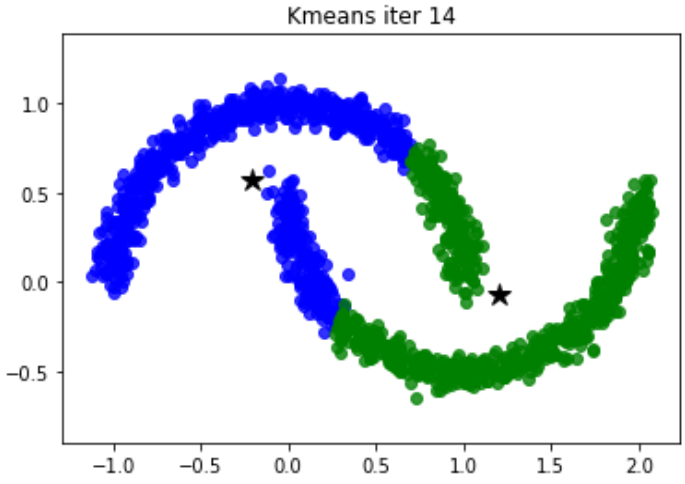
Machine Learning Homework 6

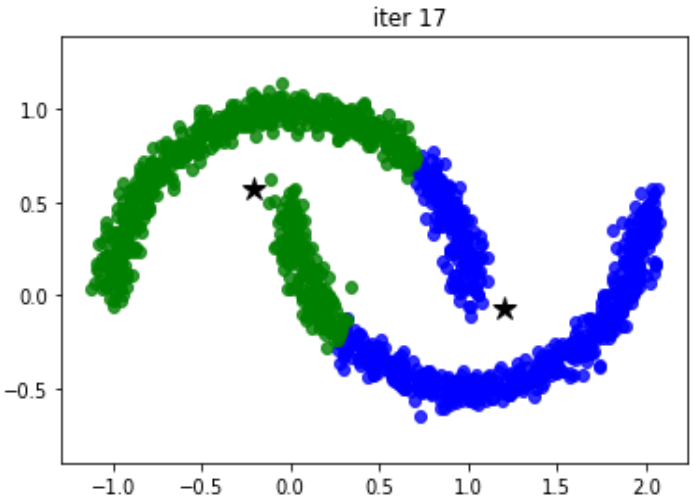
k-means clustering, kernel k-means, spectral clustering, DBSCAN

1. k-means

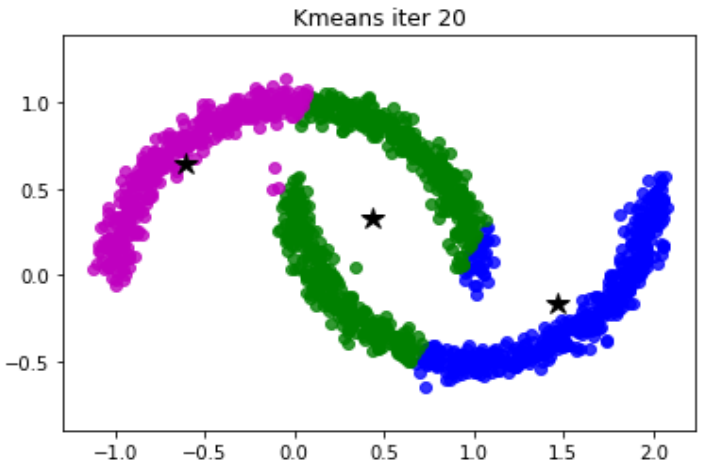
- ✓ k-means clustering on moon dataset k=2 with k-means++ initialization



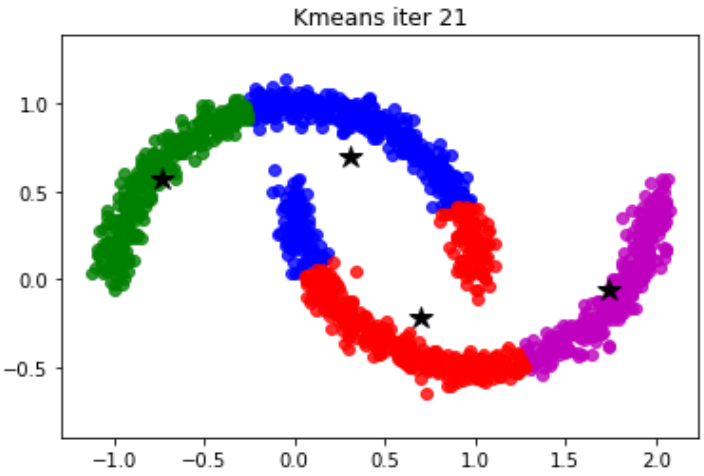
- ✓ k-means clustering on moon dataset k=2 with random initialization



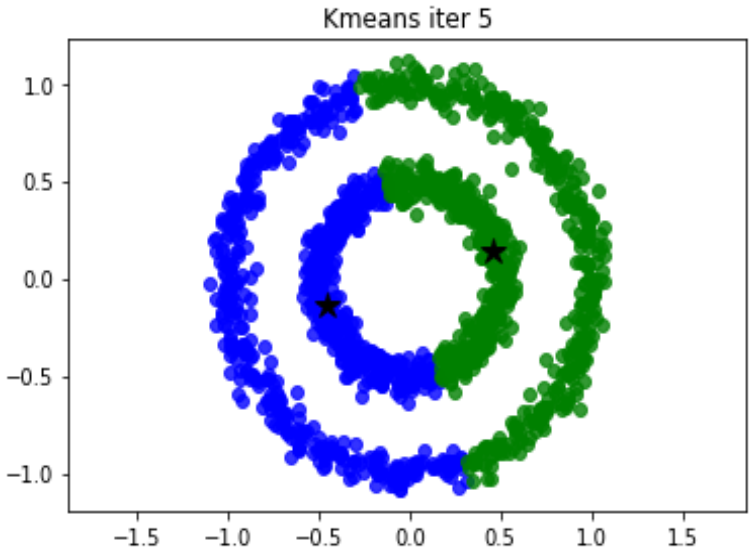
✓ k-means clustering on moon dataset k=3 with k-means++ initialization



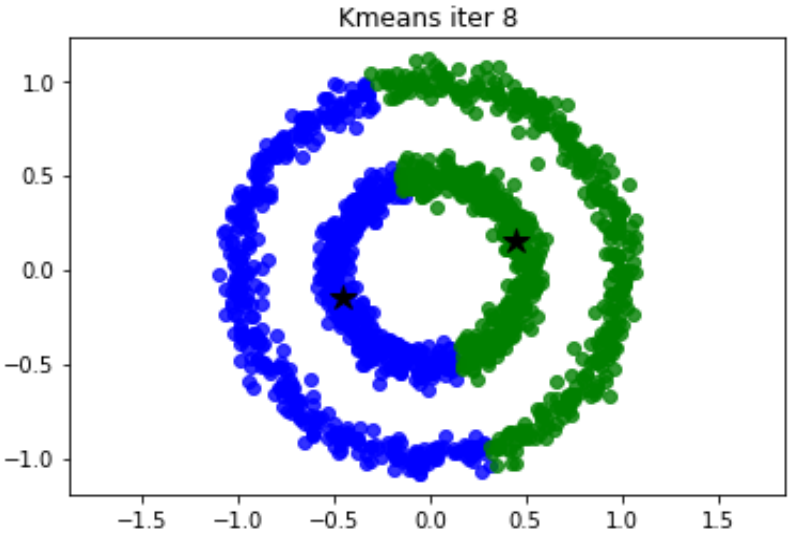
✓ k-means clustering on moon dataset k=4 with k-means++ initialization



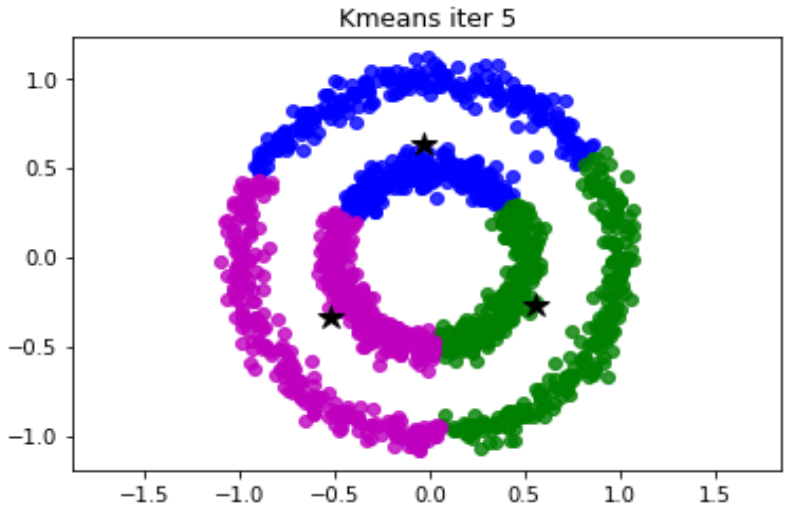
✓ k-means clustering on circle dataset k=2 with k-means++ initialization



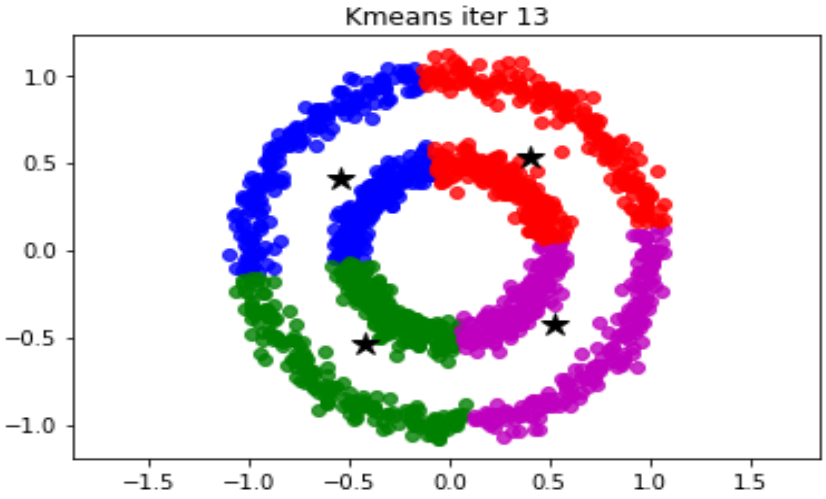
✓ k-means clustering on circle dataset k=2 with random initialization



✓ k-means clustering on circle dataset k=3 with k-means++ initialization



✓ k-means clustering on circle dataset k=4 with k-means++ initialization



- Different ways of initialization used for k-means clustering
    - ✓ K-means++
    - ✓ Random initialization of centroids
    - ✓ Random clustering of data and then centroid initialization from those clusters

- K-means Clustering results discussion

    - ✓ The plots show that k-means clustering algorithm failed to correctly cluster the moon and circle data. The reason is that k- the dataset is nonconvex and kmeans cannot solve it.
    - ✓ The Euclidean distance is used for cluster assignment and centroids are initialized using k-means++, random sampling and centroids calculated from random clusters.
    - ✓ In all the tree cases of initialization, k-means failed to find global minima.
    - ✓ Kmeans++ initialization helps kmeans to converge quickly but does not really improves the clustering results on given datasets.

- Kmeans clustering code explanations

    - ✓ Input the dataset name and number of kClusters and call kmeans function.

```
In [50]:
1 fileName = input("Enter file name:")
2 dataset_name= fileName[:4]
3 kCluster = int(input('Enter the K number of clusters :'))
4 data = pd.read_csv(fileName,header=None)
5 data = np.array(data)
6 num_data = data.shape[0]
7 kmeans(data, kCluster,num_data, 2)

Enter file name:circle.txt
Enter the K number of clusters :4
```

    - ✓ This centroid_inti_kmean_plusplus defines a list: clusters which saves k number of cluster assignments. Firstly, a variable mu (centroid) is initialized with random point from the data. Then, the successive centers are picked, stopping when we *have* length

of mu equal to the number of clusters i.e. kClusters. The next most suitable center is drawn from distribution given by the normalized distance vector (probs; see code). To implement such a probability distribution, the cumulative probabilities are computed for choosing each of the N points in data. These cumulative probabilities are partitions in the interval [0,1] with length equal to the probability of the corresponding point being chosen as a center. Therefore, by picking a random value r in [0,1] and finding the point corresponding to the segment of the partition where that r value falls, we are effectively choosing a point drawn according to the desired probability distribution. Both the clusters and mu are returned at end of function.

In [38]:
```
def centroid_init_kmeans_plusplus(data, kCluster,num_data):

    mu = data[random.sample(range(0,num_data-1),1)]
    mu= mu.tolist()
    while len(mu)<kCluster:
        dist_center = np.array([min([np.linalg.norm(x-c)**2 for c in mu]) for x in data])
        probs = dist_center/dist_center.sum()
        cumprobs = probs.cumsum()
        r = random.random()
        ind = np.where(cumprobs >= r)[0][0]
        mu.append(data[ind].tolist())

    clusters = [[[None] for i in range(0)] for j in range(kCluster)]
    # randomly assign data to clusters
    for i in range(num_data):
        k= random.randint(0,kCluster-1)
        clusters[k].append(data[i])

    return np.array(mu), clusters
```

- ✓ This centroid_inti_random defines a list: clusters which saves k number of cluster assignments. A variable mu (centroid) is initialized randomly. Both the clusters and mu are returned.

In [33]:

```
def centroid_init_random(data,kCluster,num_data):

    clusters = [[[None] for i in range(0)] for j in range(kCluster)]

    #initail the cluster centroids with random samples from data
    mu = data[random.sample(range(0,num_data-1), kCluster)]

    # randomly assign data to clusters
    for i in range(num_data):
        k= random.randint(0,kCluster-1)
        clusters[k].append(data[i])

    return mu, clusters
```

- ✓ The centroid_inti_random_cluster_mean defines a list: clusters which saves k number of cluster assignments. A variable mu (centroid) is initialized by calculating centroids from randomly assigned clusters. Both the clusters and mu are returned.

In [34]:

```
def centroid_init_random_cluster_mean(data,kCluster,num_data):

    #initail the cluster centroids with random samples from data
    mu = data[random.sample(range(0,num_data-1), kCluster)]

    clusters = [[[None] for i in range(0)] for j in range(kCluster)]

    # randomly assign data to clusters and calculte another mu
    for i in range(num_data):
        k= random.randint(0,kCluster-1)
        clusters[k].append(data[i])
    #calculate mu from randomly assigned clusters
    for k in range(kCluster):
        mu[k] = np.mean(clusters[k],axis=0)

    return mu, clusters
```

- ✓ Compute cluster assignments using Euclidean distance. Check the distance of one point with each centroids and assign the data point to that cluster whose centroid is closed to that data point.

```
        for i in range (num_data) :
            min_dist = 1000000
            nearestK = - 1
            for k in range (kCluster) :
                    temp_dist = np.linalg.norm( data[i] - mu[k])
                    if (temp_dist < min_dist) :
                        min_dist = temp_dist
                        nearestK = k

            clusters[nearestK].append (data[i])

```

- ✓ Calculate newMu each iteration from new clusters and check difference between newMu and Mu to check convergence.

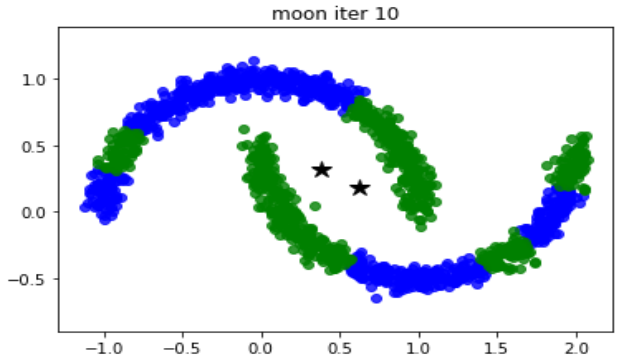
```
        for k in range (kCluster) :
            newMu[k] = np.mean(clusters[k],axis =0)
        if ( sqrt(np.linalg.norm(newMu-mu)) < 0.00001):
            iteration = it
            break
```

- ✓ Plot the clusters and centroids for each iteration

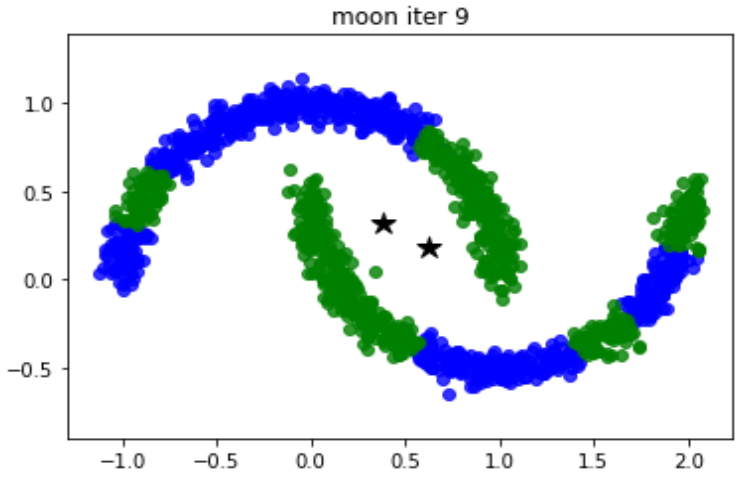
```
    for k in range (kCluster):
        plt.plot(np.array(clusters[k])[:,0],np.array(clusters[k])[:,1],color[k]+"o", alpha = 0.8)
        plt.plot(mu[k,0],mu[k,1],"k*", markersize=12, alpha = 1)

```

2. Kernel K-means

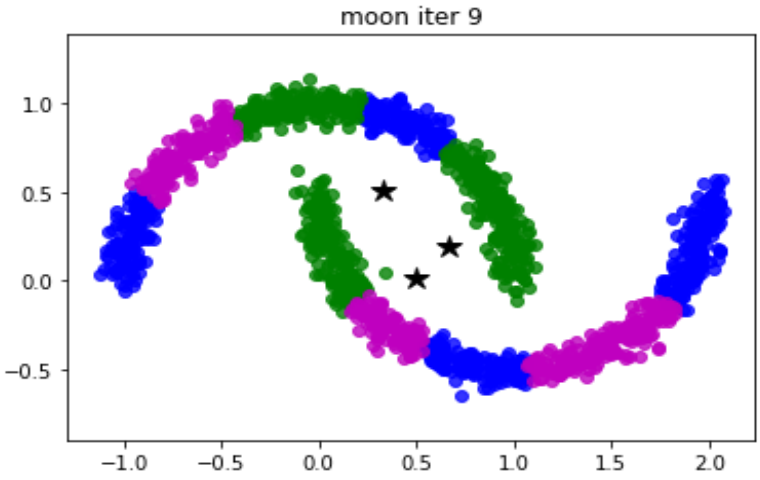
- ✓ Kernel k-means clustering on moon dataset k=2 with random initialization of data



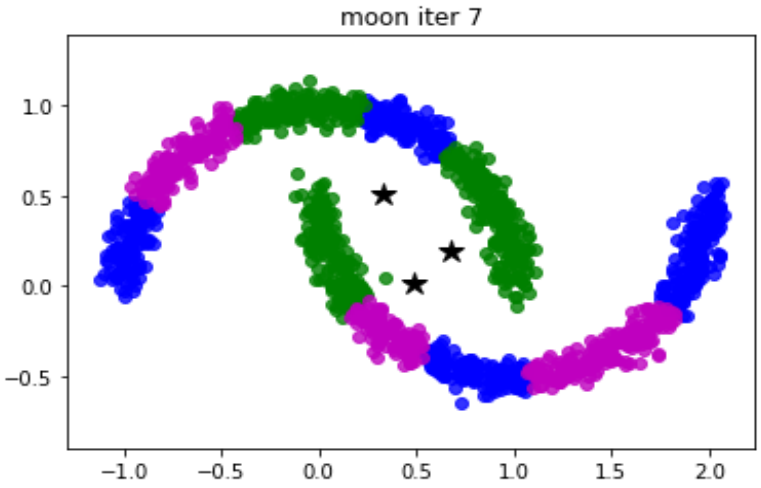
✓ Kernel k-means clustering on moon dataset k=2 with mode initialization of data



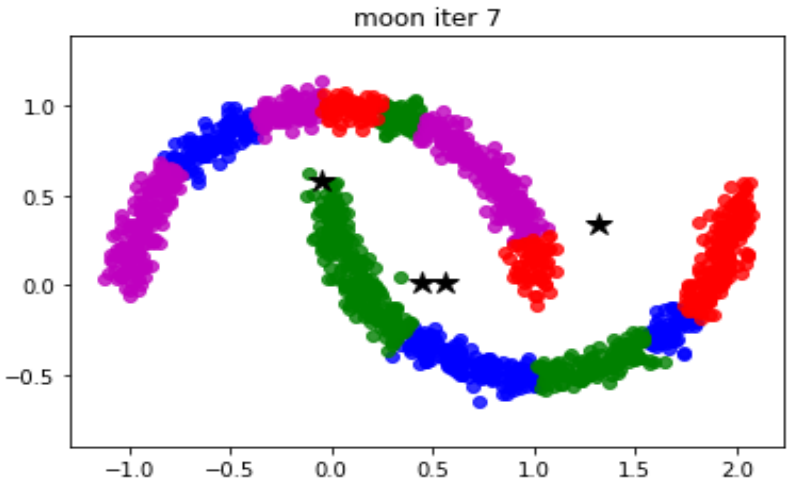
✓ Kernel k-means clustering on moon dataset k=3 with random initialization of data



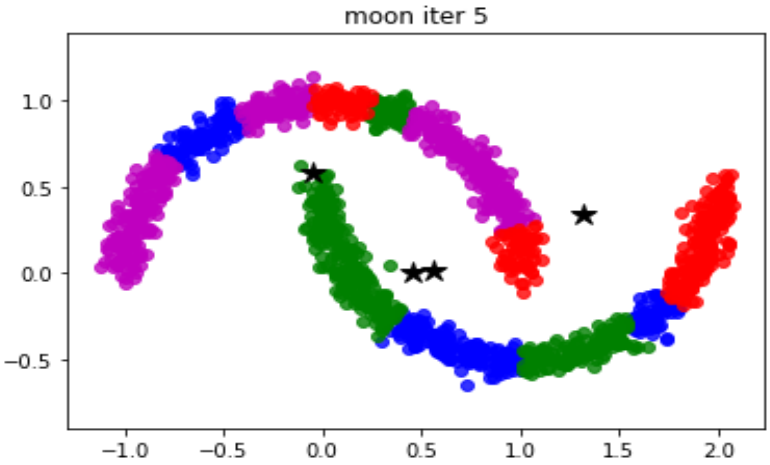
✓ Kernel k-means clustering on moon dataset k=3 with mode initialization of data



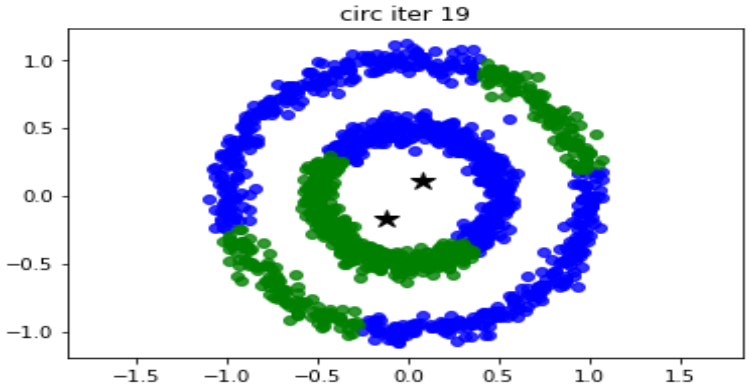
✓ Kernel k-means clustering on moon dataset k=4 with random initialization of data



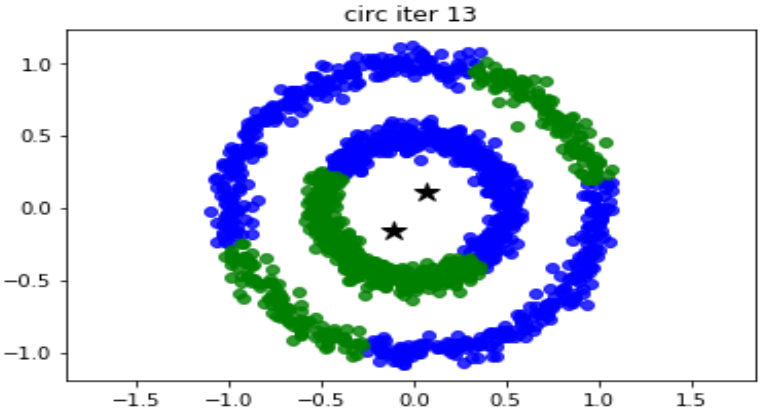
✓ Kernel k-means clustering on moon dataset k=4 with mode initialization of data



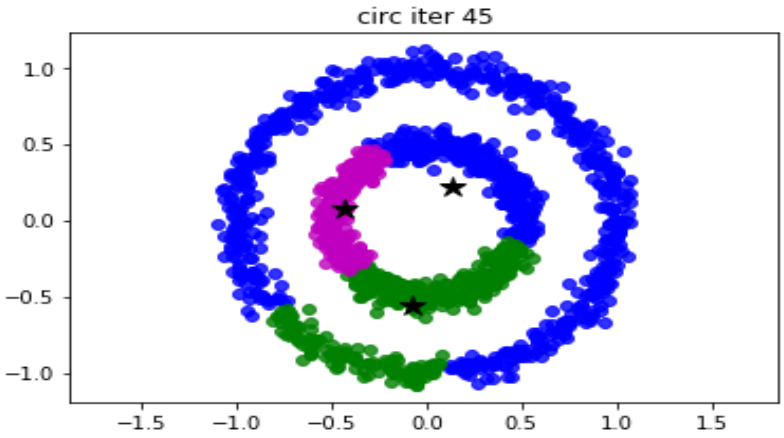
✓ Kernel k-means clustering on circle dataset k=2 with random initialization of data



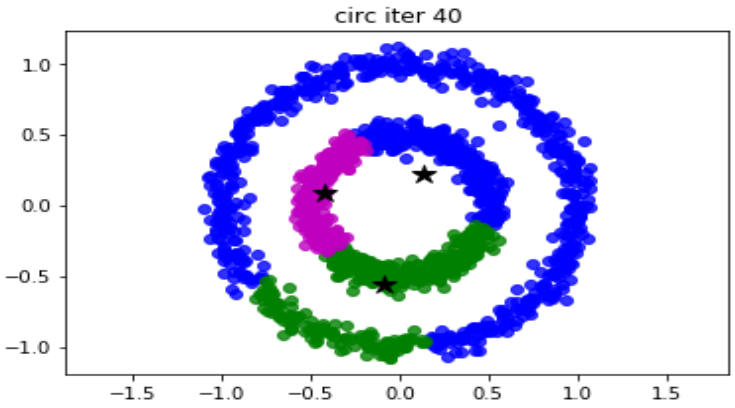
✓ Kernel k-means clustering on circle dataset k=2 with mode initialization of data



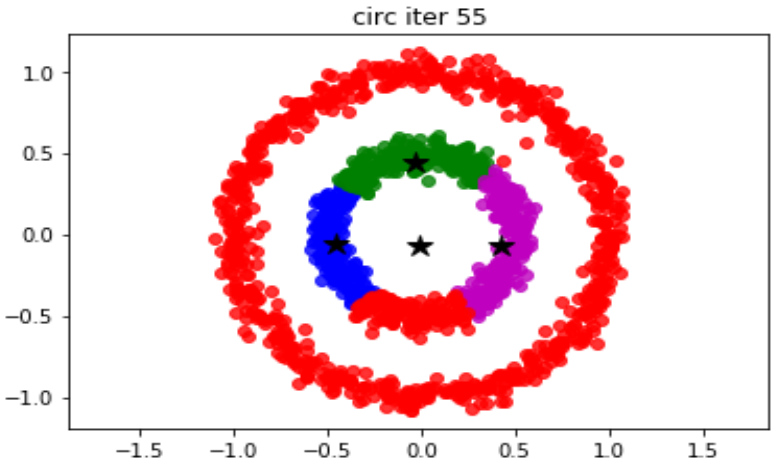
✓ Kernel k-means clustering on circle dataset k=3 with random initialization of data



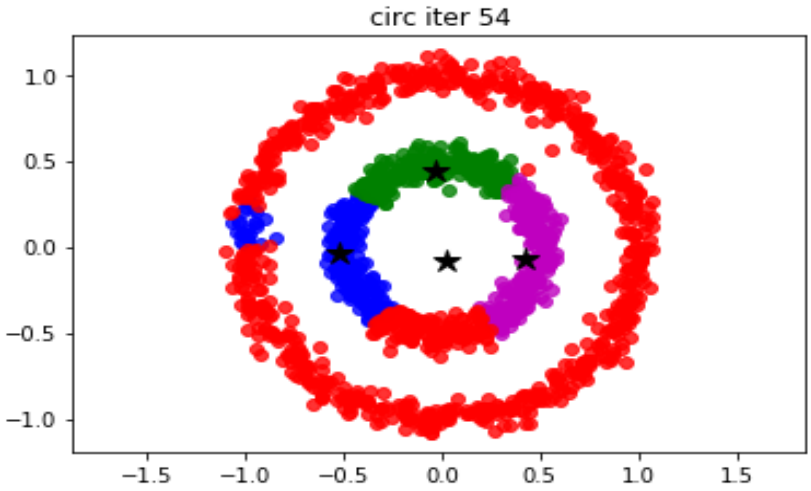
✓ Kernel k-means clustering on circle dataset k=3 with mode initialization of data



✓ Kernel k-means clustering on circle dataset k=4 with random initialization of data



- ✓ Kernel k-means clustering on circle dataset k=4 with mode initialization of data



- Different ways of centroid initialization used for k-means clustering

  - ✓ Random initialization of clusters data
  - ✓ Randomly pick a cluster number and a point to this cluster in loop
  - ✓ Take the mode of index with K (i/k) and assign data point to this cluster

- Kernel K-means Clustering results discussion

  - ✓ The plots show that kernel k-means clustering algorithm is failed to correctly cluster the moon and circle data. The reason is that the dataset is nonconvex and kernel kmeans cannot solve it.
  - ✓ The RBF kernel is used for mapping the data into high dimension.
  - ✓ In all the three cases of initialization, kernel k-means is failed to find global minima.

- Kernel Kmeans clustering code explanations

  - ✓ Input the dataset name, number of kClusters and read data and call Kernel kmeans function.

```
fileName = input("Enter file name:")
dataset_name= fileName[:4]
kCluster = int(input('Enter the K number of clusters :'))
data = pd.read_csv(fileName,header=None)
data = np.array(data)
num_data = data.shape[0]
np.random.shuffle(data)
_=kernel_kmeans(data, num_data, kCluster, init_flag=3)
```

- ✓ Kernel Kmeans functions take init_flag to select initialization method

```
def kernel_kmeans(X,num_data,K,init_flag=0):

    if init_flag==0:
        init_data =initByBatch(X,num_data,K)
    elif init_flag==1:
        init_data = initByMod(X,num_data,K)
    elif init_flag==2:
        init_data =initRandom(X,num_data,K)
    elif init_flag==3:
        init_data = randomClustering(X,num_data,K)
```

- ✓ Three ways are used for initializing cluster data which are explained above.

```
def initByMod(X,num_data,K):
    init_clust = [[[None] for i in range(0)] for j in range(K)]
    for i in range (num_data):
        init_clust[i%K].append(X[i])
    return init_clust

def initRandom(X,num_data,K):
    init_clust = [[[None] for i in range(0)] for j in range(K)]
    for i in range (num_data):
        init_clust[random.choice(range(K))].append(X[i])
    return init_clust

def randomClustering (X,num_data,K):
    init_clust = [[[None] for i in range(0)] for j in range(K)]
    #random assign
    for i in range (num_data):
        k = random.randint(0,K-1)
        init_clust[k].append(X[i])
    return init_clust
```

- ✓ The following formula is implemented for kernel kmeans lecture slides. There are three terms in final formula. The first term is ignored because is a constant and second and third terms calculated to perform the cluster assignment.

Kernel K-means

$$\underset{(C_1,\mu_1),\ldots,(C_k,\mu_k)}{\arg\min} \sum_{i=1}^{k} \sum_{x_j \in C_i} \|x_j - \mu_i\|^2$$

- write each center in kernel space: where if the data point x_n is assigned to the k-th cluster, then $a_{kn} = 1$

$$\mu_k^{\phi} = \frac{1}{|C_k|} \sum_{n=1}^{N} \alpha_{kn} \phi(x_n)$$

- Now

$$\left\|\phi(x_j) - \mu_k^{\phi}\right\| = \left\|\phi(x_j) - \frac{1}{|C_k|} \sum_{n=1}^{N} \alpha_{kn} \phi(x_n)\right\|$$
$$= \mathbf{k}(x_j, x_j) - \frac{2}{|C_k|} \sum_{n} \alpha_{kn} \mathbf{k}(x_j, x_n) + \frac{1}{|C_k|^2} \sum_{p} \sum_{q} \alpha_{kp} \alpha_{kq} \mathbf{k}(x_p, x_q)$$

- ✓ Computer the distance of each point with all centroids based on the RBF kernel and do cluster assignment.
- ✓ Firstly, calculate the third term and get newMu based on this term and also check convergence for calculating the distance between newMu and mean.
- ✓ Secondly, calculate temp_dist which the distance between centroid (newMu) and data point and do cluster assignment.
- ✓ Assign the point to that cluster whose centroid is closer to that point.

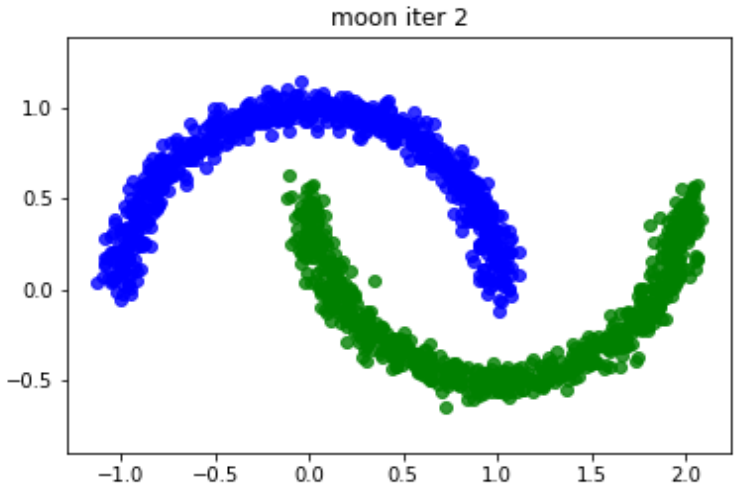
```
        # calculate mean
        newMu = np.zeros(K)
        #calculate mean
        for k in range (K):
            sizeK = len(init_data[k])
            temp_sum =0
            for i in range (sizeK):
                for j in range (sizeK):
                    temp_sum += RBF_kernel(gamma,init_data[k][i],init_data[k][j])
            newMu[k] = temp_sum/(sizeK**2)

        if ( sqrt(np.linalg.norm(newMu-mean)) < 0.00001):
            iteration = it
            break
        mean = newMu.copy()

        clusterData = [[[None] for i in range(0)] for j in range(K)]
        print("computing cluster ...")
        for i in range (num_data) :
            min_dist = 10000000
            nearestK = - 1
            for k in range (K) :
                sizeK = len(init_data[k])
                temp_dist = mean[k]
                for j in range (sizeK) :
                    temp_dist -= 2*(RBF_kernel(gamma,init_data[k][j],X[i]))/sizeK
                if (temp_dist < min_dist) :
                    min_dist = temp_dist
                    nearestK = k
            #print (minD)
            clusterData[nearestK].append (X[i])
```

- ✓ Plot the clusters and centroids for each iteration

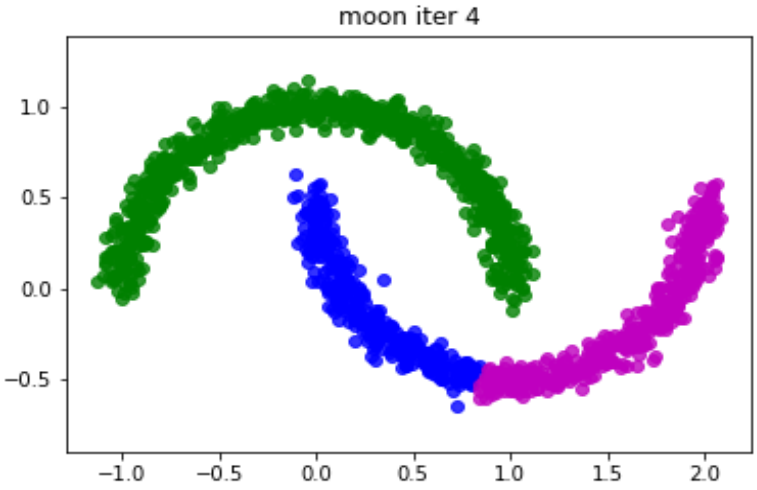
```

    for k in range (K):
        _=plt.plot(np.array(init_data[k])[:,0],np.array(init_data[k])[:,1],color[k]+"o", alpha = 0.8)
        _=plt.plot(miu[k,0],miu[k,1],"k*", markersize=12, alpha = 1)

```

3. Spectral Clustering

- ✓ Spectral clustering on moon dataset k=2 using RBF similarity matrix W to initialize Laplacian graph matrix L.
- ✓ Gamma = 0.25 for RBF similarity matrix

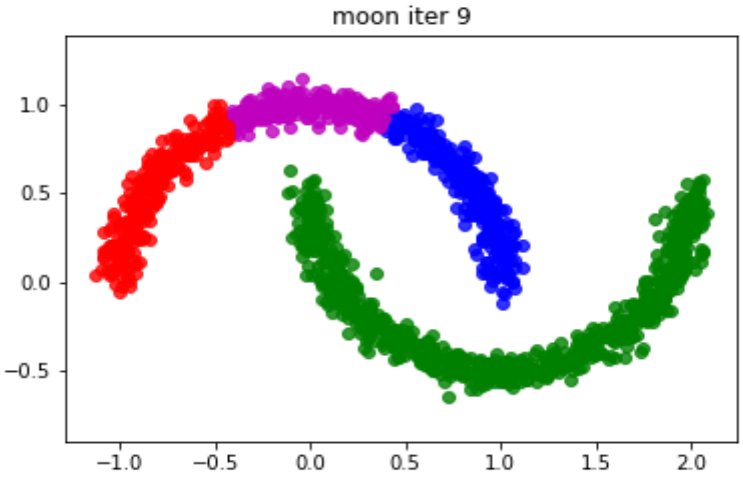


- ✓ Spectral clustering on moon dataset k=3 using RBF similarity matrix W to initialize Laplacian graph matrix L.
- ✓ Gamma = 0.25 for RBF similarity matrix

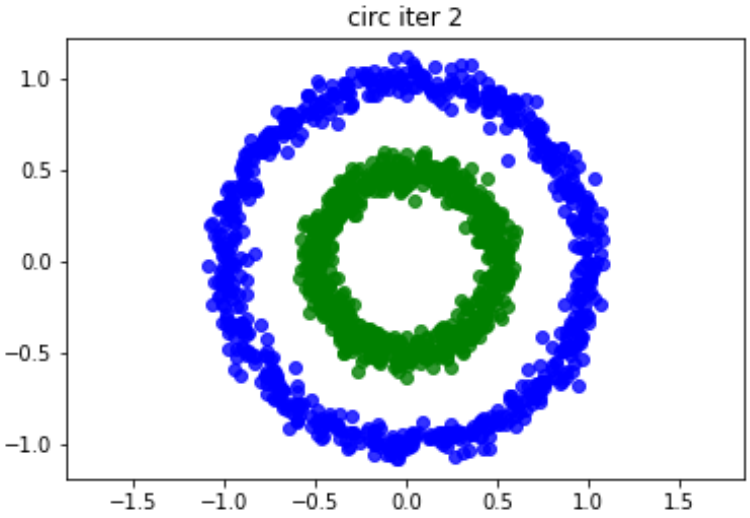


- ✓ Spectral clustering on moon dataset k=4 using RBF similarity matrix W to initialize Laplacian graph matrix L.

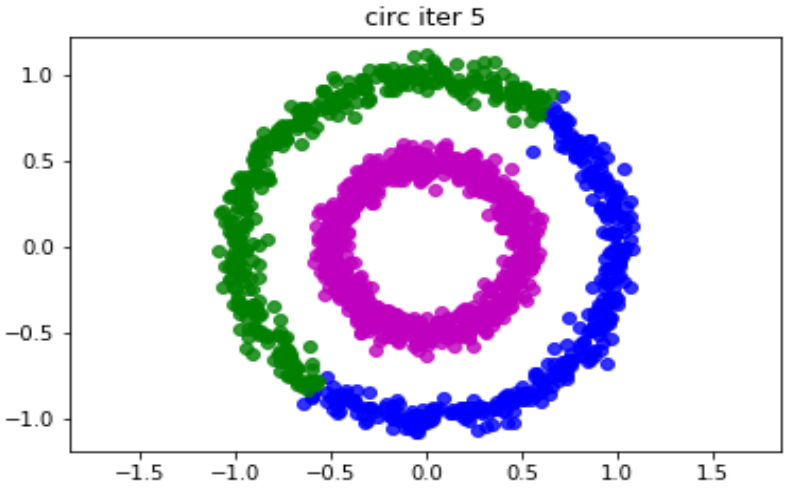
✓ Gamma = 0.25 for RBF similarity matrix



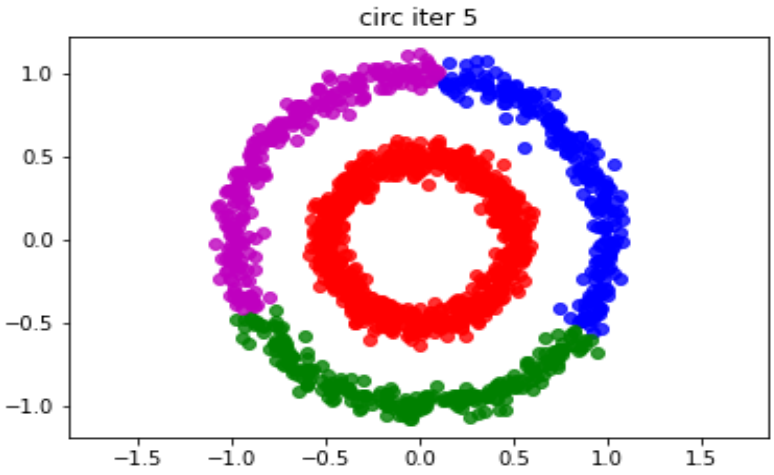
✓ Spectral clustering on circle dataset k=2 using RBF similarity matrix W to initialize Laplacian graph matrix L.
✓ Gamma = 0.1 for RBF similarity matrix



- ✓ Spectral clustering on circle dataset k=3 using RBF similarity matrix W to initialize Laplacian graph matrix L.
- ✓ Gamma = 0.1 for RBF similarity matrix



- ✓ Spectral clustering on circle dataset k=4 using RBF similarity matrix W on fully connected graph to initialize Laplacian graph matrix L.
- ✓ Gamma = 0.1 for RBF similarity matrix

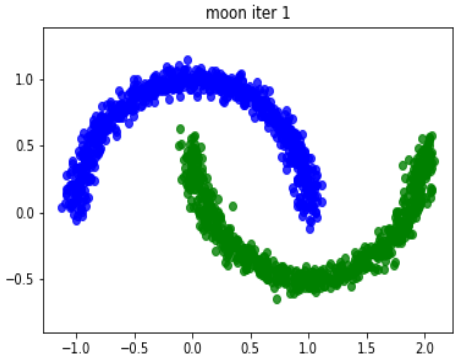


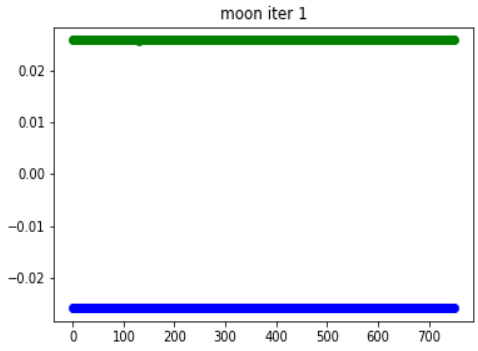
- Different ways of centroid initialization used for k-means clustering

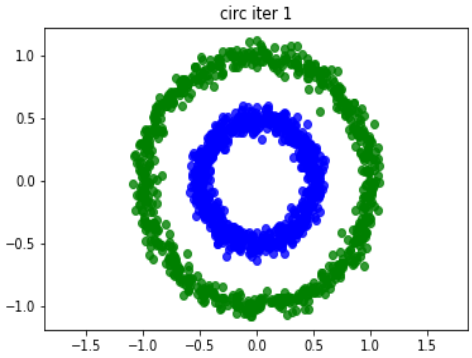
    - ✓ fully connected graph with RBF similarity matrix on
    - ✓ K-nearest neighbor graph
- Spectral Clustering results discussion

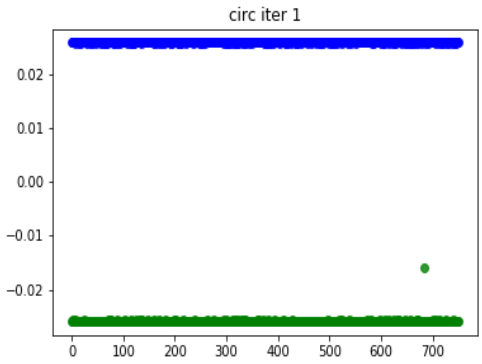
    - ✓ The plots show that spectral clustering algorithm successfully separates the two clusters in the moon and circle data.
    - ✓ In both cases of initialization, spectral clustering performed similar and successfully solved the problem.

- Discussion on eigenvectors visualization in eigenspace of graph Laplacian
    - ✓ From below figures, we can see the data points within the same cluster do have the same coordinates in the eigenspace of graph Laplacian.
    - ✓ The figures show the good clustering results for k=2.

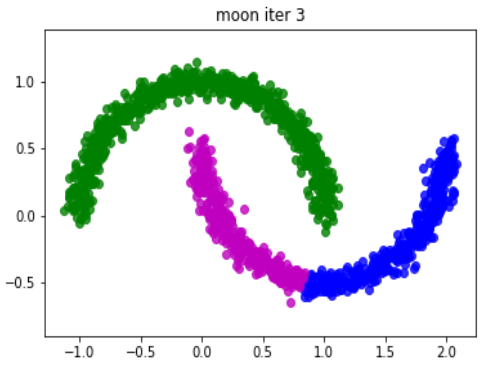


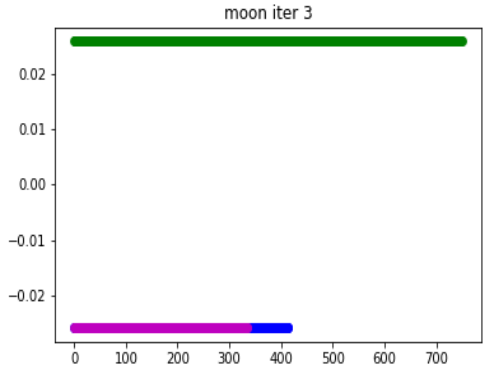


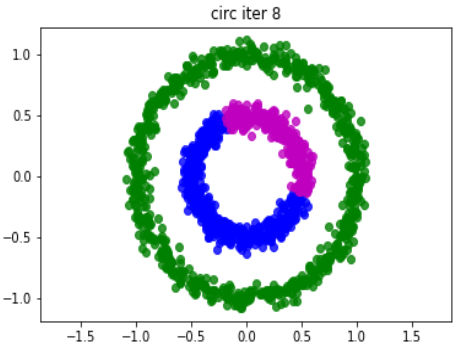


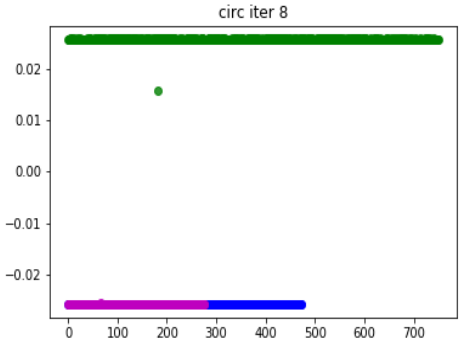


- ✓ From below figures, we can see the data points within the same cluster do have the same coordinates in the eigenspace of graph Laplacian. As we want to find 3 clusters but eigenvectors of cluster 2 and 3 lies on same coordinate in eigenspace. Therefore, it is a bad clustering solutions for k=3. Because moon and circle dataset does not contain 3 clusters in reality.









- Spectral Clustering code explanations

  - ✓ Input the dataset name, number of kClusters and read data and call Spectral Clustering function.

In [14]:
```
fileName = input("Enter file name:")
dataset_name= fileName[:4]
kCluster = int(input('Enter the K number of clusters :'))
data = pd.read_csv(fileName,header=None)
data = np.array(data)
num_data = data.shape[0]
strName= dataset_name
np.random.shuffle(data)
_=spectral_clustering(data,kCluster,num_data)
```

```
Enter file name:moon.txt
Enter the K number of clusters :4
Computing Graph Laplacian
```

- ✓ RBF kernel function to computer similarity matrix

```
def RBF_kernel(gamma,A,B) :
    diff = np.linalg.norm(A-B)
    d = np.exp(-(diff)/(2*gamma**2))
    return d
```

- ✓ Compute similarity matrix W and degree matrix D
- ✓ Initialize graph Laplacian L using similarity matric W and degree matrix D

```
def laplacian(num_data,X):
    gamma = 0.1
    print ("Computing Graph Laplacian")
    W = np.zeros((num_data,num_data))
    for i in range(0,num_data):
        for j in range(i+1, num_data,1):
            W[i,j] = RBF_kernel(gamma,X[i], X[j])
            W[j,i] = W[i,j]

    D = np.zeros((num_data,num_data))
    for i in range(num_data):
        D[i,i] = sum(W[i])

    #ratio cut
    L = D -  W

    return L
```

- ✓ Compute eigenvectors and eigenvalues from L and get smallest vectors by sorting
- ✓ Transform data to spectral space
- ✓ Randomly compute centroids in spectral space

```
#calculating eigenValues, eigenVectors
eigenValues, eigenVectors = np.linalg.eig(L)

#the first K value
smallestEigenVector =  np.argsort(eigenValues.real)
SpecSpace =np.zeros((num_data,0))

for i in range(0,K,1):
        SpecCoor = (np.array(eigenVectors[:,smallestEigenVector[i]]))
        SpecCoor = np.reshape(SpecCoor,(num_data,1))
        SpecSpace = np.concatenate((SpecSpace, SpecCoor), axis=1)


#random k means
#Randomly select K points in the new data space as the centroids
mu =  SpecSpace [random.sample(range(0, num_data), K)]

```

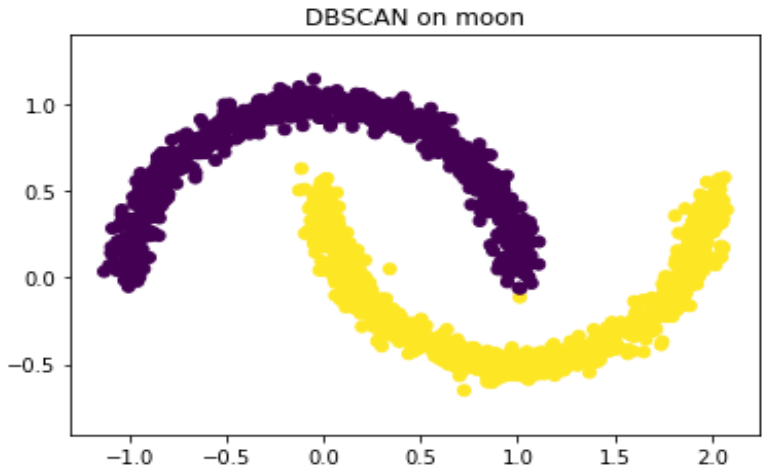
- ✓ Calculate cluster assignments in spectral space for data
- ✓ Calculate the Euclidean distance of one point in spectral space with each centroid and assign the point to cluster whose centroid is closer to that point.
- ✓ Get the cluster for original data and spectral space to verify data points within the same
- ✓ cluster do have the same coordinates in the eigenspace of graph Laplacian

```
#reset for the next iteration
clusterOriginalData = [[[None] for i in range(0)] for j in range(K)]
clusterSpecSpaceData = [[[None] for i in range(0)] for j in range(K)]

for i in range (num_data) :
    min_dist = 10000000
    nearestK = - 1
    for k in range (K) :
            temp_dist = sqrt(np.linalg.norm(SpecSpace[i]-mu[k]))
            if (temp_dist < min_dist) :
                min_dist = temp_dist
                nearestK = k

    clusterOriginalData[nearestK].append (data[i])
    clusterSpecSpaceData[nearestK].append (SpecSpace[i])
```

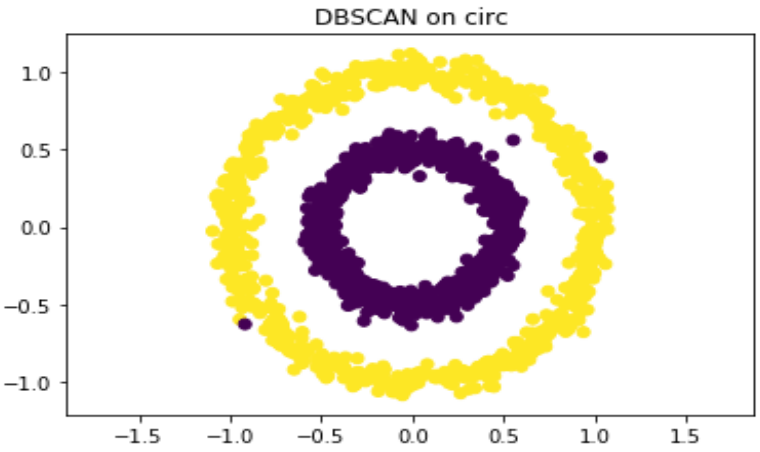
- ✓ plot cluster data in original space and spectral space and check convergence

```
for k in range (K):
    plt.plot(np.array(clusterOriginalData[k])[:,0],np.array(clusterOriginalData[k])[:,1],color[k]+"o", alpha = 0.8)


plt.title(dataset_name +" iter " +str(it))
plt.axis('equal')
plt.savefig(path+"/SpectCluster_"+ dataset_name +"_"+ str(it)+".png", bbodata_inches="tight")

for k in range (K): plt.plot(np.array(clusterSpecSpaceData[k])[:,1],color[k]+"o", alpha = 0.8)
plt.title(dataset_name +" iter " +str(it))
plt.savefig(path+"/SpectCluster_SpecSpace"+ dataset_name +"_"+ str(it)+".png", bbodata_inches="tight")
plt.clf()

new_mu = np.zeros((K,K))

for k in range (K) :
    new_mu[k] = np.mean(clusterSpecSpaceData[k],axis =0)
if ( sqrt(np.linalg.norm(new_mu-mu)) < 0.00001):
    iteration = it
    break
mu = new_mu.copy()
```

4. DBSCAN
   - ✓ DBSCAN on moon dataset with eps=0.3 and MinPts = 50



   - ✓ DBSCAN on circle dataset with eps=0.1 and MinPts = 5



- ✓ DBSCAN results discussion
  - ✓ DBSCAN successfully solves the clustering problem on both moon and circle dataset.
  - ✓ DBSCAN is sensitive to parameter eps and MinPts.
  - ✓ DBSCAN forms good cluster on circle when eps=0.1 and MinPts=5 but it does not form good cluster if the eps=0.3 and MinPts=50 for circle data.

- DBSCAN code explanations

  - ✓ Input the dataset name and read data and call DBSCAN function.

```
1  fileName = input("Enter file name:")
2  dataset_name= fileName[:4]
3  data = pd.read_csv(fileName,header=None)
4  data = np.array(data)
5  dataset_name
6  _=DBSCAN(data,dataset_name, eps=0.3 ,MinPts=50)
```

```
Enter file name:moon.txt
All data points have been processed
```

  - ✓ DBSCAN takes data, dataset_name, eps and MinPts as parameters. Eps is the threshold distance (radius) and MinPts is minimum number of points required in neighborhood.

- ✓ Define a cluster_data list with size equal to the dataset and initialize a variable C which is the ID of the current cluster.
- ✓ This list will hold the final cluster assignment for each point in D. There are two reserved values: 1 - Indicates a noise point, 0 - Means the point hasn't been considered yet.

```
def DBSCAN(data, dataset_name, eps, MinPts):

    clusters_data = [0]*len(data)

    # C is the ID of the current cluster.
    C = 0

    for pid in range(0, len(data)):

        if not (clusters_data[pid] == 0):
            continue

        NeighborPts = find_neighbors(data, pid, eps)

        if len(NeighborPts) < MinPts:
            clusters_data[pid] = -1

        else:
            C += 1
            Clusterize_data(data, clusters_data, pid, NeighborPts, C, eps, MinPts,dataset_name)

    print("All data points have been processed")
```

- ✓ Randomly pick the index of a data point and check if the cluster_data is assigned any point on that index. If the point is assigned, then continue to check next point. If the point is not assigned, then find the neighbors of this pint by calling find neighbors function.

```
def find_neighbors(dtat, P, eps):

    neighbors = []

    # For each point in the dataset...
    for Pn in range(0, len(data)):

        # If the distance is below the threshold, add it to the neighbors list.
        if np.linalg.norm(data[P] - data[Pn]) < eps:
            neighbors.append(Pn)

    return neighbors
```

- ✓ If the number of neighbors of this point are less than the MinPts, then set this point as noise point in cluster_data.
- ✓ If the number of neighbors for this point are more than the MinPts, then grow the cluster by calling the calusterize function.
- ✓ Then check the neighbors of each neighbor recursively and grow cluster.

```
def Clusterize_data(data, labels, Pts_index, NeighborPts, C, eps, MinPts,dataset_name):

    # set path for saving the plots after each iteration

    path ="plots/DBSCAN_on_"+dataset_name+"_dataset_"+str(Pts_index)
    #make directory using above path
    try:
        os.makedirs(path)
    except OSError as e:
        if e.errno != errno.EEXIST:
            raise

    # Assign the cluster label to the seed point.
    labels[Pts_index] = C
    i = 0
    while i < len(NeighborPts):

        Pts_idx = NeighborPts[i]
        if labels[Pts_idx] == -1:
           labels[Pts_idx] = C

        elif labels[Pts_idx] == 0:

            labels[Pts_idx] = C
            PnNeighborPts = find_neighbors(data,Pts_idx, eps)
            if len(PnNeighborPts) >= MinPts:
                NeighborPts = NeighborPts + PnNeighborPts
```

- ✓ Plot the cluster and save figures

```
        #save figure after 1000 iterations
        if i%1000==0:
            it=0
            plt.scatter(data[:,0],data[:,1],c=labels)
            plt.title("DBSCAN on "+dataset_name)
            plt.axis('equal')
            plt.savefig(path+"/DBSCAN_"+ str(i)+".png", bbox_inches="tight")
            plt.clf()
            it +=1

        i += 1
```

- ✓ While breaks when all the neighbors are checked

5. Comparison of k-means clustering, kernel k-means, spectral clustering, DBSCAN

- ✓ K-means and Kernel k-means cannot provide good clustering solution for non-convex datasets like moon and circle
- ✓ Spectral Clustering and DBSCAN provides good solution to find out 2 cluster.