Assignment 2 Report

CS18S012
CE16B034

October 29, 2019

Problem 1

Part (i)

The plot of the data set is shown in Fig 1. We can see that it has a highly non linear structure and hence running Kmeans on it might not give us good results. We also found that the decision region changes from each iteration due to the changes in random initialization. We give the informal pseudocode for Kmeans below:

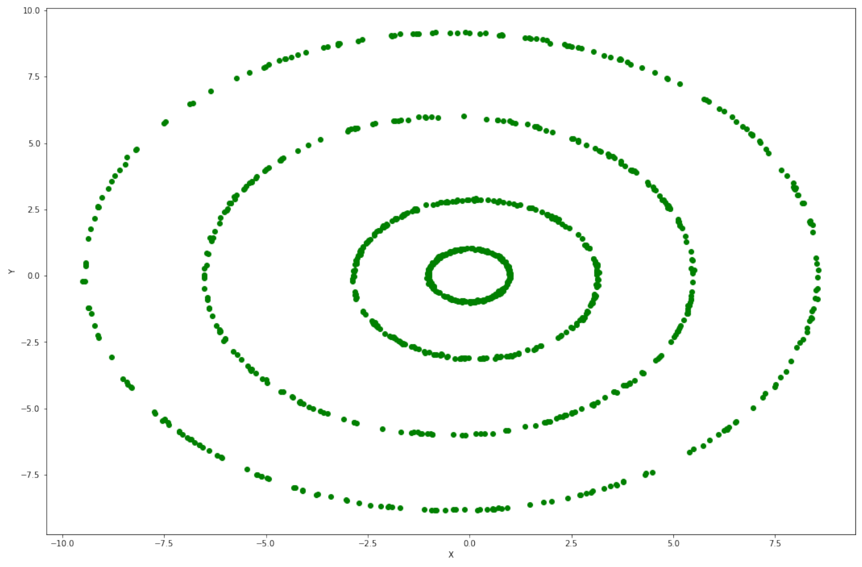

Figure 1: Plot of Data

Note: We assume k as the the number of clusters given to us. clustermembers is an array containing the cluster membership of the data points, clustermembers[i] gives us the cluster of the i-th data point(integer between 0 and k-1). clustermeans is an array that contains our cluster centers, clustermeans[i] gives us the cluster center of the i-th cluster.

We also give the cost function J below:

$$J = \frac{1}{N}\sum_{i=1}^{N} \|X[i] - clustermeans[clustermembers[i]]\|^2 \tag{1}$$

where X[i] gives us the i-th data point ,[] notation means the familiar array/matrix indexing notation and N stands for number of data points.

1. Initialize clustermembers randomly.
2. Repeat until convergence:
   A)Update the cluster center of each cluster as the mean of points contained in that particular cluster.
   B)Reassign each point to the nearest cluster.(Euclidean distance)

The code implemented in Python is given below:

```
# plot for 5 different initilizations
for u in range(5):
    # fix a value of k
    k = 4
    # the k means algorithm for given k
    # Let us define and initialize some required arrays and matrices
    # membership vector members( 1 * num_data_points) contains the ↘
    →membership of each point
    cluster_members = np.zeros(len(data),dtype=int)
    # cluster_means (num of clusters * dim of data) contains the means↘
    → of each cluster
    cluster_means = np.zeros((k,data.shape[1]))
    # predefine num of iterations
    num_iters = 20
    # do random initialzation of the members
    for i in range(len(cluster_members)):
        cluster_members[i] = i%k
    # randomly shuffle the numbers
    np.random.shuffle(cluster_members)
    # store the distortion measure in a list
    distortion_measure = []
    for i in range(num_iters):
        # initialise current distortion measure
        curr_cost = 0
        #M step : Update the cluster_means
        for j in range(len(cluster_means)):
            # check if slice is empty, if empty just use previous ↘
            →value of mean
            cluster_means[j] = np.mean(X[cluster_members==j],0) if len↘
            →(X[cluster_members==j]) else cluster_means[j]
        # E step : Update the cluster_members
        for j in range(len(data)):
            cluster_members[j] = np.argmin(np.sum((cluster_means - X[j↘
            →])**2,1))
            curr_cost += np.sum((cluster_means[cluster_members[j]] - X↘
            →[j])**2)
        distortion_measure.append(curr_cost)
    # plot a decision region
    x_min,x_max = min(X[:,0])-1,max(X[:,0])+1
    y_min,y_max = min(X[:,1])-1,max(X[:,1])+1
    # create a grid of points for plotting decision regions
    x_plot,y_plot = np.meshgrid(np.arange(x_min,x_max,0.1),np.arange(↘
    →y_min,y_max,0.1))
    # create predictions for these points in grids
    data_plot = np.c_[x_plot.ravel(),y_plot.ravel()]
    pred_plot = np.zeros(len(data_plot),dtype=int)
    plt.figure(figsize=(18,12))
    for i in range(len(data_plot)):
        pred_plot[i] = np.argmin(np.sum((cluster_means - data_plot[i])↘
        →**2,1))
    pred_plot = pred_plot.reshape(x_plot.shape)
    plt.contourf(x_plot,y_plot,pred_plot,alpha=0.3)
    plt.scatter(X[:,0],X[:,1],c=cluster_members,label = "Random␣↘
    →initialization␣"+str(u+1))
    plt.xlabel("X")
    plt.ylabel("Y")
    plt.legend()
    plt.figure(figsize=(18,12))
    plt.scatter(list(range(1,len(distortion_measure)+1)),↘
    →distortion_measure,c='g',label="Cost␣function␣v/s␣iterations␣for␣↘
    →Random␣initialization␣" + str(u+1))
    plt.xlabel("Iterations")
    plt.ylabel("Cost␣function")
    plt.legend()
```

We plot the decision regions and the cost function v/s iterations for each random initialization in Figures 2 through 11. We see that the decision regions are changing with each initialization and the cost function converges after a few iterations.

Part (ii)

In this question we fix a random initialization for each value of k, runt the Kmeans algorithm for each k, and then plot the decision regions. We slightly modify the above code as follows:

```
for k in [2,3,4,5]:
    cluster_members = np.zeros(len(data),dtype=int)
    # cluster_means (num of clusters * dim of data) contains the means↘
    → of each cluster
```

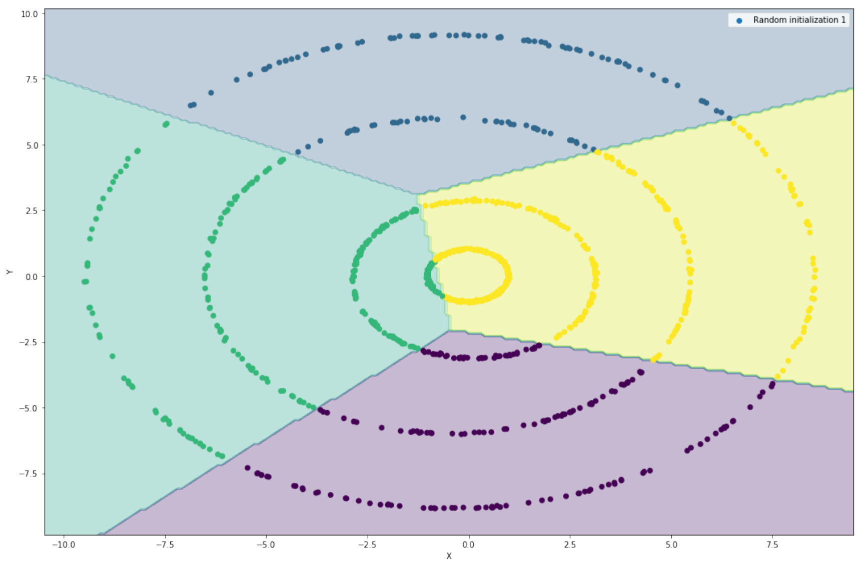

Figure 2:

```
cluster_means = np.zeros((k,data.shape[1]))
# predefine num of iterations
num_iters = 20
# do random initialzation of the members
for i in range(len(cluster_members)):
    cluster_members[i] = i%k
# randomly shuffle the numbers
# this initialization is fixed for this particular k
np.random.shuffle(cluster_members)
# store the distortion measure in a list
distortion_measure = []
for i in range(num_iters):
    # initialise current distortion measure
    curr_cost = 0
    # M step : Update the cluster_means
    for j in range(len(cluster_means)):
        cluster_means[j] = np.mean(X[cluster_members==j],0) if len↘
        →(X[cluster_members==j]) else cluster_means[j]
    # E step : Update the cluster_members
    for j in range(len(data)):
        cluster_members[j] = np.argmin(np.sum((cluster_means - X[j↘
        →])**2,1))
        curr_cost += np.sum((cluster_means[cluster_members[j]] - X↘
        →[j])**2)
    distortion_measure.append(curr_cost)
```

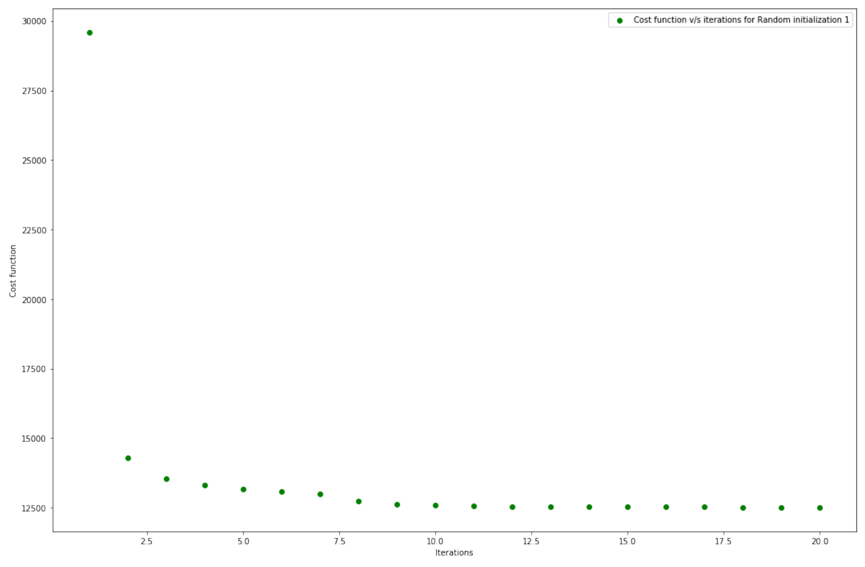

Figure 3:

```
plt.figure(figsize=(18,12))
x_min,x_max = min(X[:,0])-1,max(X[:,0])+1
y_min,y_max = min(X[:,1])-1,max(X[:,1])+1
# create a grid of points for plotting decision regions
x_plot,y_plot = np.meshgrid(np.arange(x_min,x_max,0.1),np.arange(↘
→y_min,y_max,0.1))
# create predictions for these points in grids
data_plot = np.c_[x_plot.ravel(),y_plot.ravel()]
pred_plot = np.zeros(len(data_plot),dtype=int)
for i in range(len(data_plot)):
    pred_plot[i] = np.argmin(np.sum((cluster_means - data_plot[i])↘
    →**2,1))
pred_plot = pred_plot.reshape(x_plot.shape)
plt.contourf(x_plot,y_plot,pred_plot,alpha=0.4)
plt.scatter(X[:,0],X[:,1],c=cluster_members,label="k = "+str(k))
plt.scatter(cluster_means[:,0],cluster_means[:,1],c='r',label="↘
→Cluster centers")
plt.xlabel("X")
plt.ylabel("Y")
```

The scatter plots and the decision regions are given in Figure 12 though 15. The regions in different colours indicate the different clusters. We can see that the KMeans algorithm is failing to capture the structure of the data.
Note: The above plots might change if we run the code again because of the changing nature of the random initialization.

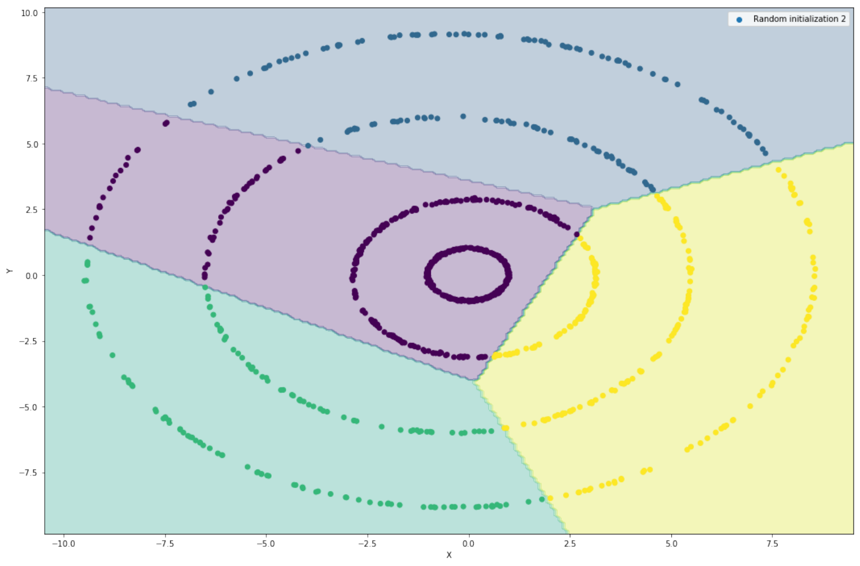

Figure 4:

Part (iii)

For this section we use the Kernel PCA code written in the previous assignment. The informal algorithm for spectral clustering is as follows:

1. Run Kernel PCA on the data set and obtained the transformed data.
2. Run KMeans Clustering on the transformed data set.
3. The points are assigned to clusters as determined by their membership to the clusters in the transformed data.

First let us try running spectral clustering for polynomial kernel, degree = 2 and 3 and dimension of transformed data = 2. The code is given below:

```
## Kernel PCA from previous assignment
## Polynomial kernel
dim = 2
# center the data
for degree in [2,3]:
    X -= np.mean(X,0)
    # compute the kernel matrix
    kernel_matrix = (1 + X @ X.T)**degree
    # center the kernel matrix
    one_n = np.ones(kernel_matrix.shape) / len(X)
    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
```

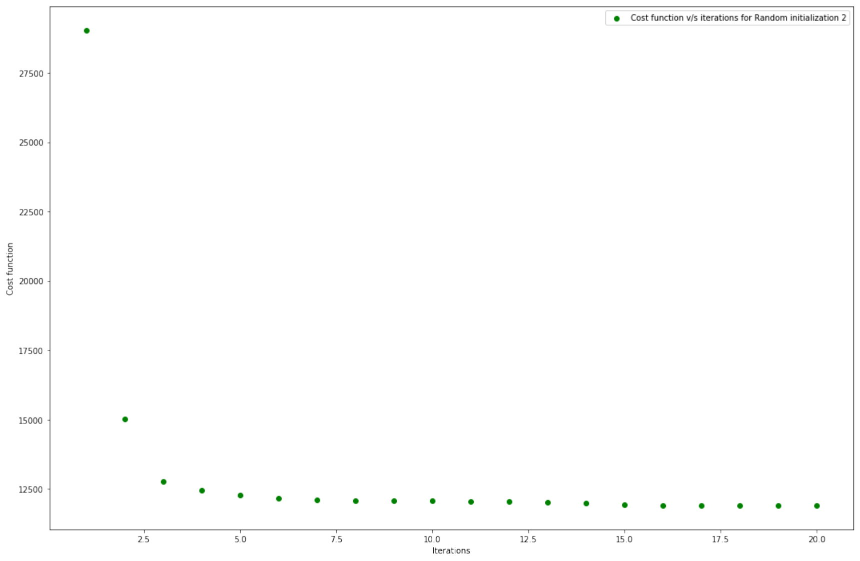

Figure 5:

```
eigen_values = np.linalg.eigh(centered_kernel_matrix)[0][-dim↘
→:][::-1]
eigen_vectors = np.linalg.eigh(centered_kernel_matrix)[1][:,-dim:]
# swap the columns, to get the greatest eigenvector first
eigen_vectors[:,list(range(dim))] = eigen_vectors[:,list(reversed(↘
→list(range(dim))))]
# 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()
## Run k means now on the transformed data
k = 4
cluster_members = np.zeros(len(data),dtype=int)
# cluster_means (num of clusters * dim of data) contains the means↘
→ of each cluster
cluster_means = np.zeros((k,dim))
# predefine num of iterations
num_iters = 20
# do random initialzation of the members
for i in range(len(cluster_members)):
    cluster_members[i] = i%k
```

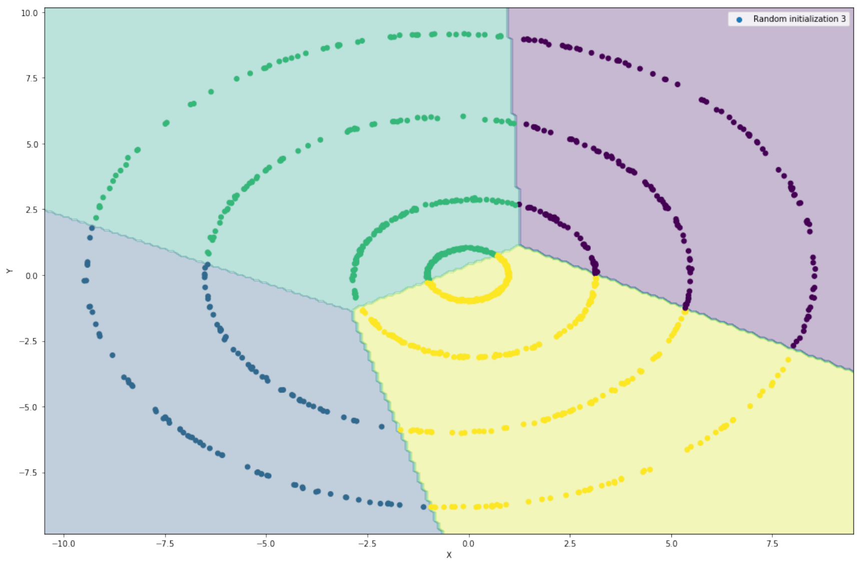

Figure 6:

```
# randomly shuffle the numbers
np.random.shuffle(cluster_members)
# store the distortion measure in a list
distortion_measure = []
for i in range(num_iters):
    # initialise current distortion measure
    curr_cost = 0
    # M step : Update the cluster_means
    for j in range(len(cluster_means)):
        cluster_means[j] = np.mean(trans_data[cluster_members==j↘
        →],0) if len(trans_data[cluster_members==j]) else ↘
        →cluster_means[j]
    # E step : Update the cluster_members
    for j in range(len(data)):
        cluster_members[j] = np.argmin(np.sum((cluster_means - ↘
        →trans_data[j])**2,1))
        curr_cost += np.sum((cluster_means[cluster_members[j]] - ↘
        →trans_data[j])**2)
    distortion_measure.append(curr_cost)
plt.figure(figsize=(18,12))
for i in range(k):
    plt.scatter(X[:,0][cluster_members == i],X[:,1][↘
    →cluster_members == i],label=str(i+1)+" st cluster , degree = "↘
    →+str(degree))
plt.xlabel("X")
```

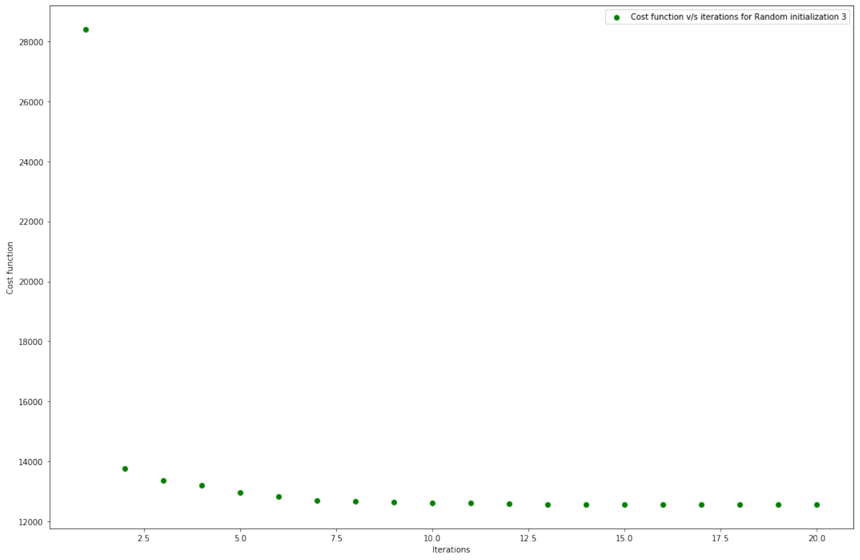

Figure 7:

```
    plt.ylabel("Y")
    plt.legend()
```

The results are given in Fig 16 through 19. We can that it is not successfully separating the 4 clusters(Figs 17 and 19), the reason being that the data in transformed space(Figs 16 and 18) also doesn't have 4 "separable" groups and running KMeans on them would be futile. Briefly we want such a transformation that the data in transformed space should be "easily separable" so that KMeans can cluster them, with this goal in mind we ran Gaussian kernel on data for various values of sigma, we found the case for sigma = 4 interesting, the results of Kernel PCA are given in Fig 20. We can see that the data is definitely 4 separate groups but the variation in the Projected Axis 2(for each group) is not allowing KMeans to separate them properly, but we observe that the Projected Axis 1 shows promise in separating the groups. So let us try Gaussian kernel, sigma = 4, and dimension of projected data = 1. The code is given below:

```
# Gaussian Kernel
sigma = 4
dim = 1
# vectorized implementation of the kernel matrix
kernel_matrix = np.exp(-(np.sum(X**2,1).reshape(-1,1) + np.sum(X**2,1)↘
→.reshape(1,-1) - 2*X @ X.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
```

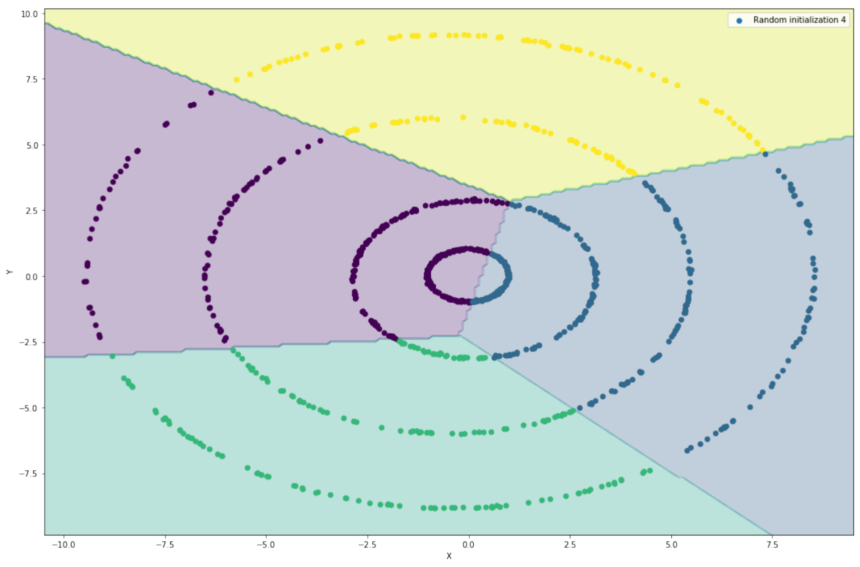

Figure 8:

```
eigen_values = np.linalg.eigh(centered_kernel_matrix)[0][-dim:][::-1]
eigen_vectors = np.linalg.eigh(centered_kernel_matrix)[1][:,-dim:]
# swap the columns, to get the greatest eigenvector first
eigen_vectors[:,list(range(dim))] = eigen_vectors[:,list(reversed(list(range(dim))))]
trans_data = eigen_vectors * np.sqrt(eigen_values)
plt.figure(figsize=(18,12))
plt.scatter(trans_data,[0]*len(trans_data),label="sigma = 4")
plt.xlabel("Projected Axis 1")
plt.ylabel("0")
plt.legend()
## Run k means now on the transformed data
k = 4
cluster_members = np.zeros(len(trans_data),dtype=int)
# cluster_means (num of clusters * dim of data) contains the means of each cluster
cluster_means = np.zeros((k,trans_data.shape[1]))
# predefine num of iterations
num_iters = 50
# do random initialzation of the members
for i in range(len(cluster_members)):
    cluster_members[i] = i%k
# randomly shuffle the numbers
np.random.shuffle(cluster_members)
# store the distortion measure in a list
```

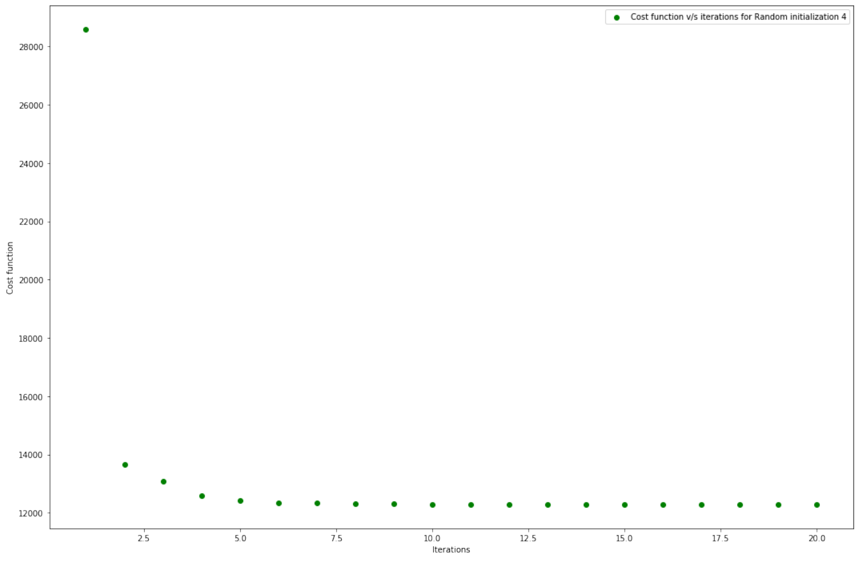

Figure 9:

```
distortion_measure = []
for i in range(num_iters):
    # initialise current distortion measure
    curr_cost = 0
    # M step : Update the cluster_means
    for j in range(len(cluster_means)):
        cluster_means[j] = np.mean(trans_data[cluster_members==j],0) ↘
        →if len(trans_data[cluster_members==j]) else cluster_means[j]
    # E step : Update the cluster_members
    for j in range(len(data)):
        cluster_members[j] = np.argmin(np.sum((cluster_means - ↘
        →trans_data[j])**2,1))
        curr_cost += np.sum((cluster_means[cluster_members[j]] - ↘
        →trans_data[j])**2)
    distortion_measure.append(curr_cost)
plt.figure(figsize=(18,12))
for i in range(k):
    plt.scatter(X[:,0][cluster_members == i],X[:,1][cluster_members ==↘
    → i],label=str(i+1)+" st cluster , sigma = "+str(sigma))
plt.xlabel("X")
plt.ylabel("Y")
plt.legend()
```

The results are given in Fig 21 and 22. We can see from Fig 21 that we managed to succesfully separate the data into 4 groups in transformed space and after runnning KMeans on them, we can see from Fig 22 that the original data has also been separated successfully into four clusters.

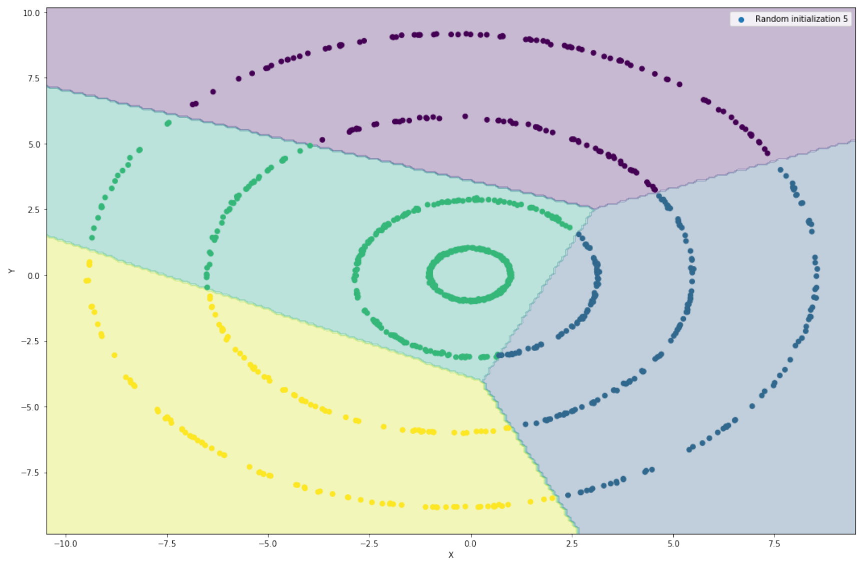

Figure 10:

Problem 2

Part (i)

We know from class that the formula for W_{ML} is given by:

$$\beta = (X^T X)^{-1} X^T y \tag{2}$$

where X is the data matrix and y contains the values. The code is as follows:

```
beta = np.linalg.inv(trans_data.T @ trans_data) @ trans_data.T @ y
```

The beta vector is too long to list here.

Part (ii)

The gradient descent update rule for regression is given by:

$$\beta_{new} = \beta - \eta X^T (X\beta - y) \tag{3}$$

where η is the learning rate. After experimenting with various learning rates we found that the value $\eta = 5 * 10^{-6}$ gives good convergence, we ran it for 2000 iterations. The code is given below for reference:

```
# define number of iterations
num_iter = 2000
# initialise betas
beta_curr = np.zeros((trans_data.shape[1],1))
```

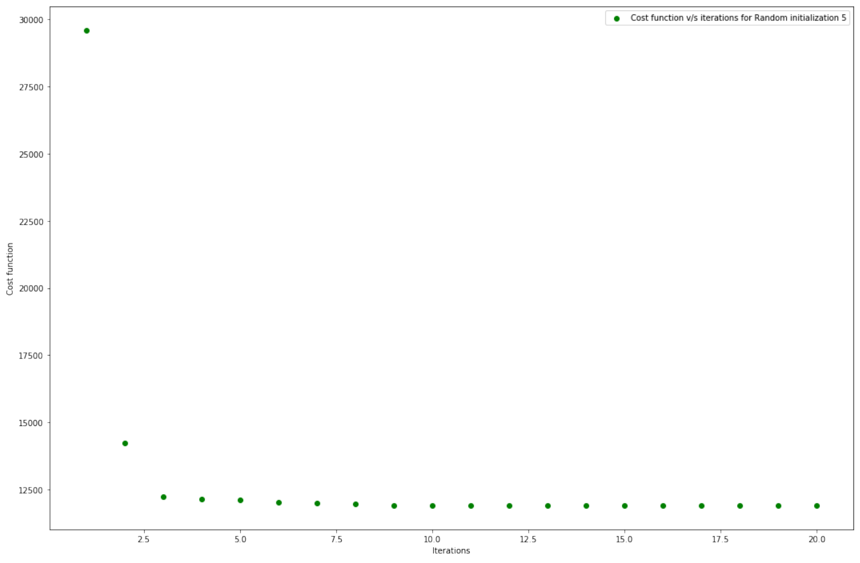

Figure 11:

```
# define step size
neta = 5e-6
# store differences in the norm
diff_norm = []
for i in range(num_iter):
    beta_curr = beta_curr - neta * (trans_data.T@(trans_data@beta_curr
    - y))
    diff_norm.append(np.linalg.norm(beta - beta_curr))
plt.figure(figsize=(18,12))
plt.scatter(list(range(len(diff_norm))),diff_norm,c='g',label="n = 5e
-6")
plt.xlabel("Iterations")
plt.ylabel("||beta - beta_ML||")
plt.legend()
```

The plot for $\|\beta_t - \beta_{ML}\|$ v/s number of iterations is given in Fig 23. We can see from the figure the gradient descent coefficients are converging beautifully to the closed form solution coefficients, also we do not see any fluctuations in the graph.

Part (iii)

The learning rule for Stochastic gradient descent is same as above expect that instead of considering all the points for computing the gradient, we randomly sample 100 points(given in question) and compute the gradient using only those points hoping that the true gradient is "close" to the approximated gradient. The code is the same as above except at each iteration we sample 100 points(without replacement), and we used a learning rate of 5e-4, number of iterations = 1500.

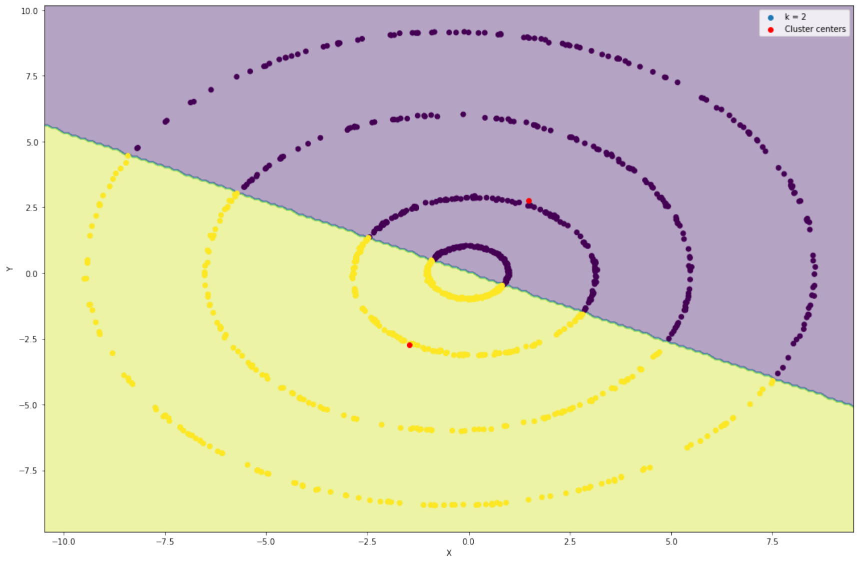

Figure 12:

The code is given below:

```
# define number of iterations
num_iter = 1500
# initialise betas
beta_curr = np.zeros((trans_data.shape[1],1))
# define step size
neta = 5e-4
diff_norm = []
for i in range(num_iter):
    # random sampling of points
    curr_batch = np.random.choice(len(trans_data),100,replace=False)
    beta_curr = beta_curr - neta * (trans_data[curr_batch].T@(↘
    →trans_data[curr_batch]@beta_curr - y[curr_batch]))
    diff_norm.append(np.linalg.norm(beta - beta_curr))
plt.figure(figsize=(18,12))
plt.scatter(list(range(len(diff_norm))),diff_norm,c='g',label="↘
→Stochastic Gradient descent")
plt.xlabel("Iterations")
plt.ylabel("||beta - beta_ML||")
plt.legend()
```

The plot for $\|\beta_t - \beta_{ML}\|$ v/s number of iterations is given in Fig 24, we can see that from the plot that although the norm difference is close to zero at the end of iterations it never truly converges and keeps oscillating because we are approximating the gradient rather than using true gradient. We can also see that the learning rate is higher than gradient decent hence number of iterations

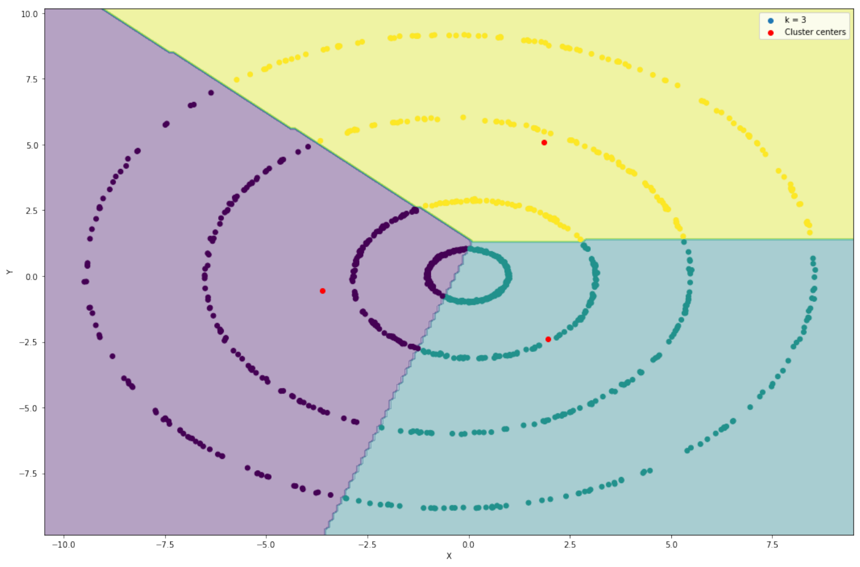

Figure 13:

are lower and hence the algorithm converges quickly than the gradient descent for a reasonable tolerance.

Question 3

Part (i)

The gradient descent update rule for ridge regression is given by:

$$\beta_{new} = \beta - \eta X^T(X\beta - y + \lambda\beta) \tag{4}$$

where η is learning rate and λ is a regularising parameter. The Python code for the update rule is:

```
beta_curr = beta_curr - neta * (trans_data.T@(trans_data@beta_curr - y↘
→) + lam*beta)
```

Part (ii)

We divide the given data set into training and validation sets in the ratio of 4:1, by randomly selecting indices using np.random.choice(), please find code below:

```
# let us create a validation set of 20%
validation_indices = np.random.choice(len(X),int(0.2*len(X)),replace=↘
→False)
X_val,y_val = X[validation_indices],y[validation_indices]
X_train,y_train = X[list(set(range(len(X))) - set(validation_indices))↘
→],y[list(set(range(len(X))) - set(validation_indices))]
```

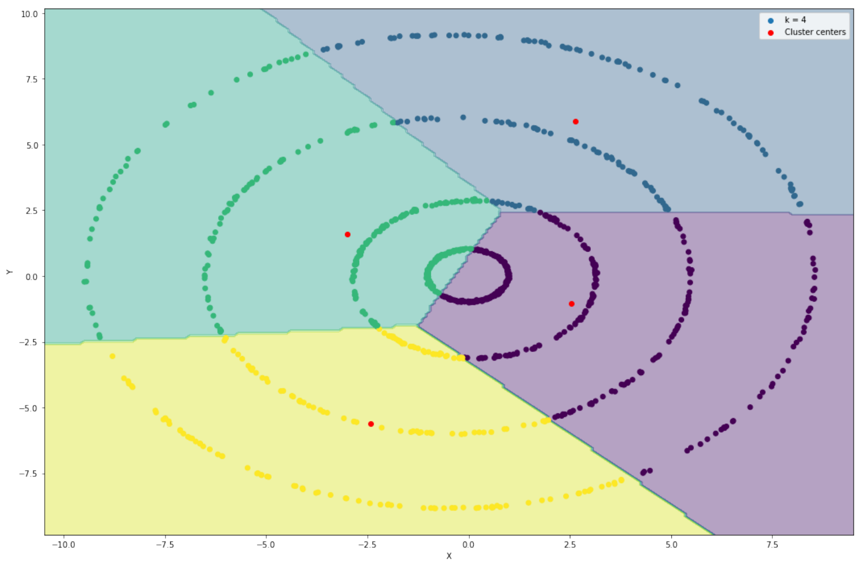

Figure 14:

We do training on the training set for each value of lambda and then we evaluate the loss function on the validation set(for each lambda), then plotting it on a graph will help us determine the optimal value of lambda(it is just the lambda value of minimum validation error). We use same hyperparameters as above(learning rate and number of iterations, we checked for each lambda whether the algorithm has converged). Please find the code below:

```
num_iter = 2000
neta = 5e-6
# store the val error
val_error = []
# let us determine the validation error for various lambda values
for lam in np.linspace(0,5,20):
    beta_curr = np.zeros((X_train.shape[1],1))
    # do training on the training set
    for i in range(num_iter):
        beta_curr = beta_curr - neta * (X_train.T@(X_train@beta_curr -↘
        → y_train) + lam*beta)
    # use validation set to evaluate model performance
    val_error.append(np.linalg.norm(X_val@beta_curr - y_val)**2)
plt.figure(figsize=(18,12))
plt.scatter(np.linspace(0,5,20),val_error,c="g",label="Rigde␣↘
→regression")
plt.xlabel("lambda")
plt.ylabel("Validation␣error")
plt.legend()
```

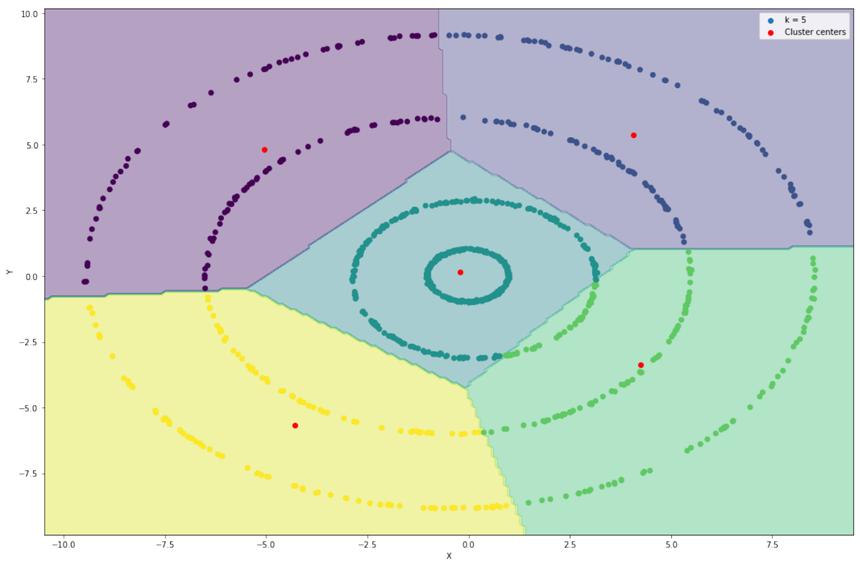

Figure 15:

The validation error plot is given in Fig 25. We can see from the figure that the validation error increases as λ increases, the minimum validation error is at $\lambda = 0$, which makes our optimal solution to be β_{ML}. The test error of β_{ML} was evaluated and it is :

$$validationerror : 80.454963474763 testerror : 185.37790336131047 \tag{5}$$

The reason that β_{ML} outperforms the ridge regressor may be that the increase in bias due to the ridge regressor might be higher than the decrease in the variance thereby increasing mean squared error(ridge regressor might be underfitting the data).

Part (iii)

We code the Coordinate gradient as discussed in class, the code is given below:

```
# coordinate descent for lasso regression
def evaluate_cases(lam, coeff):
    if coeff > lam:
        return coeff-lam
    elif coeff < -lam:
        return coeff+lam
    else:
        return 0
# define number of iterations
num_iter = 1000
# initialise betas
beta_curr = np.zeros((X.shape[1],1))
```

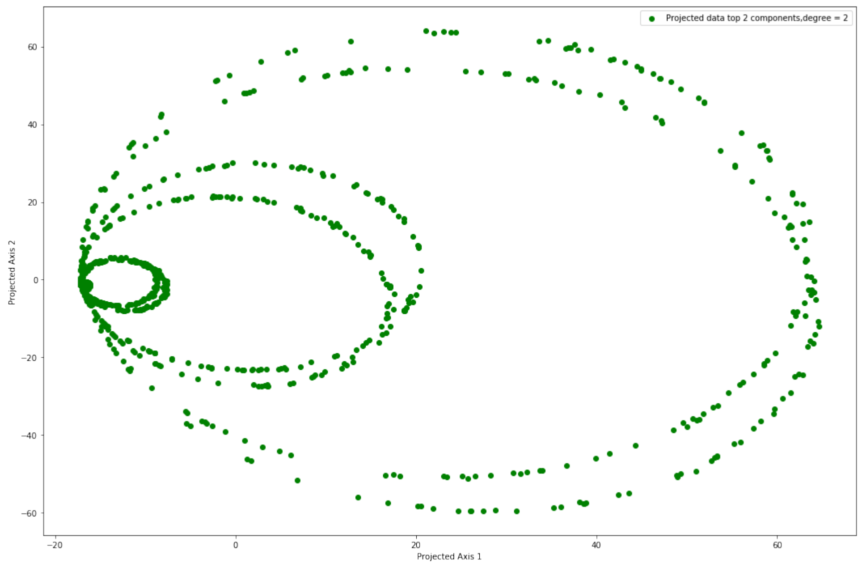

Figure 16:

```
# initialse lambda
lam = 10
# normalise X
X = X/(np.linalg.norm(X,axis = 0))
for i in range(num_iter):
    for y_cord in range(X.shape[1]):
        coeff = X[:,y_cord].reshape(1,−1) @ (y − X@beta_curr+beta_curr↘
        →[y_cord]*X[:,y_cord].reshape(−1,1))
        beta_curr[y_cord] = evaluate_cases(lam,coeff)
```

Next we implement the same cross validation code as above the difference being the instead of Ridge we use LASSO:

```
num_iter = 1000
# store the val error
val_error = []
# normalise X_train values for numerical stability
X_train = X_train/(np.linalg.norm(X_train,axis = 0))
# let us determine the validation error for various lambda values
for lam in np.linspace(0,5,20):
    beta_curr = np.zeros((X_train.shape[1],1))
    # do training on the training set
    for i in range(num_iter):
        for y_cord in range(X_train.shape[1]):
            coeff = X_train[:,y_cord].reshape(1,−1) @ (y_train − ↘
            →X_train@beta_curr+beta_curr[y_cord]*X_train[:,y_cord].↘
```

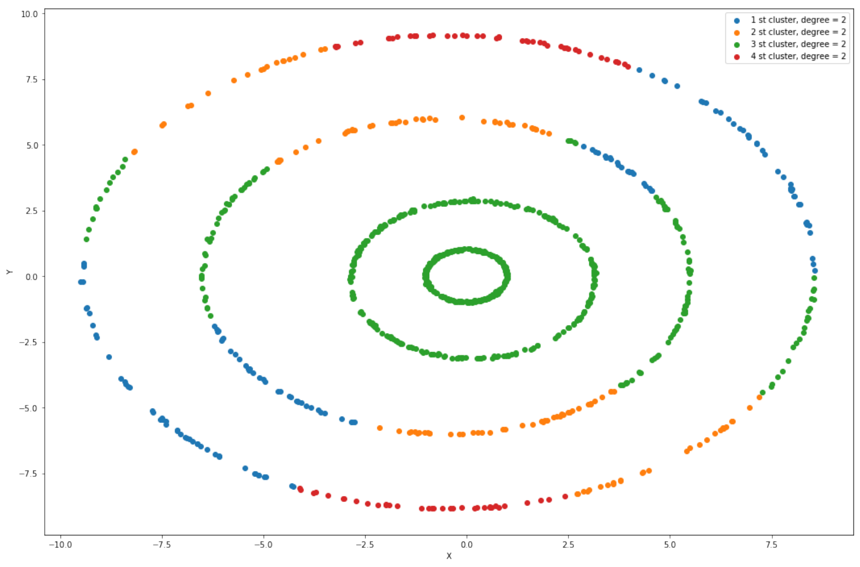

Figure 17:

```
                →reshape(-1,1))
                beta_curr[y_cord] = evaluate_cases(lam, coeff)
        # use validation set to evaluate model performance
        val_error.append(np.linalg.norm(X_val@beta_curr - y_val)**2)
plt.figure(figsize=(18,12))
plt.scatter(np.linspace(0,5,20),val_error,c="g",label="LASSO ↘
→regression")
plt.X_trainlabel("lambda")
plt.ylabel("Validation error")
plt.legend()
```

The validation error graph is given in Fig 26.

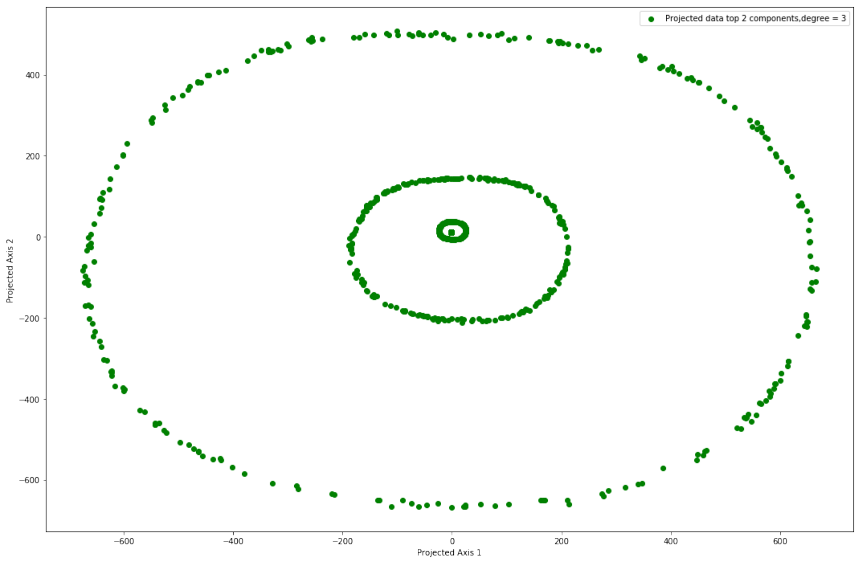

Figure 18:

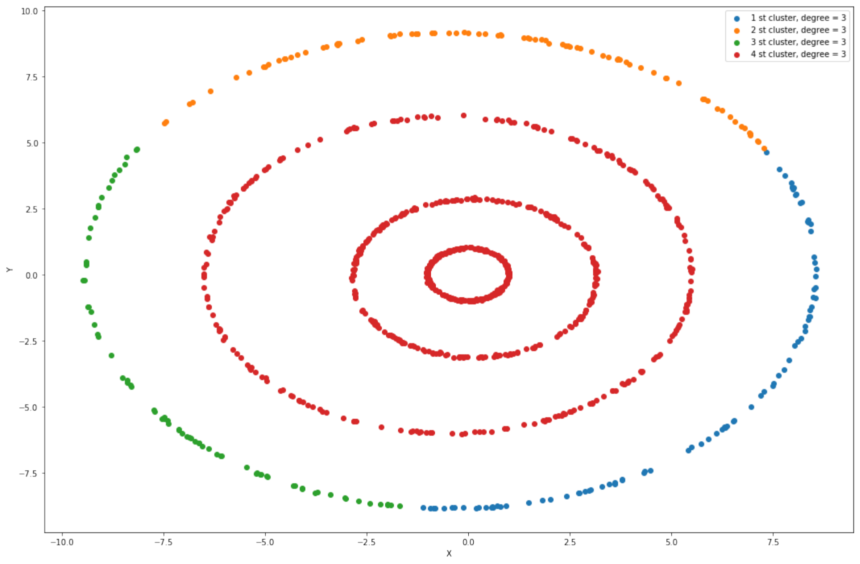

Figure 19:

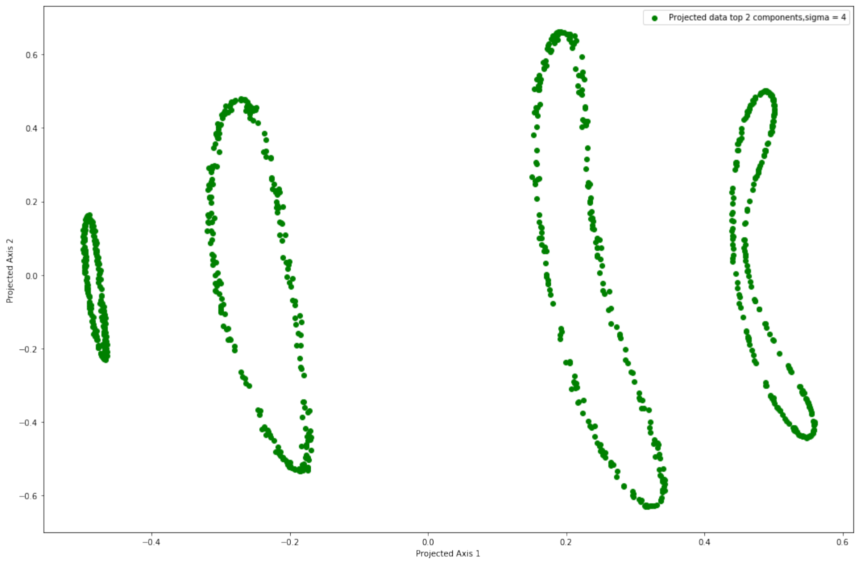

Figure 20:

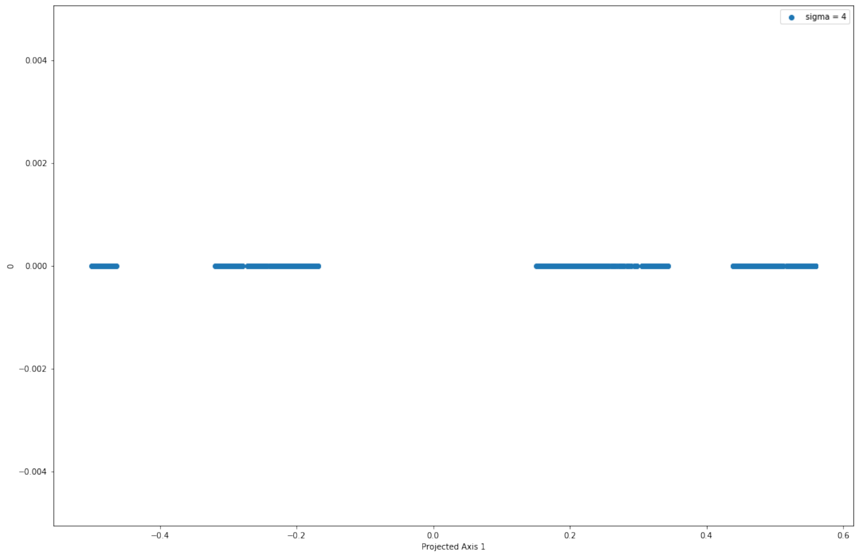

Figure 21:

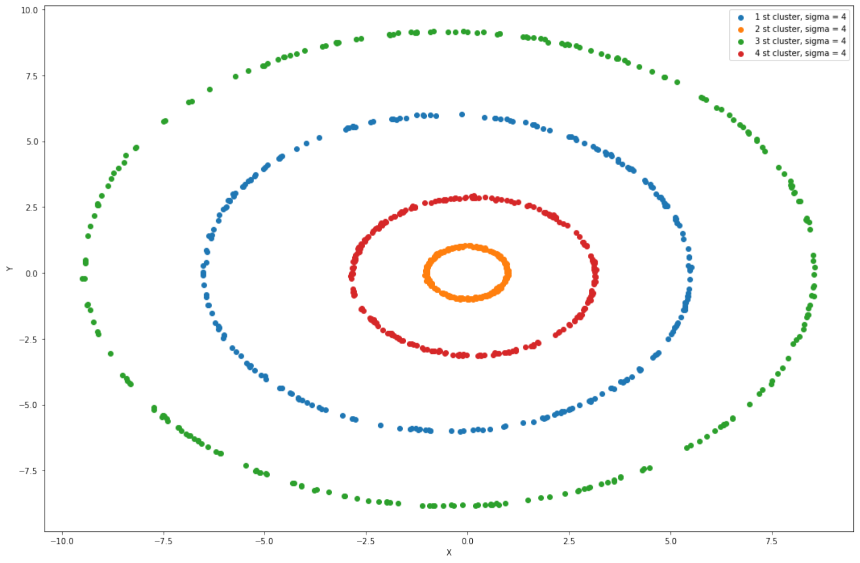

Figure 22:

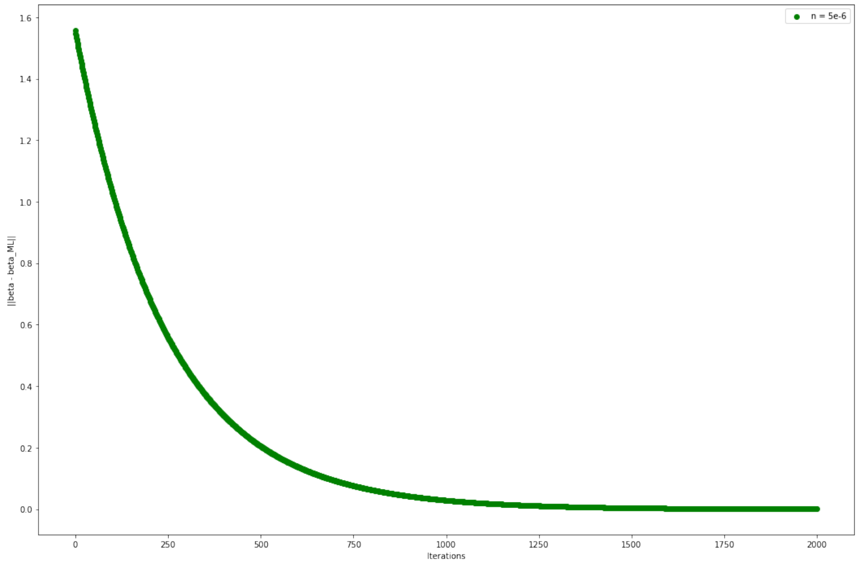

Figure 23:

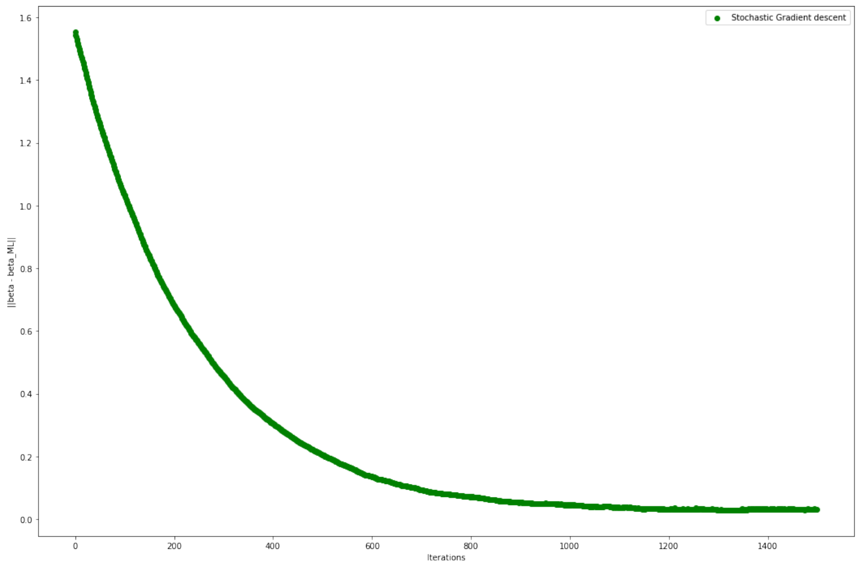

Figure 24:

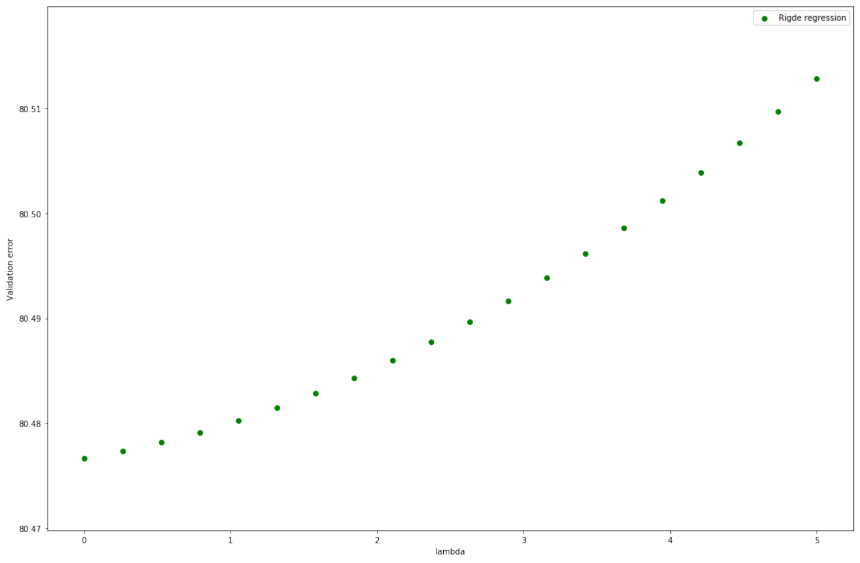

Figure 25:

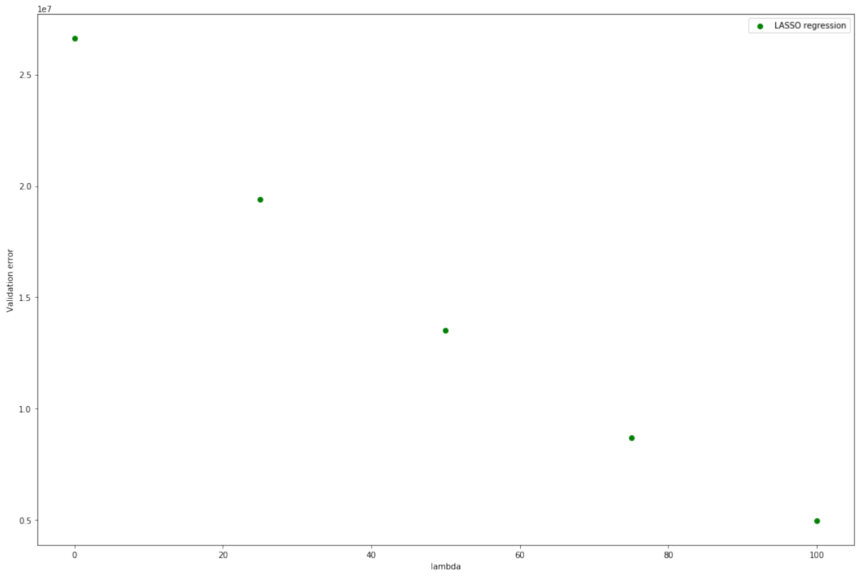
1e7
LASSO regression
Validation error
lambda
2.5
2.0
1.5
1.0
0.5
0
20
40
60
80
100

Figure 26: