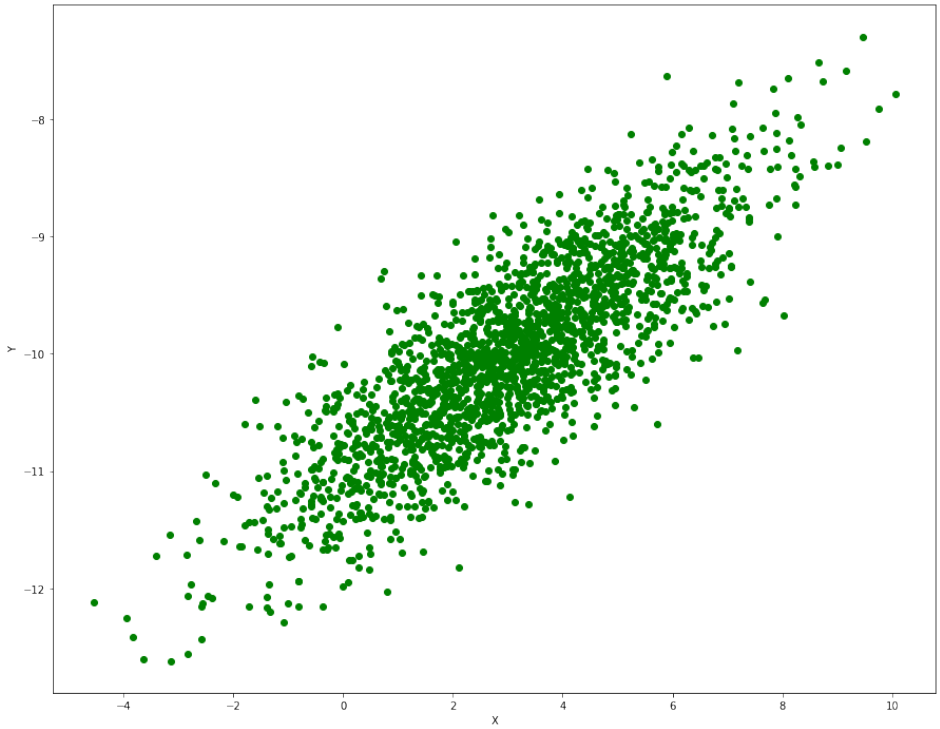

Figure 1: Plot of Data1

Part (ii)

We know that the MLE estimators of the mean and the covariance matrix for the multivariate gaussian distribution are as follows.

$$\hat{\mu}_{ML} = \frac{\sum_{i=1}^{n} \mathbf{x_i}}{n} \tag{1}$$

$$\hat{\Sigma}_{ML} = \frac{\sum_{i=1}^{n} (\mathbf{x_i} - \hat{\mu}_{\mathbf{ML}})(\mathbf{x_i} - \hat{\mu}_{\mathbf{ML}})^{\mathbf{T}}}{n} \tag{2}$$

We implement the same in code:

```
mean = np.mean(data_mat,0)
var_covar = ((data_mat-mean).T @ (data_mat-mean))/len(data_mat)
```

We obtained the following mean and co-variance matrices:

$$\hat{\mu}_{ML} = \begin{bmatrix} 3.01386685 & -10.01880925 \end{bmatrix} \tag{3}$$

$$\hat{\Sigma}_{ML} = \begin{bmatrix} 4.6381915 & 1.45437352 \\ 1.45437352 & 0.65570033 \end{bmatrix} \tag{4}$$

Part (iii)

The multivariate Gaussian distribution has the form:

$$f_Y(\mathbf{x}) = \frac{1}{(2\pi)^{\mathrm{D}/2}} \frac{1}{\left|\Sigma\right|^{1/2}} \exp\left\{ -\frac{1}{2}(\mathbf{x}-\mu)^T \Sigma^{-1} (\mathbf{x}-\mu) \right\}$$

From the above formula, we can derive the log-liklihood for $D = 2$ as follows:

$$\mathscr{L}(\mathbf{x}; \mu, \mathbf{\Sigma}) = -nlog(2\pi) - \frac{n}{2} log\left|\Sigma\right| - \frac{1}{2}\sum_{i=1}^{n} (\mathbf{x_i}-\mu)^T \Sigma^{-1} (\mathbf{x_i}-\mu)$$

We implement the above in code:

```
-len(data) * np.log(2*np.pi) - (len(data) / 2) * np.log(np.linalg.det(var_covar)) - 0.5 * np.sum(((data_mat-np.mean(data_mat,0)) @ np.linalg.inv(var_covar)) * (data_mat-np.mean(data_mat,0)))
```

We obtained the likelihood as $\mathscr{L} = -5598.93933795472$

Part (iv)

The formula for the MLE estimator of the mean does not change, it is still given by equation (1) above. So the MLE estimator of mean remains the same and is given by:

$$\hat{\mu}_{ML} = \begin{bmatrix} 3.01386685 & -10.01880925 \end{bmatrix} \tag{5}$$

Part (v)

We are given that the covariance matrix is

$$\Sigma = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{6}$$

We calculated the loglikelihood function for all possible components of μ between -10 and 10, i.e we calculated the loglikleihood for $11 * 11 = 121$ values of μ and plotted the results on a heatmap (Fig 2). We can see from the plot that the values of loglikelihood are well below what we obtained in Part (ii), the reason being that the covariance matrix has been fixed to a "suboptimal" value, indeed we can see from Fig 1 itself, that X and Y have positive correlation, but here we assumed covariance between X and Y to be 0.(eqn (6))

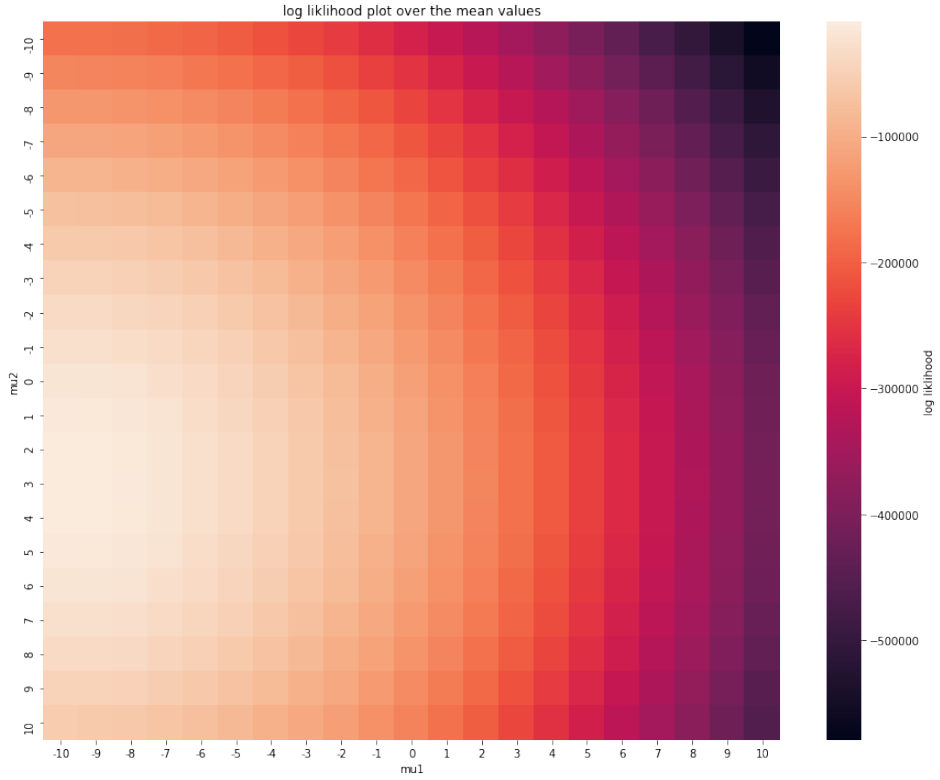

Figure 2: Heatmap of loglikelihood values over μ

Question 2

Part (i)

We implement the EM algorithm given in Bishop, repeated here for completeness:

1. Randomly initialize the means μ_k, covariances Σ_k and the mixing coefficients π_k.
2. **E Step:** Evaluate the responsibility matrix as:

$$\gamma_{nk} = \frac{\pi_k \mathcal{N}(\mathbf{x_n}; \mu_k, \Sigma_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x_n}; \mu_j, \Sigma_j)}$$

3. **M Step:** Update the means,covariance matrices and the mixing coefficients as follows:

$$\mu_{\mathbf{k}}^{\mathbf{new}} = \frac{1}{N_k}\sum_{n=1}^{N}\gamma_{nk}\mathbf{x_n} \tag{7}$$

$$\Sigma_k^{new} = \frac{1}{N_k}\sum_{n=1}^{N}\gamma_{nk}(\mathbf{x_n} - \mu_k^{new})(\mathbf{x_n} - \mu_k^{new})^T \tag{8}$$

$$\pi_k^{new} = \frac{N_k}{N} \tag{9}$$

$$N_k = \sum_{n=1}^{N}\gamma_{nk} \tag{10}$$

The loglikelihood after simplification is given by:

$$\mathcal{L}(\mathbf{X};\mu,\mathbf{\Sigma},\pi) = \sum_{n=1}^{N} log\Big\{\sum_{k=1}^{K}\pi_k\mathcal{N}(\mathbf{x_n};\mu_k,\Sigma_k)\Big\}$$

We implemented the above pseudocode in Python, the important part of which is given below for ready reference:

```
#### Code assumes k is already given ####
# function to evaluate the normal density for ease of use
def norm(x,mu,var):
    return (1 / (np.sqrt(2*np.pi) * np.sqrt(var))) * np.exp(-(x -↘
    → mu)**2/(2*(var)))
# let us vectorize the above function
norm_v = np.vectorize(norm)
# good initialization for mean and variance
mean = np.arange(k)+1
var = (np.arange(k)+1)*0.1
# define the number of iterations here
num_iters = 100
# save the log-liklihood cost here
log_like = []
for i in range(num_iters):
    # E step: first compute the responsibilities matrix
    respon_matrix = norm_v(data_mat,mean.flatten(),var.flatten())↘
    → * mix_coeff
    log_like.append(np.sum(np.log(np.sum(respon_matrix,1))))
    # normalize it
    respon_matrix = respon_matrix / np.sum(respon_matrix,1).↘
    →reshape(-1,1)
```

```
    # we have now computed the responsibilities for each data point
    # M step: now update the means, variances and mix coefficients
    mean = np.sum(respon_matrix * data_mat,0) / np.sum(respon_matrix,0)
    var = np.sum(respon_matrix * (data_mat - mean)**2,0) / np.sum(respon_matrix,0)
    mix_coeff = np.mean(respon_matrix,0)
```

The above code runs the EM algorithm for GMM. We run the algorithm with user given num_iter, and check for convergence of loglikelihood by plotting it against the number of iterations.

Part (ii)

The above algorithm was slightly modified to automatically calculate various parameters for different k's. The modified code for ready reference is:

```
# evaluate for different k's
final_log_like = []
for k in range(1,11):
    mean = np.arange(k)+1
    var = (np.arange(k)+1)*0.1
    mix_coeff = np.array([1/k]*k)
    # define the number of iterations here
    num_iters = 100
    # save the log-liklihood cost here
    log_like = []
    for i in range(num_iters):
        # E step: first compute the responsibilities matrix
        respon_matrix = norm_v(data_mat,mean.flatten(),var.flatten()) * mix_coeff
        log_like.append(np.sum(np.log(np.sum(respon_matrix,1))))
        # normalize it
        respon_matrix = respon_matrix / np.sum(respon_matrix,1).reshape(-1,1)
        # we have now computed the responsibilities for each data point
        # M step: now update the means, variances and mix coefficients
        mean = np.sum(respon_matrix * data_mat,0) / np.sum(respon_matrix,0)
```

```
        var = np.sum(respon_matrix * (data_mat - mean)**2,0) / np.sum(respon_matrix,0)
        mix_coeff = np.mean(respon_matrix,0)
    final_log_like.append(log_like[-1])
    print(k,"mean",mean)
    print(k,"var",var)
    print(k,"mix_coeff",mix_coeff)
```

Ouput:

```
k = 1 mean is : [-1.03289019]
k = 1 var is: [17.01069807]
k = 1 mix_coeff is: [1.]
k = 2 mean is : [-0.91832246 -1.03374258]
k = 2 var is: [6.43788228e-04 1.71371558e+01]
k = 2 mix_coeff is: [0.00738513 0.99261487]
k = 3 mean is : [-0.1857632  -4.98581719  5.10017204]
k = 3 var is: [2.16573954 1.09146911 1.25609185]
k = 3 mix_coeff is: [0.34413132 0.42772268 0.228146  ]
k = 4 mean is : [ 0.12422088 -1.29822342 -5.02288597  5.10537926]
k = 4 var is: [1.92205502 2.27269601 1.04109424 1.24440383]
k = 4 mix_coeff is: [0.2547258  0.09999999 0.41755222 0.22772198]
k = 5 mean is : [-0.18982015  0.27561947 -5.85517826 -4.56348991
 5.12098143]
k = 5 var is: [2.0807506  4.75062385 0.49522582 0.8410026
1.2260216 ]
k = 5 mix_coeff is: [0.31910693 0.02820008 0.13907692 0.28870837
0.2249077 ]
k = 6 mean is : [-0.24984672  0.56371987 -1.99291025 -5.04382395
 0.9990028   5.25157707]
k = 6 var is: [1.11907225 4.46246301 3.06698218 1.02660857
4.11855704 1.064793  ]
k = 6 mix_coeff is: [0.15811503 0.02054446 0.08095866 0.40513072
0.13187607 0.20337506]
k = 7 mean is : [-0.2635948   0.06958083 -1.99205981 -5.04574874
 0.37561309  1.35157254
  5.25960095]
k = 7 var is: [1.03742342 4.33937805 3.08053895 1.02411967
4.36770295 3.90610975
 1.05393618]
k = 7 mix_coeff is: [0.14378887 0.0163426  0.07297425 0.40462648
0.09208631 0.06774473
 0.20243676]
```

```
k = 8 mean is : [-0.24849264  0.02306649 -1.90954171 -5.04139972 ↘
→ 0.27898978  0.92932931
  4.3970377   5.84712817]
k = 8 var is: [1.09548631 3.93028668 3.19237179 1.02795396 ↘
→3.91510365 3.63061767
 0.54475196 0.58100099]
k = 8 mix_coeff is: [0.13842402 0.01659564 0.06992208 0.40654622 ↘
→0.09228777 0.06625703
 0.09021637 0.11975088]
k = 9 mean is : [-0.25803685  0.04577774 -1.89504965 -5.04026254 ↘
→ 0.29835064  0.90366709
  4.50882162  4.98129824  6.33856053]
k = 9 var is: [1.09085041 3.96169783 3.20912013 1.02928955 ↘
→3.94224376 3.65078129
 0.70088682 0.65527592 0.26926277]
k = 9 mix_coeff is: [0.13721126 0.01674184 0.06904119 0.4069346  ↘
→0.09315945 0.06745831
 0.05930851 0.09152616 0.05861867]
k = 10 mean is : [-0.24124024  0.01165026 -1.87933769 -5.04057148↘
→  0.25843128  0.84488918
  4.4930124   4.96200687  5.59034044  6.35862118]
k = 10 var is: [1.10171802 3.87951598 3.22348159 1.02872214 ↘
→3.91962196 3.80874835
 0.76294992 0.88435441 0.95883694 0.05149024]
k = 10 mix_coeff is: [0.13863569 0.01659832 0.06830445 0.40693276↘
→ 0.09207785 0.06632919
 0.04328227 0.0837673  0.05810338 0.0259688 ]
Final loglikelihood for each k: [-1417.9298776801775, ↘
→-1416.7164239276176, -1287.1492898382862, -1287.170202057349, ↘
→-1285.5766778171087, -1286.8007964155915, -1286.8189810722547, ↘
→-1285.864807127587, -1285.0634704348636, -1284.3202966818856]
```

We listed for each k, the means,variances and the mix coefficients. Finally we also listed the converged loglikelihood values for each k.

Note: The above values might change slightly for each run due to the stochastic nature of the EM algorithm.

Now, the convergence of the loglikelihood was checked by plotting the logliklihood function versus the number of iterations, for each k. We include the plot for k = 3 in Fig 3, similar plots were checked for all values of k, we do not list them to keep the length of the report reasonable. We can see from Fig 3 that the loglikelihood converges beautifully.

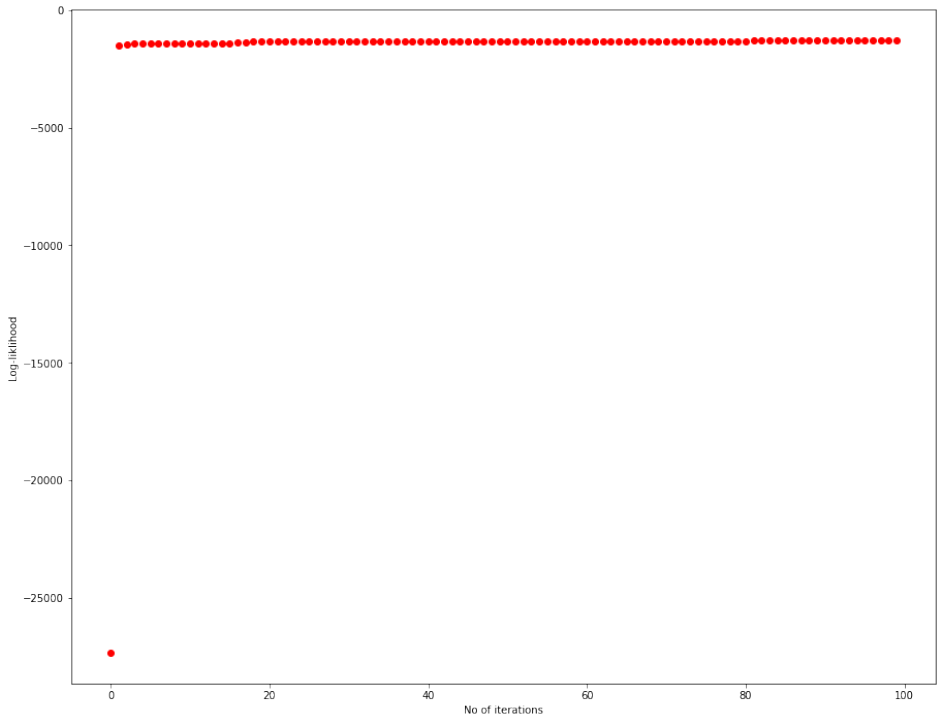

Figure 3: Plot of loglikelihood v/s number of iterations

Part (iii)

The above generated final values of loglikelihood were plotted against k in Fig 4. From the elbow method, we can see that $\mathbf{k = 3}$ makes sense, since taking higher values of k does not increase the loglikelihood by much. Indeed since the data is one dimensional let us visualize it in Fig 5. In Fig 5 we can clearly see that the data belongs to mixture of 3 gaussians, thus confirming our earlier value from elbow method.

Question 3

Part (i)

We are required to run PCA algorithm on this data set which we can do using the following code:(Since the number of components are not given we take it as 1)

```
# center the data
data_mat = data_mat - np.mean(data_mat,0)
# compute the variance covariance matrix
var_covar = data_mat.T @ data_mat / len(data_mat)
# perform eigenvalue decomposition of the variance-covariance ↘
→matrix and get the eigenvector corresponding to greatest ↘
```

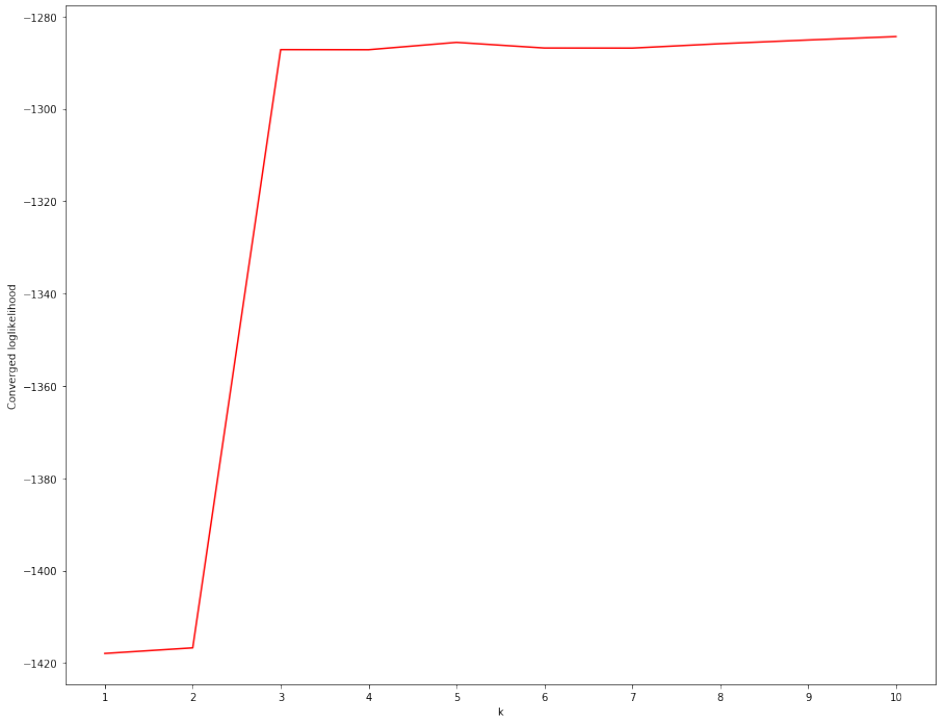

Figure 4: Final loglikelihood v/s k

```
→eigenvalue
pc_vec = np.linalg.eig(var_covar)[1][:,1].reshape(-1,1)
# now just project the actual data onto the principal components
projec = data_mat @ pc_vec
# get the reconstructed datapoints for plotting
actual_projec = actual_projec = projec @ pc_vec.T
# plot the datapoints
plt.figure(figsize=(18,12))
plt.scatter(actual_projec[:,0],actual_projec[:,1],c='g',label="↘
→PCA")
plt.scatter(data_mat[:,0],data_mat[:,1],c='r',label="actual_data"↘
→)
plt.xlabel("X")
plt.ylabel("Y")
plt.legend()
```

We obtain the plot in Fig 6. The eigenvalues are given by $14.48960475, 17.1319144$. We know that the explained variance of each component(say k^{th} component and there are n components) is given by $\frac{\lambda_k}{\sum_{i=1}^{n}\lambda_i}$. So the explained variance of the first principal component is $\frac{17.1319144}{17.1319144+14.48960475} = 0.54$ and the second principal component is $\frac{14.48960475}{17.1319144+14.48960475} = 0.46$.

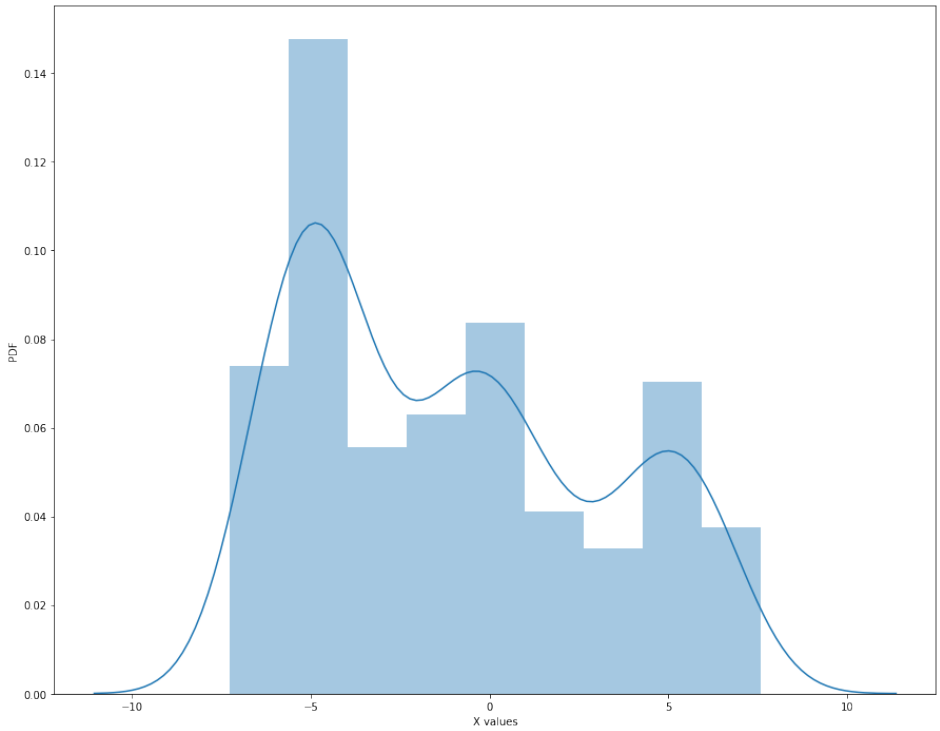

Figure 5: Distribution of Data

So we can see that if we use only first principal component, not much of the variance is explained due to the highly non-linear nature of data as evidenced by Fig 6.

Part (ii)

A.

Here is a rough sketch of the algorithm:

```
1. Center the data
2. Compute the kernel matrix using given kernel function
3. Center the kernel matrix
4. Get the eigenvalue decomposition of the centered kernel matrix
5. The eigenvectors of the centered kernel matrix scaled by the ↘
→root of the eigenvalues are the projected components.
```

We implement the same code, given below for ready reference:

```
# specify the degree
degree = 3
# center the data
data_mat -= np.mean(data_mat,0)
```

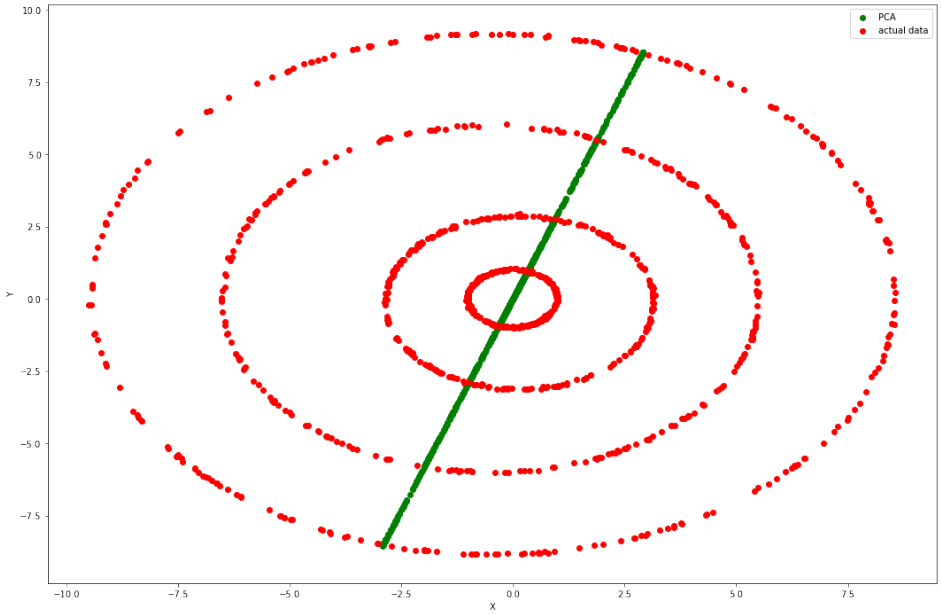

Figure 6: Plot of Data and it's projection on 1^{st} PC

```
# compute the kernel matrix
kernel_matrix = (1 + data_mat @ data_mat.T)**degree
# center the kernel matrix
one_n = np.ones(kernel_matrix.shape) / len(data_mat)
centered_kernel_matrix = kernel_matrix - one_n@kernel_matrix - ↘
→kernel_matrix@one_n + one_n@kernel_matrix@one_n
# get the eigenvalues and eigenvectors of the kernel matrix
eigen_values = np.linalg.eigh(centered_kernel_matrix)↘
→[0][-2:][::-1]
eigen_vectors = np.linalg.eigh(centered_kernel_matrix)[1][:,-2:]
# swap the columns, to get the greatest eigenvector first
eigen_vectors[:,[0, 1]] = eigen_vectors[:,[1, 0]]
# scale the eigenvectors to get transformed data
trans_data = eigen_vectors * np.sqrt(eigen_values)
plt.figure(figsize=(18,12))
plt.scatter(trans_data[:,0],trans_data[:,1],c='g',label="↘
→Projected data top 2 components, degree = "+str(degree))
plt.xlabel("Projected Axis 1")
plt.ylabel("Projected Axis 2")
plt.legend()
```

Using the above code, we plotted the projected components of the top 2 eigenvectors for polynomial kernel in Fig 7 and 8.

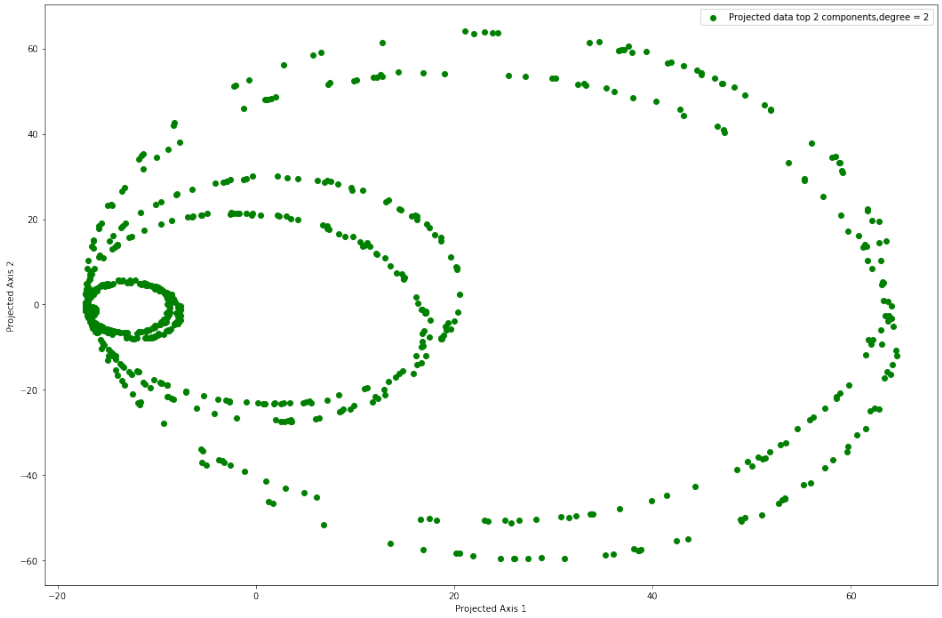

Figure 7: polynomial kernel,degree=2

B.

For the gaussian kernel we just need to change the code for the calculation of kernel matrix, rest of the code is the same. Here is the modified code:

```
# loop over values of sigma
for sigma in [0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1]:
    # vectorized implementation of the kernel matrix
    kernel_matrix = np.exp(-(np.sum(data_mat**2,1).reshape(-1,1) ↘
    →+ np.sum(data_mat**2,1).reshape(1,-1) - 2*data_mat @ ↘
    →data_mat.T)/(2*sigma**2))
    # we now have to center the kernel matrix
    one_n = np.ones(kernel_matrix.shape) / len(kernel_matrix)
    # center the kernel matrix
    centered_kernel_matrix = kernel_matrix - one_n@kernel_matrix ↘
    →- kernel_matrix@one_n + one_n@kernel_matrix@one_n
    # do eigenvalue deomposition
    eigen_values = np.linalg.eigh(centered_kernel_matrix)↘
    →[0][-2:][::-1]
    eigen_vectors = np.linalg.eigh(centered_kernel_matrix)↘
    →[1][:,-2:]
    # swap the columns, to get the greatest eigenvector first
    eigen_vectors[:,[0, 1]] = eigen_vectors[:,[1, 0]]
    pc_vec = eigen_vectors * np.sqrt(eigen_values)
```

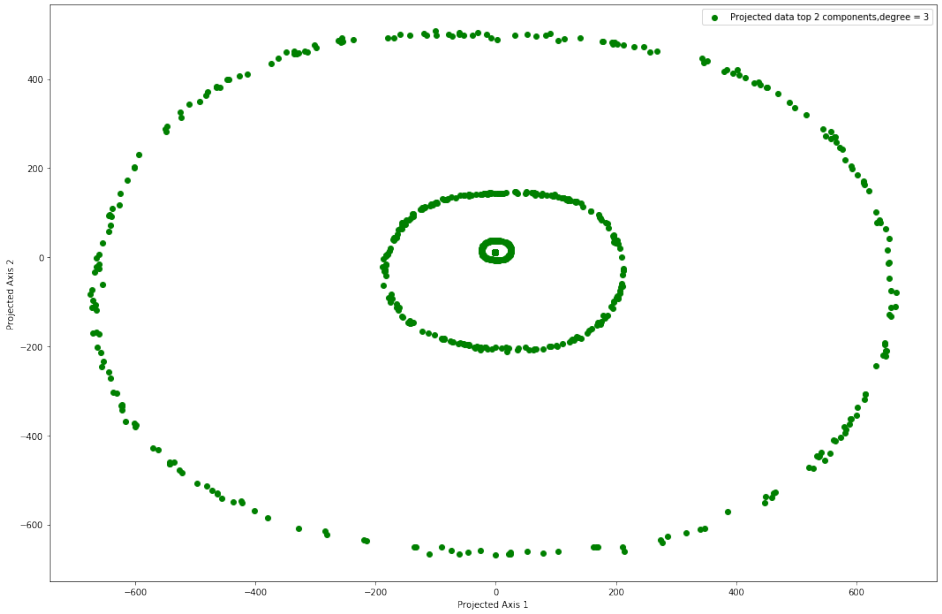

Figure 8: polynomial kernel,degree=3

```
plt.figure(figsize=(18,12))
plt.scatter(pc_vec[:,0],pc_vec[:,1],c='g',label="Projected data top 2 components,sigma = "+str(sigma))
plt.xlabel("Projected Axis 1")
plt.ylabel("Projected Axis 2")
plt.legend()
```

We plotted the projected components of the top 2 eigenvectors for gaussian kernel for various values of σ in Fig 9 through 18.

Part (iii)

Qualitatively we feel that the structure of the actual data plotted in Fig 6 is best described by Fig 8. So Polynomial kernel with degree = 3 seems to be the best kernel.

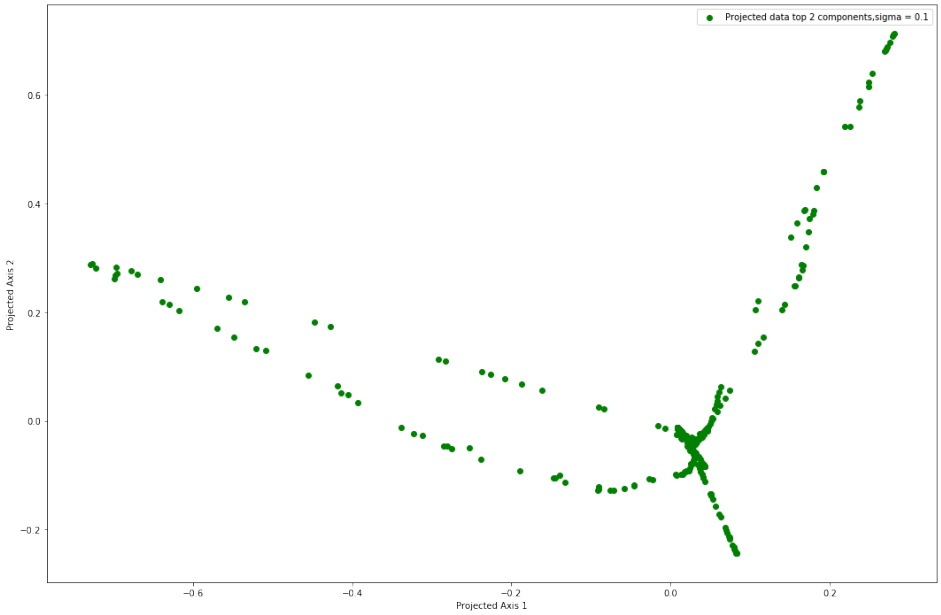

Figure 9: gaussian kernel,sigma=0.1

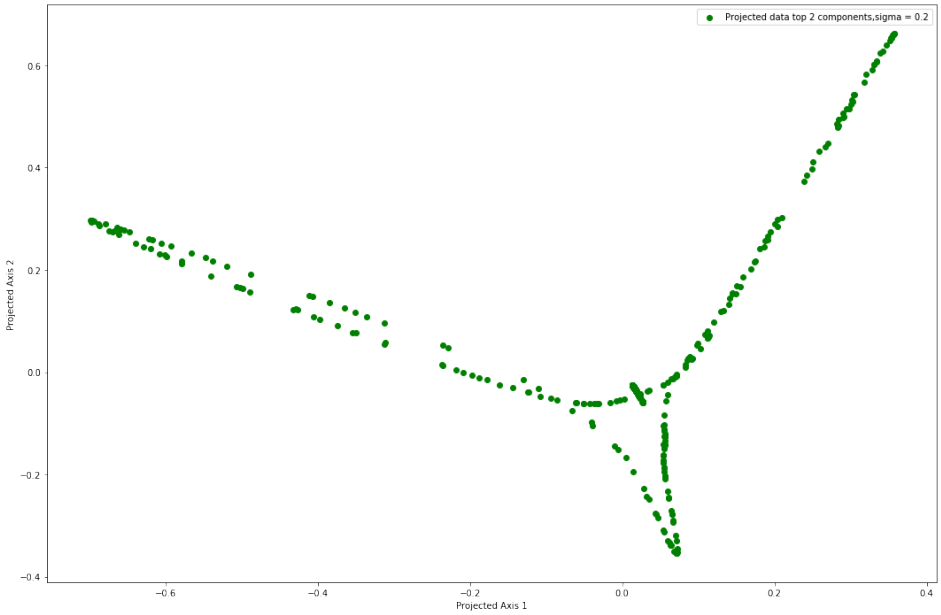

Figure 10: gaussian kernel,sigma=0.2

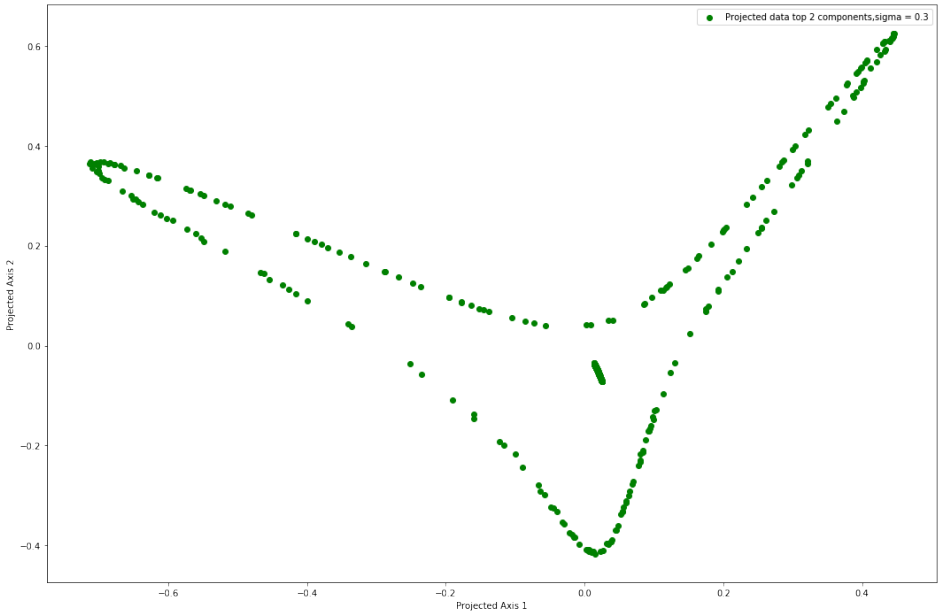

Figure 11: gaussian kernel,sigma=0.3

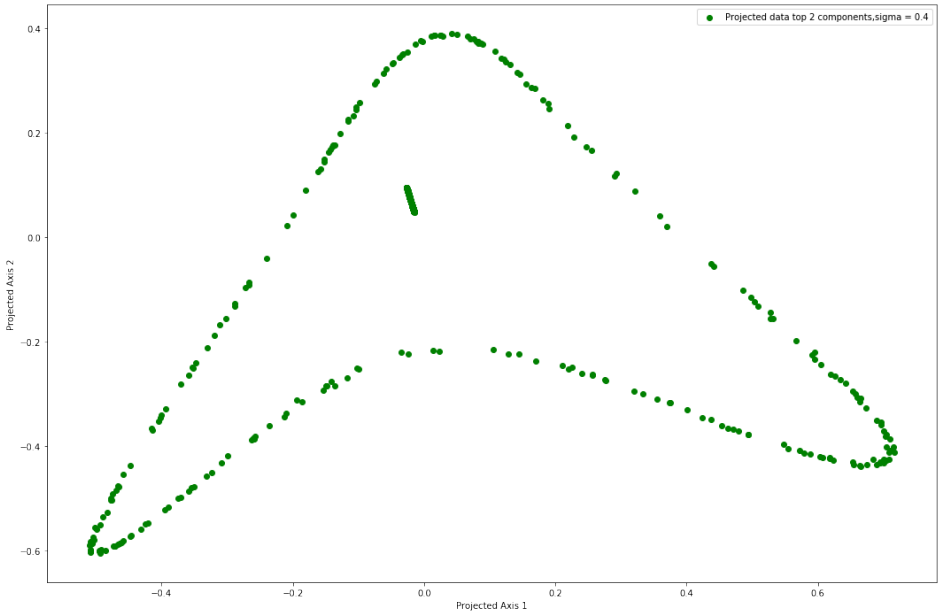

Figure 12: gaussian kernel,,sigma=0.4

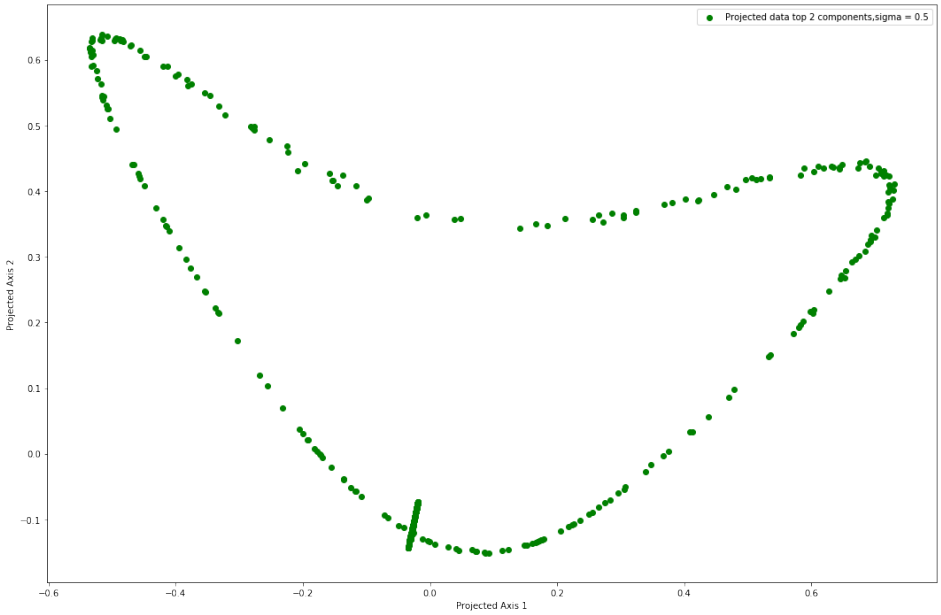

Figure 13: gaussian kernel,sigma=0.5

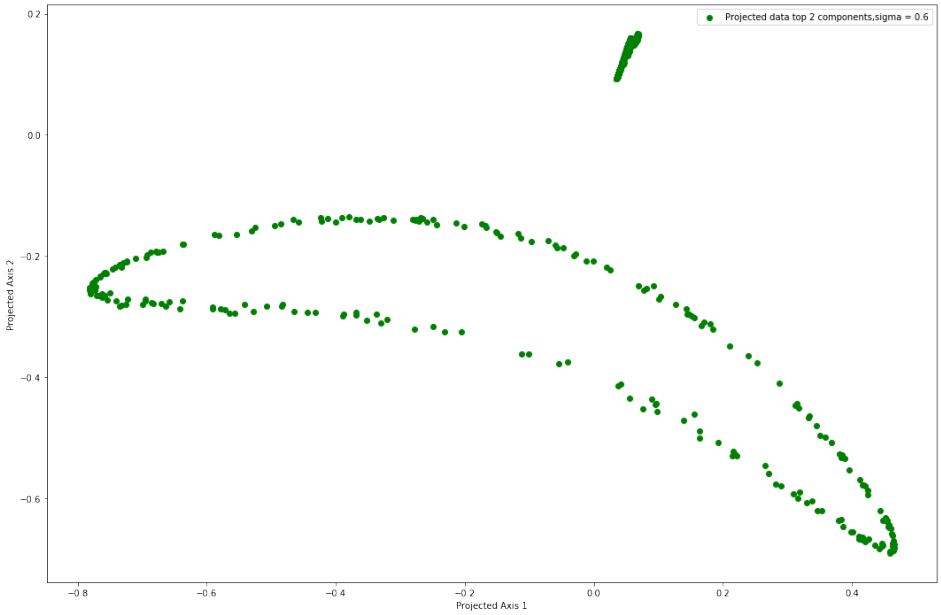

Figure 14: gaussian kernel,sigma=0.6

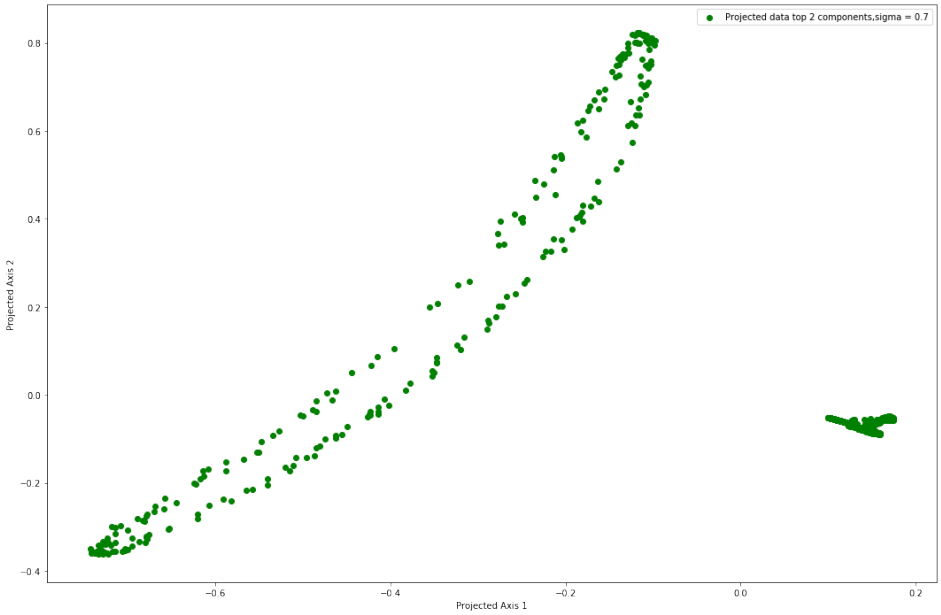

Figure 15: gaussian kernel,sigma=0.7

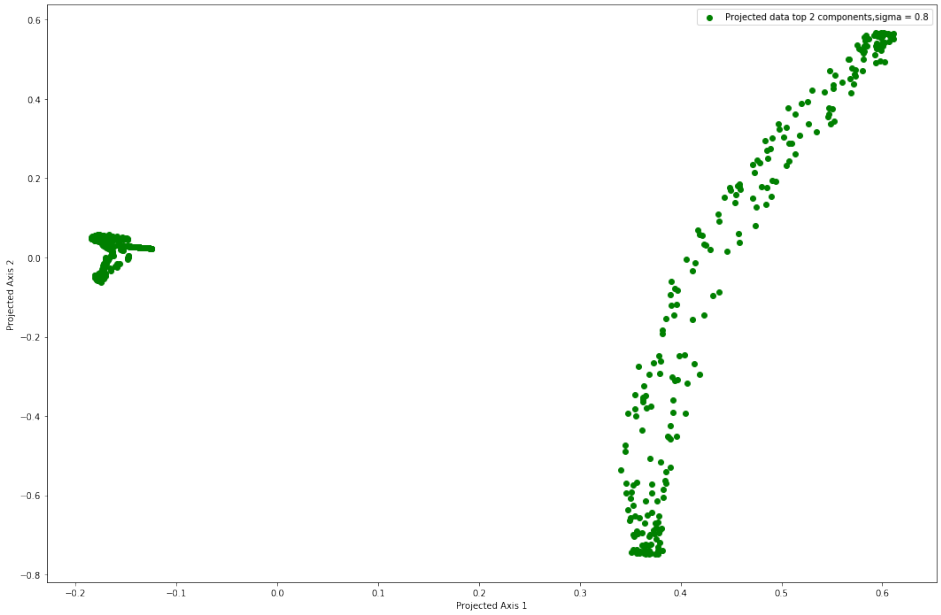

Figure 16: gaussian kernel,sigma=0.8

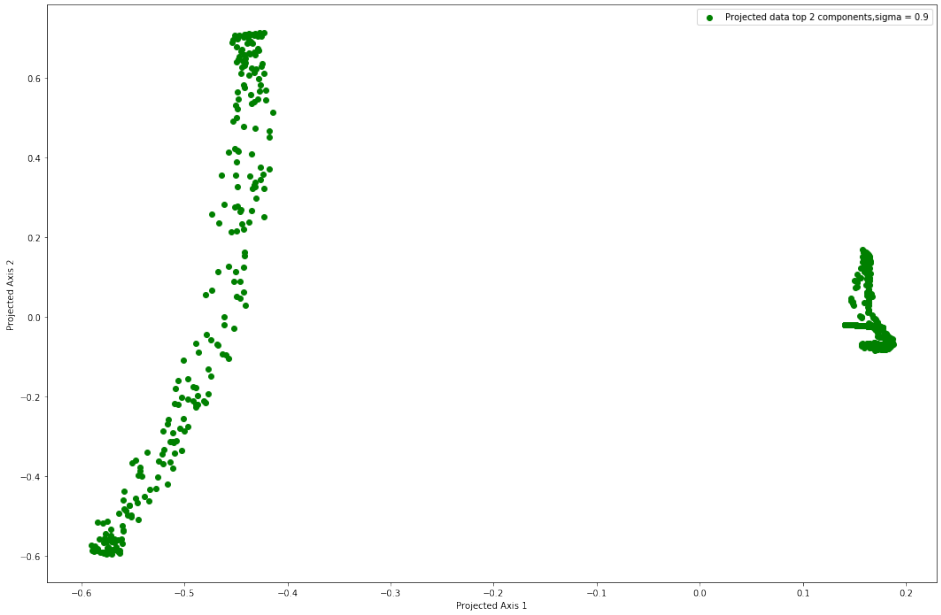

Figure 17: gaussian kernel,sigma=0.9

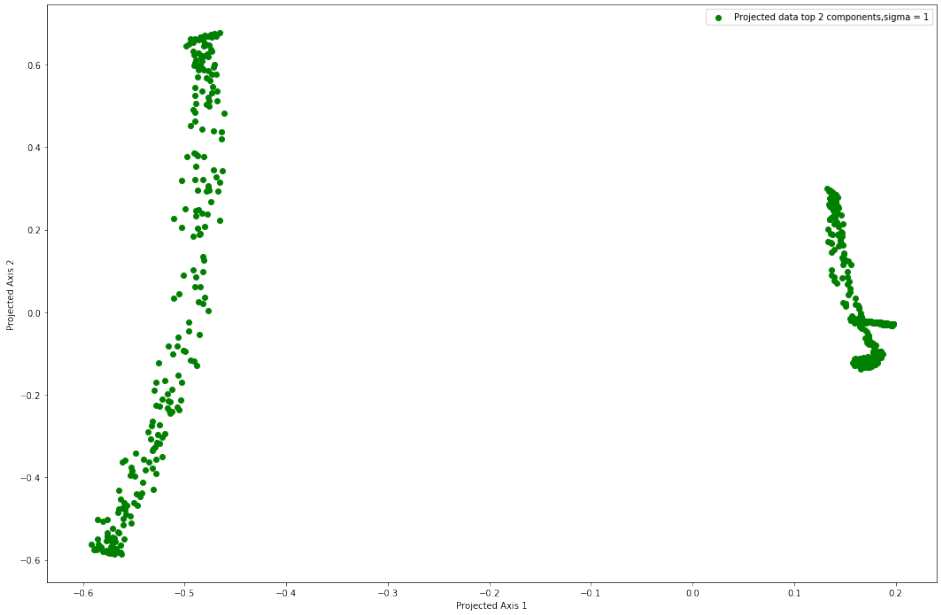

Figure 18: gaussian kernel,sigma=1