Homework Objective: Use whatever your favorite language to code out k-means clustering, kernel k-means, spectral clustering and DBSCAN. You are allowed to use NumPy and it can help you to solve the eigenvalue problem. Using the matplotlib is legal. **But you can't use scikit-learn and SciPy in this homework.**
(kernel k-means and spectral clustering both based on RBF kernels)

Dataset: 2 datasets with points on 2d space, **circle.txt** and **moon.txt**

Using Library

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.pyplot import cm
import random
import copy
```

Read Data

```
with open('circle.txt') as fc:
    circle = [[float(x) for x in line.split(',')] for line in fc]
#circle = np.asarray(circle)

with open('moon.txt') as fm:
    moon = [[float(x) for x in line.split(',')] for line in fm]
#moon = np.asarray(moon)
```

k-means clustering(k=2)

- data: input data(moon/circle)
- k: cluster group number
- iteration: count for iteration (←for distinguish each picture)

```
data = moon
k = 2
iteration = 0

centerProcess= []
dims = len(data[0])
centers = randc(dims,k)
centerProcess.append(centers)

clusterData = cluster(data, centers, dims, True)

while(True):
    iteration +=1
    newCenters = meanc(clusterData, centers, dims)
    centerProcess.append(newCenters)
    clusterData = cluster(data, newCenters, dims, False)
    plotshow(clusterData, newCenters, iteration)
    if centers == newCenters:
        break
    centers=newCenters
```

1. Randomly chosen group center
2. Record the set of data

3. First cluster

4. Using group mean to be an new set of center.
5. Update group center

- randc : Randomly form a group (x, y) from -3 to 3 as a group center

```
def randc(dim,k):
    c = []
    for i in range(k):
        center = []
        for d in range(dim):
            rand = random.uniform(-3,3)
            center.append(rand)
        c.append(center)
    return c
```

- clusterData : original data add a column with the value of the lable category.

```
def cluster(data, centers, dims, firstCluster):
    for point in data:
        nearestCenter = 0
        nearestCenterDist = None
        for i in range(0, len(centers)):
            euclidean = 0
            for d in range(0, dims):
                dist = abs(point[d] - centers[i][d])
                euclidean += dist
            euclidean = np.sqrt(euclidean)
            if nearestCenterDist == None:
                nearestCenterDist = euclidean
                nearestCenter = i
            elif nearestCenterDist > euclidean:
                nearestCenterDist = euclidean
                nearestCenter = i
        if firstCluster:
            point.append(nearestCenter)
        else:
            point[-1] = nearestCenter

    return data
```

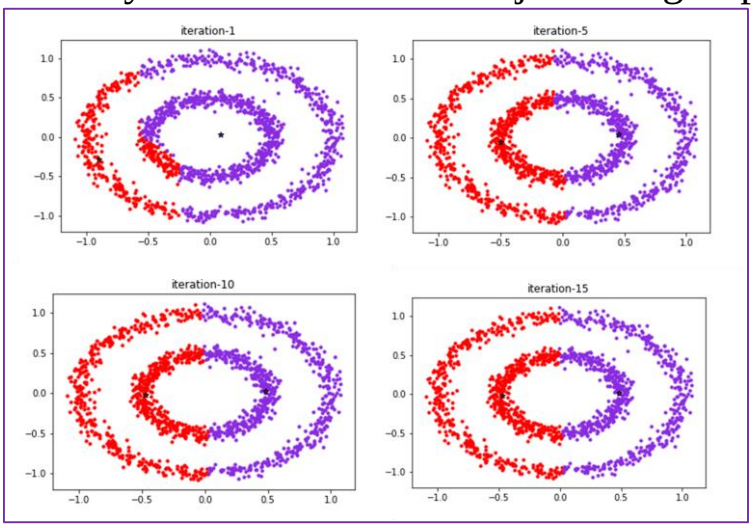
```
def meanc(data, centers, dims):
    print('centers:', centers)
    newCenters = []
    for i in range(len(centers)):
        newCenter = []
        npoints = 0
        points = []
        for point in data:
            if point[-1] == i:
                npoints += 1
                for dim in range(0,dims):
                    if dim < len(points):
                        points[dim] += point[dim]
                    else:
                        points.append(point[dim])
        if len(points) != 0:
            for dim in range(0,dims):
                newCenter.append(points[dim]/npoints)
            newCenters.append(newCenter)
        else:
            newCenters.append(centers[i])

    return newCenters
```

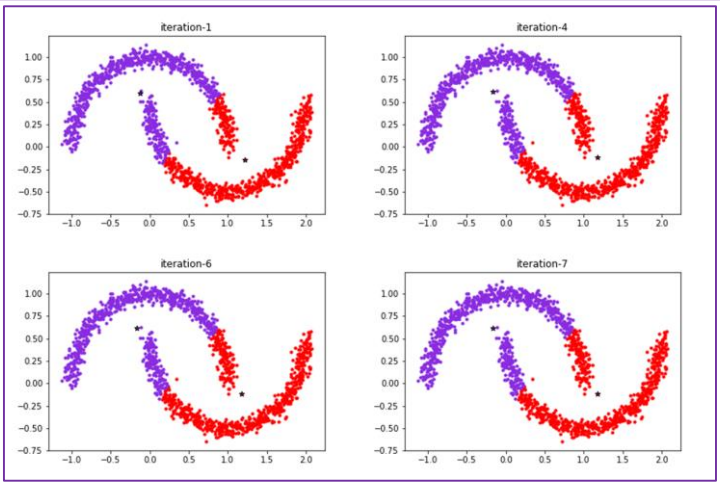
- plotshow : for output data

```
def plotshow(clusterData, centers,iteration):
    class2,class1,class0 = [],[],[]
    fileName = 'k-means{:d}.png'.format(iteration)
    for n in range(len(clusterData)):
        if clusterData[n][2]==1:
            class1.append(clusterData[n])
        elif clusterData[n][2]==0:
            class0.append(clusterData[n])
        else:
            class2.append(clusterData[n])

    class2 = np.asarray(class2)
    class1 = np.asarray(class1)
    class0 = np.asarray(class0)
    plt.title("iteration-" + str(iteration))
    plt.scatter(class1[: ,0],class1[: ,1], s=10, c='r')
    plt.scatter(class0[: ,0],class0[: ,1], s=10, c='blueviolet')
    plt.scatter(centers[0][0], centers[0][1],marker="*",s=40 ,c='blueviolet', edgecolors="black")
    plt.scatter(centers[1][0], centers[1][1],marker="*",s=40 ,c='r', edgecolors="black")
    if len(centers)>2:
        plt.scatter(class2[: ,0],class2[: ,1], s=10, c='b')
        plt.scatter(centers[2][0], centers[2][1],marker="*",s=40 ,c='b', edgecolors="black")
    plt.savefig(fileName)
    plt.show()
```

Result

According to the results, we can find that the most basic k-means can't divide circles and moons, but they can divide them into just two groups.

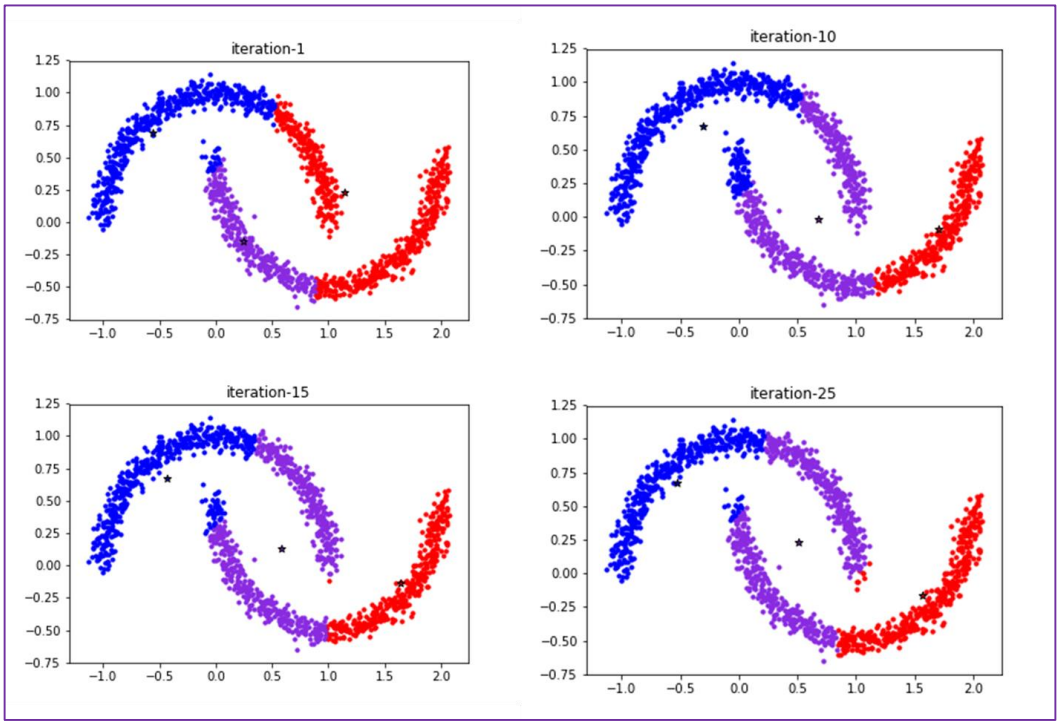


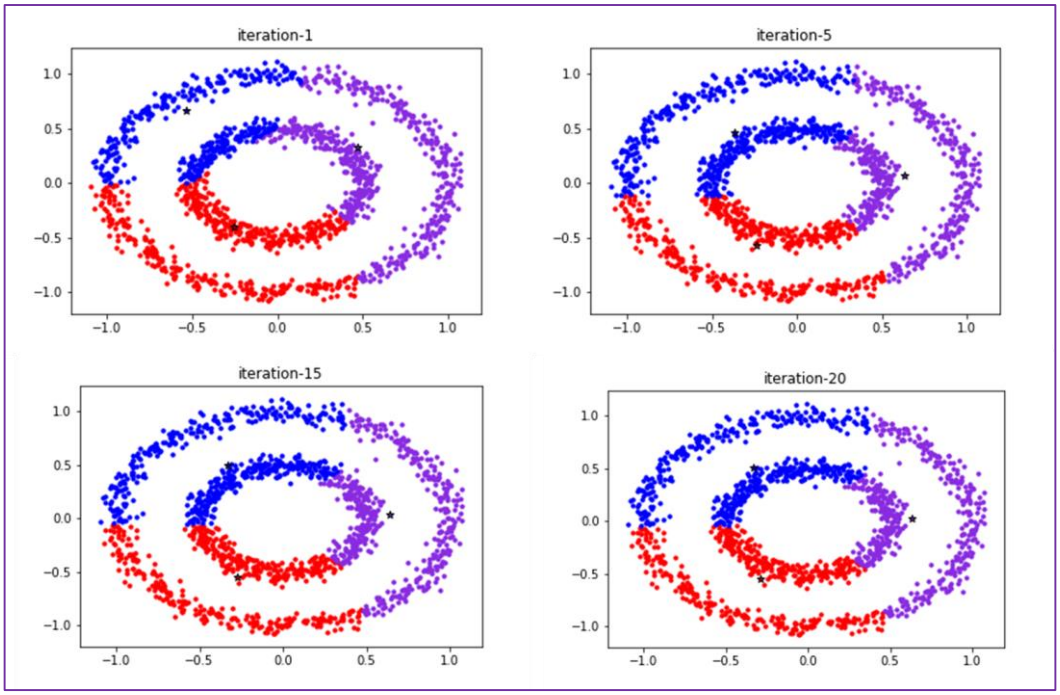


k-means clustering(k=3)

by iteration number, we can find that when k = 3, k-means need more time to converge.

And the distance between point and center, determine the category to which the point belongs.





kernel k-means(k=2)

Starting with this method, we have one more parameter - sigma

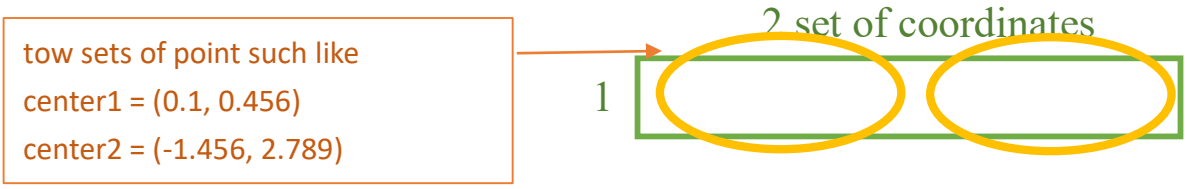
- sigma: the parameter for BRF kernel

In this method, first we need to do is creating a gram Matrix.

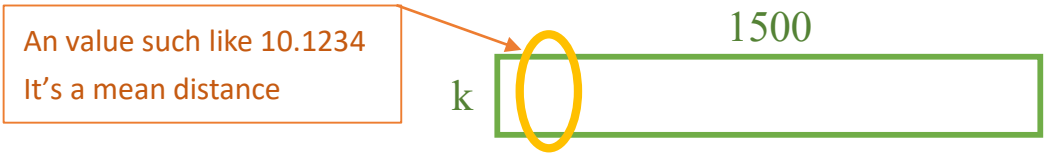
- gram Matrix: a matrix consisting of the distance between each point and other points. [in this case, we have a matrix of 1500*1500 size]

second, you need to distinguish between 「center」 and 「centerMatrix」.

- Center: a list for save the point we show in figure, such as below:



- centerMatrix: an array for save the center's gram matrix, such as below:




```
data = moon
k = 2
sigma = 0.25 # circle=0.15; moon =0.25
iteration = 0

clust = [[] for i in range(k)]
clustMat = [[] for i in range(k)]
newCenter = [[] for i in range(k)]
centerMat = np.zeros((k,len(data)))
newcenterMat = np.zeros((k,len(data)))

gramMat = RBF(data,sigma)# gramMat = 1500*1500
centerMat = initCentr(gramMat, k) #centerMat = k*1500

for ii in range(25):
    iteration+=1
    gramDist = [[] for i in range(k)] # gramDist = [[class0],[class1],[class2]]
    for i in range(k):
        gramDist[i] = np.linalg.norm((gramMat-centerMat[i]),axis=1)
    clustindex = np.argmin(gramDist, axis=0) #clustindex = 1500*1
    for i in range(k):
        clust[i] = data[np.where(clustindex[:] == i)]
        clustMat[i] = gramMat[np.where(clustindex[:] == i)]
    for i in range(k):
        newCenter[i]=(clust[i]).mean(axis=0)
        newcenterMat[i] = (clustMat[i]).mean(axis=0) #newcenterMat = k*1500
    plotshow(newCenter,clust,str(iteration))

    centerMat = newcenterMat
```

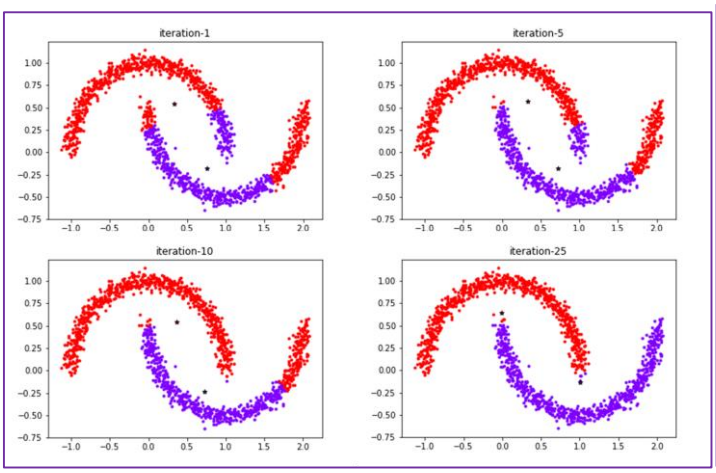
- RBF function follow the formula：

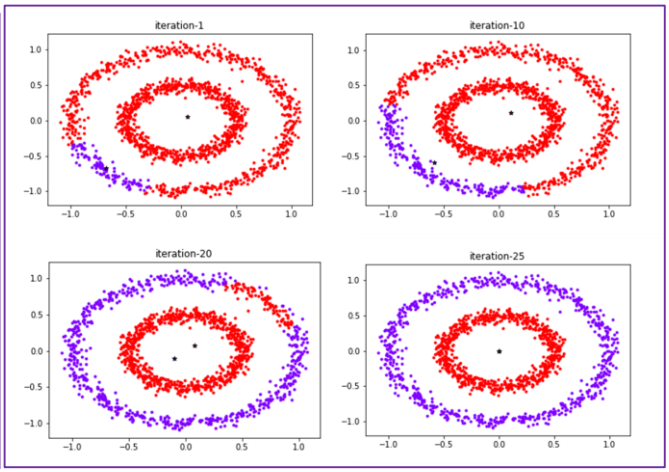
For each point in gram matrix: $e^{\frac{-/point1-\ point2/}{2\sigma^2}}$

Because the distance from A to B is the same as the distance from B to A. gram matrix should be an Symmetric matrix.

```
def initCentr(data, k):
    #centerMat = data[np.random.choice(len(data), k, replace=False)]
    for i in range(k):
        centerMat[i] = data[i]
    return centerMat

def RBF(data,sigma):
    nData = len(data)
    gramMat = np.zeros((nData,nData))
    for i in range(nData):
        delta = np.array(abs(data[i]-data[:]))
        sqrEuclid = (np.square(delta).sum(axis=1)).reshape(1,1500)
        gramMat[i] = np.exp(-(sqrEuclid)/(2*sigma**2))
    return gramMat

def plotshow(newCenter,clust,iteration):
    n = len(newCenter)
    color = iter(cm.rainbow(np.linspace(0, 1, n)))
    plt.clf()
    plt.title("iteration-" + iteration)
    fileName = 'kernelk{:s}.png'.format(iteration)
    for i in range(n):
        col = next(color)
        memberC = np.asarray(clust[i])
        plt.scatter(memberC[:, 0], memberC[:, 1], s=8, c=col)
    color = iter(cm.rainbow(np.linspace(0, 1, n)))
    for i in range(n):
        col = next(color)
        plt.scatter(newCenter[i][0], newCenter[i][1], s=30, c=col, marker="*", edgecolors="black")
    plt.savefig(fileName)
    plt.show()
```

Result

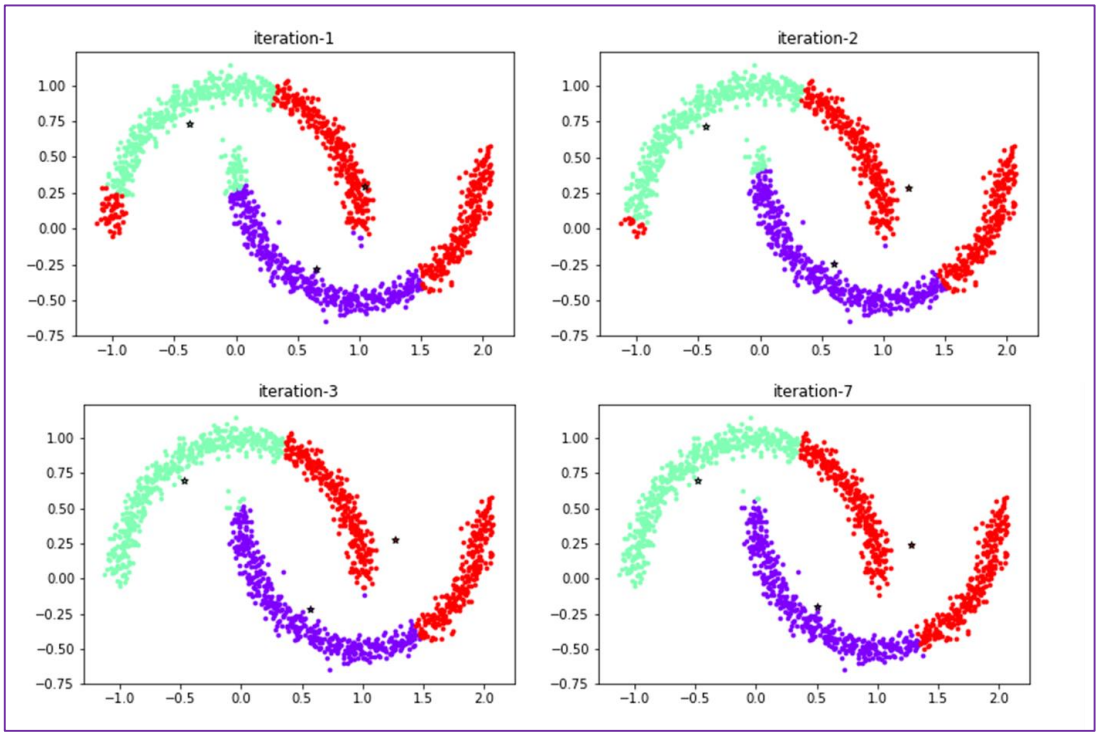


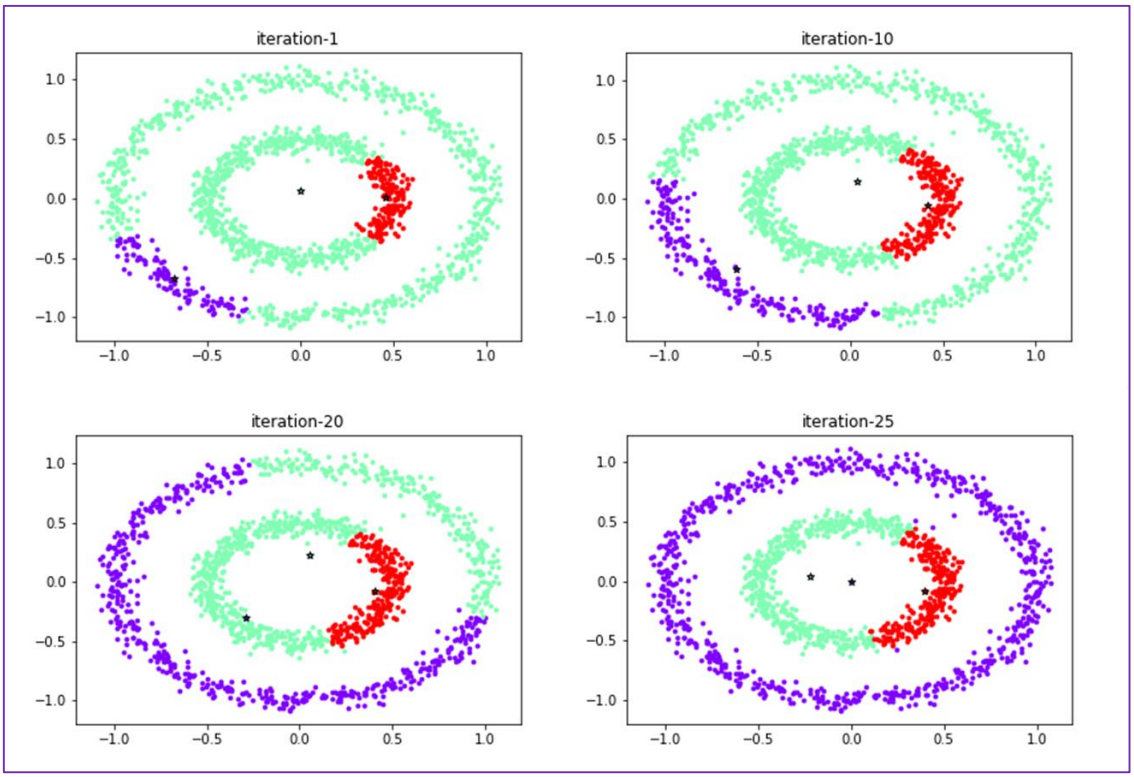


As you can see from the result, kernel k-means perfectly separates the moon and the circle.

kernel k-means(k=3)

To be honest, I don't quite know the correct appearance of circle and moon when k=3, but as you can see from the figure, they become a cluster in harmony at their proper distance.





spectral clustering(k=2)

Spectral clustering has the fastest iteration speed because of its pre-processing

```
circle = np.asarray(circle)
moon = np.asarray(moon)

data = moon
k = 2
sigma = 0.1 #circle = 0.1
iteration = 0

simMat = Simmilarity(data,sigma)
degMat = np.diag(np.array(simMat.sum(axis=1)).ravel())
lapMat = degMat - simMat #unnormalizedLaplacian
DataT = Spectral(lapMat)
initC = initCentr(DataT, k)

nCluster = len(initC)
```

Assign data to cluster whose centroid is the closest one.

```
while(True):
    iteration +=1
    euclidean = np.ndarray(shape=(len(DataT), 0))

    for i in range(0, nCluster):
        centRe = np.repeat(initC[i, :], len(DataT), axis=0)
        deltaMat = abs(DataT- centRe)
        euclideanMat = np.sqrt(np.square(deltaMat).sum(axis=1))
        euclidean = np.concatenate((euclidean, euclideanMat), axis=1)
    clusterMat = np.ravel(np.argmin(np.matrix(euclidean), axis=1))
    clusterT = [[] for i in range(k)]
    clusterOri = [[] for i in range(k)]
    for i in range(0, len(DataT)):
        clusterT[np.asscalar(clusterMat[i])].append(np.array(DataT[i, :]).ravel())
        clusterOri[np.asscalar(clusterMat[i])].append(np.array(data[i, :]).ravel())

    newCTr = np.ndarray(shape=(0, len(initC[0])))
    newCOri = np.ndarray(shape=(0, len(data[0])))
    for i in range(0,nCluster):
        centroidTr = np.asmatrix(clusterT[i]).mean(axis=0)
        centroidOri = np.asmatrix(clusterOri[i]).mean(axis=0)
        newCTr = np.concatenate((newCTr, centroidTr), axis=0)
        newCOri = np.concatenate((newCOri, centroidOri), axis=0)
    plotResult(clusterOri, newCOri, str(iteration), 0)

    if((initC == newCTr).all()):
        break

    initC = newCTr
```

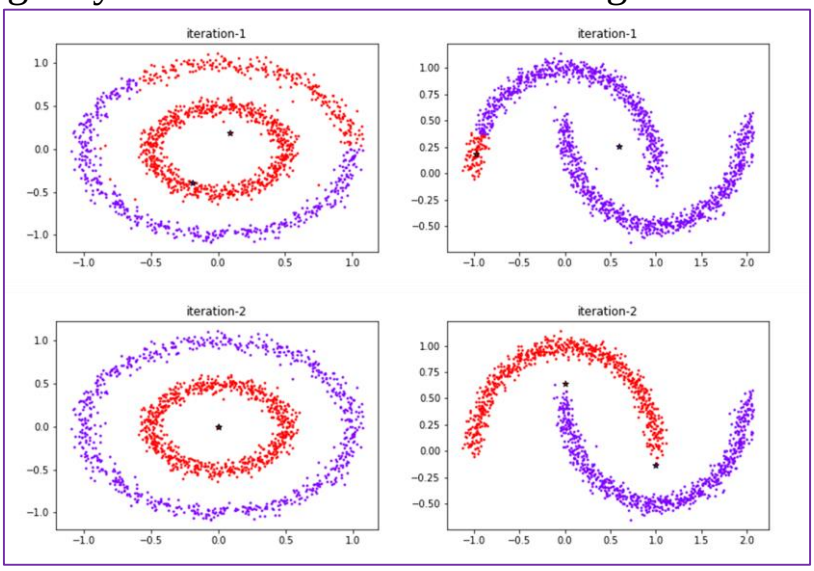
Function Simmilarity is consistent with the RBF function of kernel k means

```
def initCentr(data, k):
    result = data[np.random.choice(len(data), k, replace=False)]
    return result

def Simmilarity(data,sigma):
    nData = len(data)
    result = np.zeros((nData,nData))
    for i in range(nData):
        delta = np.matrix(abs(data[i]-data[:]))
        sqrEuclid = (np.square(delta).sum(axis=1)).reshape(1,1500)
        result[i] = np.exp(-(sqrEuclid)/(2*sigma**2))
    return result

def Spectral(laplacian):
    global k
    e_vals, e_vecs =np.linalg.eig.eig(np.array(laplacian))
    ind = e_vals.real.argsort()[:k]
    result = np.ndarray(shape=(len(laplacian),0))
    for i in range(1, len(ind)):
        cor_e_vec = np.transpose(np.array(e_vecs[:,np.asscalar(ind[i])]))
        result = np.concatenate((result, cor_e_vec), axis=1)
    return result
```

```
def plotshow(cluster, center, iteration, converged):
    n = len(cluster)
    color = iter(cm.rainbow(np.linspace(0, 1, n)))
    plt.figure("result")
    plt.clf()
    fileName = 'spectral{:s}.png'.format(iteration)
    plt.title("iteration-" + iteration)
    for i in range(n):
        col = next(color)
        member = np.asmatrix(cluster[i])
        plt.scatter(np.ravel(member[:, 0]), np.ravel(member[:, 1]), s=5, c=col)
    color = iter(cm.rainbow(np.linspace(0, 1, n)))
    for i in range(n):
        col = next(color)
        plt.scatter(center[i, 0], center[i, 1], marker="*", s=30, c=col, edgecolors="black")
    if(converged == 0):
        plt.ion()
        plt.savefig(fileName)
        plt.show()
        plt.pause(0.1)
    if (converged == 1):
        plt.savefig(fileName)
        plt.show(block=True)
```

Result

Spectral clustering only needs 1 or 2 times to converge



DBSCAN(k=2)

DBSCAN has no iterations. When judging each point, it has already decided which group he ultimately belongs to.

- eps: a threshold distance, such like drawing circle radius.
- MinPts: Determine whether it is the threshold for the same group, which is the minimum required number of neighbors.
- labels: save the final cluster assignment for each point in dataset.

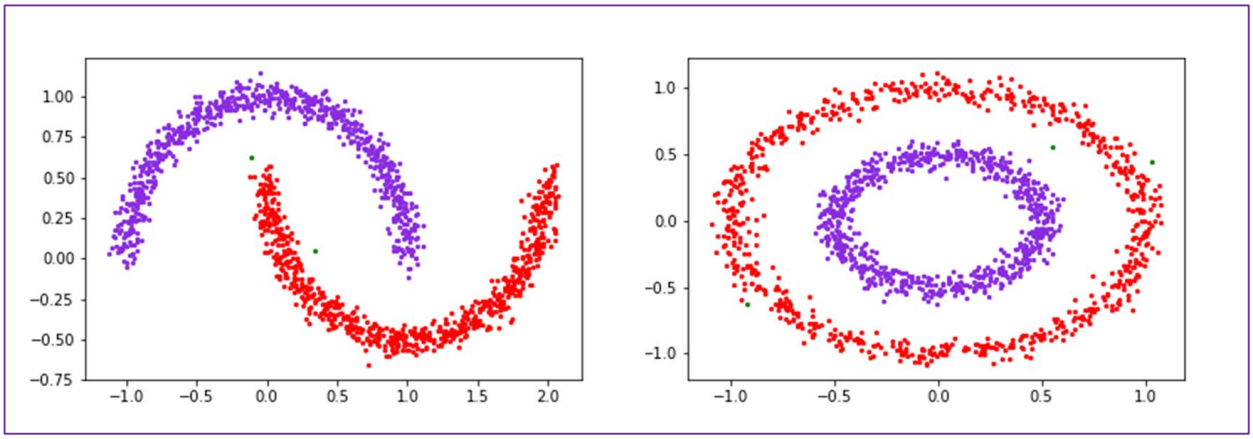
```
data = moon
eps=0.05
MinPts=4

labels = [0]*len(data)
cluster = 0

for point in range(len(data)):
    if not (labels[point] == 0):
        continue

    NeighborPts = regionQuery(data, point, eps)

    if len(NeighborPts) < MinPts:
        labels[point] = -1
    else:
        cluster += 1
        growCluster(data, labels, point, NeighborPts, cluster, eps, MinPts)
```

```
def regionQuery(data, point, eps):
    neighbors = []
    data = np.asarray(data)
    for Pn in range(0, len(data)):
        if np.linalg.norm(data[point] - data[Pn]) < eps:
            neighbors.append(Pn)

    return neighbors

def growCluster(data, labels, point, NeighborPts, cluster, eps, MinPts):
    labels[point] = cluster
    i = 0
    while i < len(NeighborPts):
        Pn = NeighborPts[i]
        if labels[Pn] == -1:
            labels[Pn] = cluster
        elif labels[Pn] == 0:
            labels[Pn] = cluster
            PnNeighborPts = regionQuery(data, Pn, eps)
            if len(PnNeighborPts) >= MinPts:
                NeighborPts = NeighborPts + PnNeighborPts
        i += 1
```

```
data = np.asarray(data)
labels = np.asarray(labels)
plt.scatter(data[np.where(labels == 1)][:, 0], data[np.where(labels == 1)][:, 1], c='r', s=5)
plt.scatter(data[np.where(labels == 2)][:, 0], data[np.where(labels == 2)][:, 1], c='blueviolet', s=5)
plt.scatter(data[np.where(labels == -1)][:, 0], data[np.where(labels == -1)][:, 1], c='g', s=5)
plt.savefig("DBSCAN.png")
plt.show()
```

Result

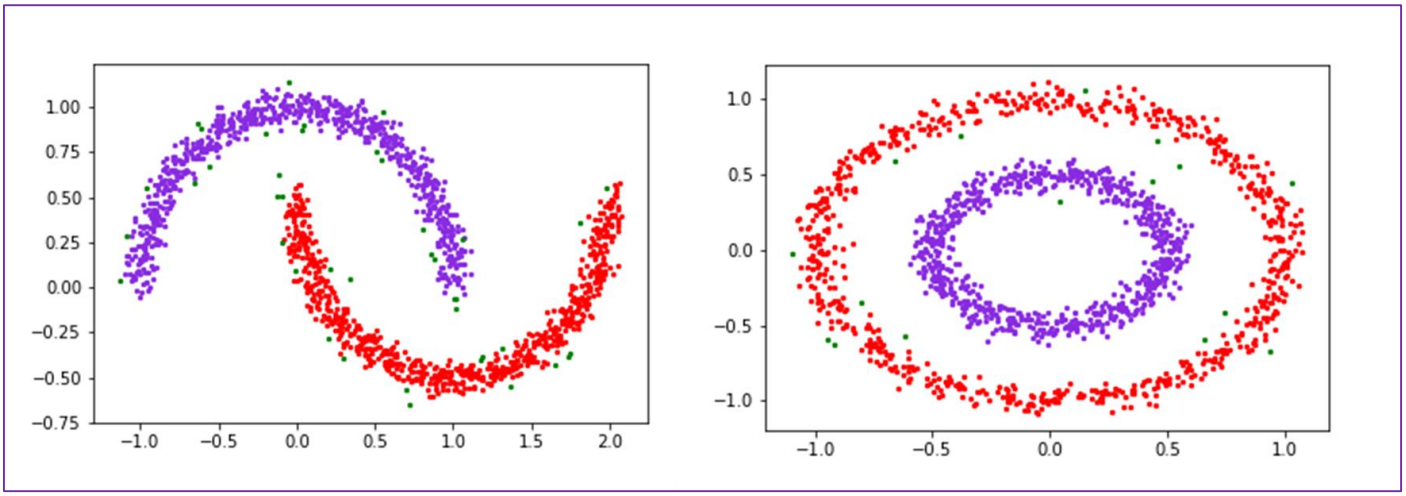
Red is a group, purple is another group, green is a noise.



DBSCAN(eps=0.05, MinPts=4)

Result

When the conditions become harsh, you can see that the green dots representing the noise become more.



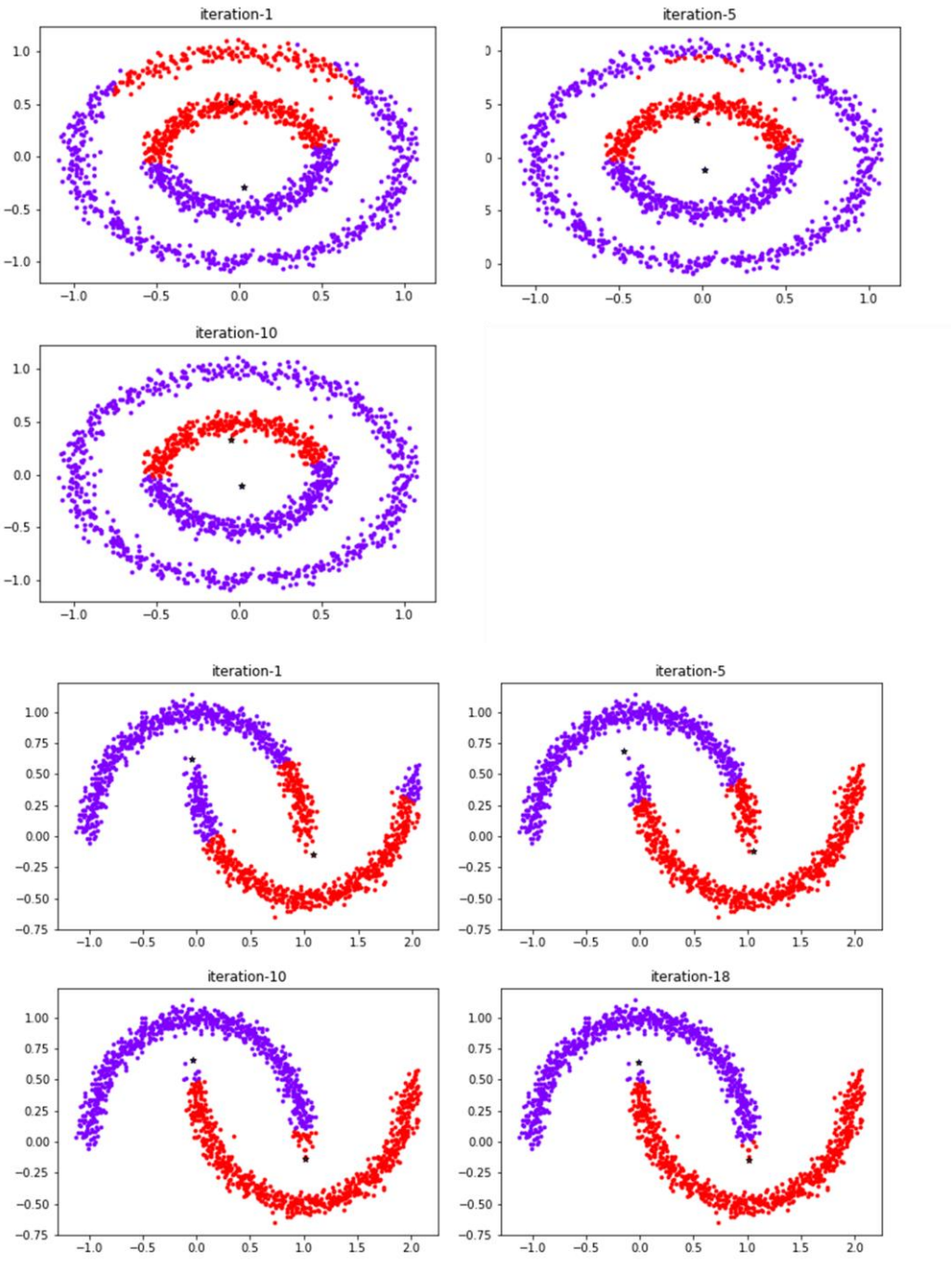
k-means++_with kernel k-means

When selecting the initial center with k-means++, one will be randomly selected first, and as far as possible (by probability), the point farther from this point is selected as the initial value of the other center point.

```
def initCent(data, k):
    euclidean = np.ndarray(shape=(len(data), 0))
    allCent = np.ndarray(shape=(0,len(data[0])))
    first = data[np.random.choice(len(data), 1, replace=False)]
    allCent = np.concatenate((allCent, first), axis=0)
    repeatC = np.repeat(first, len(data), axis=0)
    deltaMat = abs(data-repeatC)
    euclideanMat = np.sqrt(np.square(deltaMat).sum(axis=1))
    indexNextC = (np.argmax(np.array(euclideanMat)))
    if(k>1):
        for a in range(1,k):
            nextC = np.matrix(data[np.asscalar(indexNextC),:])
            allCent = np.concatenate((allCent, nextC), axis=0)
            for i in range(0,len(allCent)):
                repeatC = np.repeat(allCent[i,:], len(data), axis=0)
                deltaMat = abs(data-repeatC)
                euclideanMat = np.sqrt(np.square(deltaMat).sum(axis=1))
                euclidean = np.concatenate((euclidean, euclideanMat), axis=1)
            euclideanFinal = np.min(np.array(euclidean), axis=1)
            indexNextC = np.argmax(np.array(euclideanFinal))
    result = allCent

    return result
```

Result

Because the initial value is not fixed, it is difficult for me to quickly find the appropriate parameters (sigma). The following results are the results of several excellent tests.

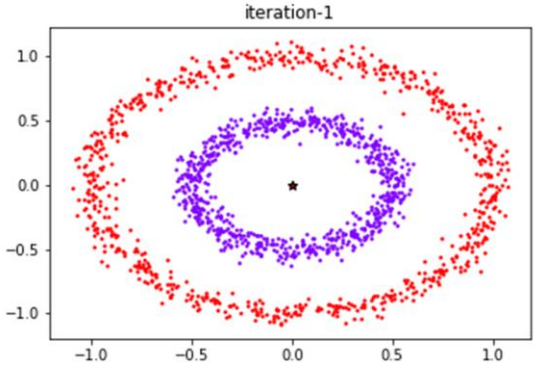


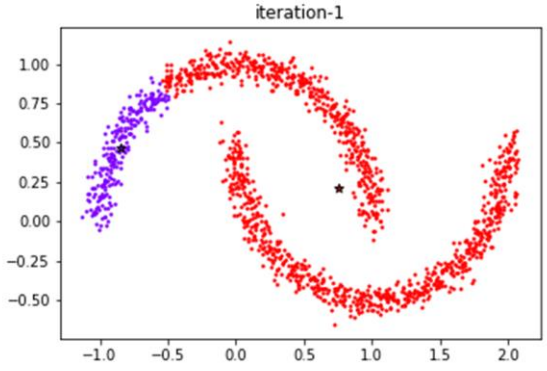
k-means++_with spectral clustering

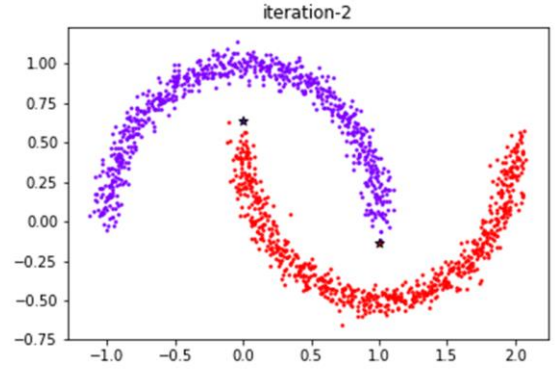
Use the same k-means++ as kernel k-means

Result

Because spectral clustering can converge quickly, there is no difference in the choice between using Random and k-means++ as initial center chosen.


iteration-1
1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0


iteration-1
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0


iteration-2
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0