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

Before the clustering:

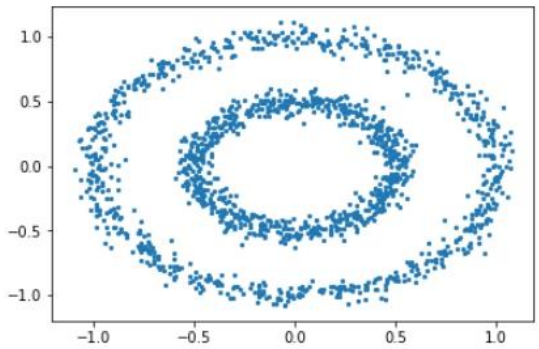
I load each txt file to variable data.

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
content = []
with open('circle.txt') as f:
    for lines in f.readlines():
        content.append(lines.strip().split(','))

data = np.array(content).astype('float')
data
```

```
array([[-0.67799938, -0.69875698],
       [ 0.93143746,  0.19139133],
       [ 0.54829131, -0.00601715],
       ...,
       [-0.34518816, -0.35804797],
       [ 0.01719727, -0.94513802],
       [ 0.91377877, -0.59884164]])
```

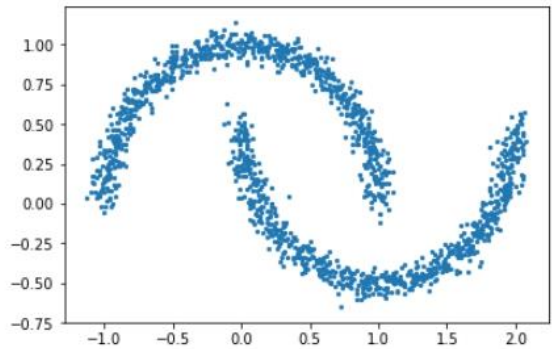
Circle.txt

```
plt.scatter(data[:, 0], data[:, 1], s=5)
plt.show()
```




Moon.txt

```
plt.scatter(data[:, 0], data[:, 1], s=5)
plt.show()
```



### 1. *k-means clustering*

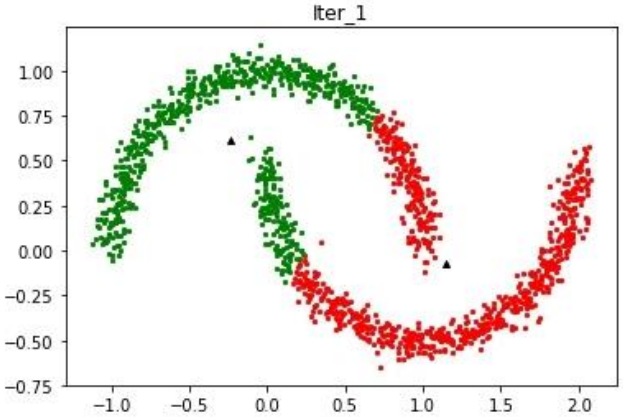
I took k-means into two parts: first is pick the centers for initial, second is calculate each point in data and find the min distance into there clusters.

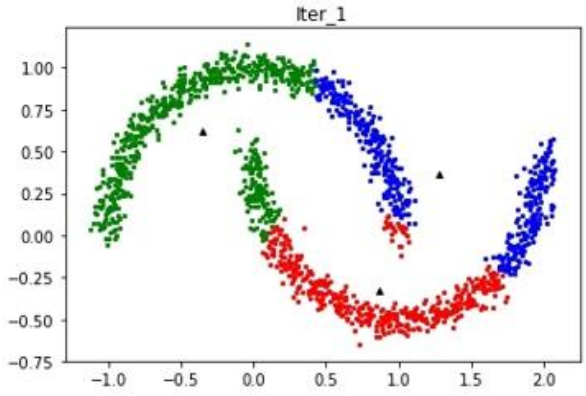
```
def Kmeans(dataset, K=2):
    color = ['r', 'g', 'b']
    last = np.zeros((K,dataset.shape[1]))
    # init center point
    centers = np.zeros((K, dataset.shape[1]))
    for i in range(K):
        centers[i] = dataset[i]
```

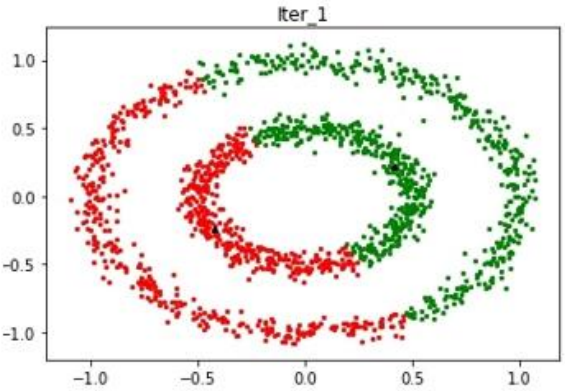
The picture above , I pick first data in dataset be my first cluster center, and pick second data be my second cluster center. The reason why I use this method is that I can easily debug when I was coding.

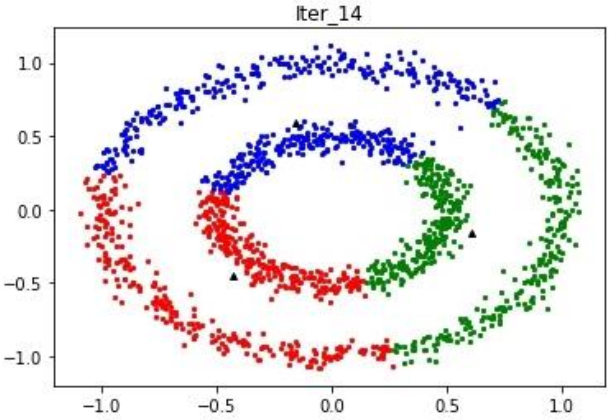
```
    for times in range(15):
        dists = []
        for i in range(K):
            dist = (dataset - centers[i])*(dataset - centers[i])
            dists.append(np.sqrt(np.sum(dist, axis=1)))

        ans = np.array(dists)
        result = np.argmin(ans, axis=0)

        # 收入相同的Cluster
        clusters = []
        for i in range(K):
            clusters.append(dataset[np.where(result == i)])
            centers[i] = np.mean(np.array(clusters[i]), axis=0)
#             print("Cluster_{}: {}".format(i, clusters[i]))
            print("Center_{}: {}".format(i, centers[i]))
            plt.scatter(clusters[i][:, 0], clusters[i][:, 1], c=color[i], s=5)
        plt.scatter(centers[:, 0], centers[:,1], marker='^', s=15, c='k')
        plt.title("Iter_{}".format(times))
        plt.show()
        if (last == centers).all():
            break
#             print("OK")
        last = centers
```

Next part, I calculate the center of each clusters for 15 times, if the last center coordinate is same as this times center it will be break .


Iter_1
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


Iter_1
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


Iter_1
1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0


Iter_14
1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0

### 2. *kernel k-means*

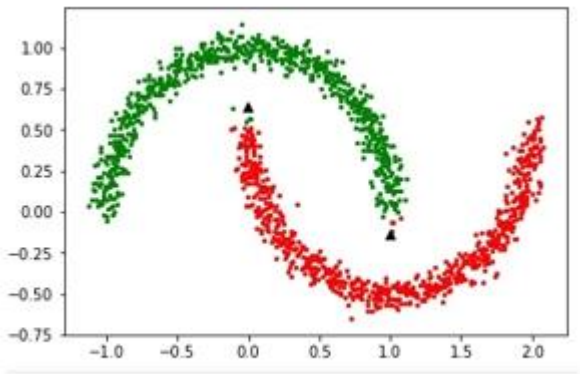
I use RBF kernel function: e^((|x-y|^2)/(2*sigma^2)).

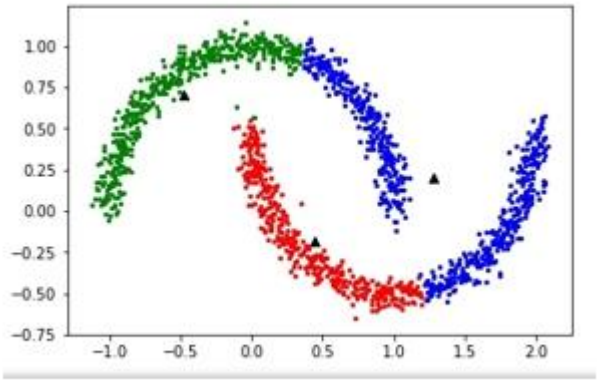
For the picture below, first I initial a zero gram matrix for size(n, n), n for the instances of the data. Np.linalg.norm can help me calculate the dist between two vectors. Thus, I can complete my gram matrix quickly.

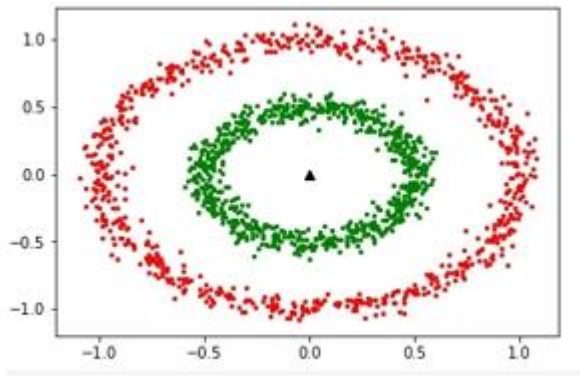
```
def kernel_KMeans(dataset, sigma, K=2):
    n = len(dataset)
    gram = np.zeros((n, n))
    # Gram matrix
    for index, point in enumerate(dataset):
        dist = np.linalg.norm((dataset - point), axis=1).reshape(1, -1)
        gram[index] = np.exp((-1 * dist**2) / (2 * sigma**2))
```

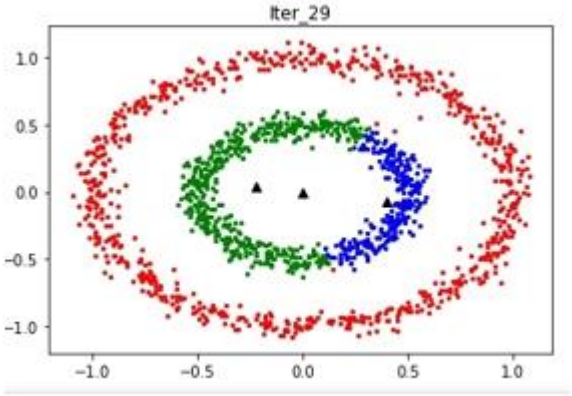
I used same pick method for my initial cluster center, after that I try hard to find the better sigma in order to separate the points into two clusters. I found that if the data is “moon.txt”, the sigma should be 0.25; if data is “circle.txt” , the sigma should be 0.15. For kernel k-means, it’s same method to calculate two vectors, the only different is kernel k means turn the original dataset into higher dimension.

```
    # Centers
    centers = np.zeros((K, gram.shape[1]))
    for i in range(K):
        centers[i] = gram[i]
#     centers = kpp(gram)

    for _ in range(30):
        # Distance from each center
        dists = np.zeros((K, gram.shape[1]))
        for i in range(K):
            dist = np.linalg.norm(gram - centers[i], axis=1)
            dists[i] = dist
        result = np.argmin(dists, axis=0)
        clusters = []
        t = []
        color = ['r', 'g', 'b']
        for i in range(K):
            clusters.append(gram[np.where(result == i)])
            t.append(data[np.where(result == i)])
            centers[i] = np.mean(np.array(clusters[i]), axis=0)
            print(np.mean(t[i], axis=0))
            print("Center_{}: {}".format(i, centers[i]))
            plt.scatter(t[i][:, 0], t[i][:, 1], c=color[i], s=5)
            plt.scatter(np.mean(t[i], axis=0)[0], np.mean(t[i], axis=0)[1], marker='^', c='k')
#         plt.savefig("kernel_moon{}.png".format(_))
        plt.show()
```


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


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


1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0


Iter_29
1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0

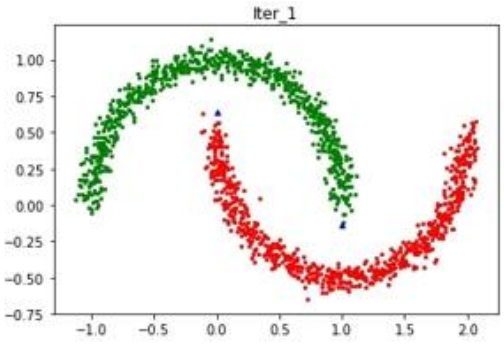
### *3. spectral clustering*

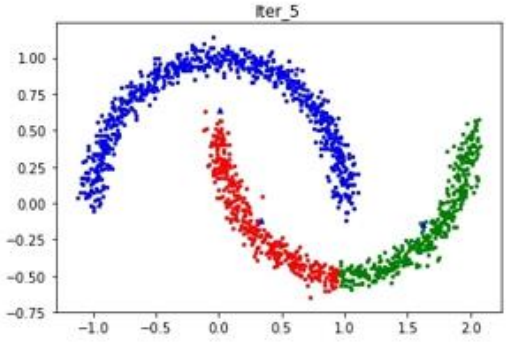
```
def similarity(dataIn, sigma=0.1):
    n = len(dataIn)
    result = np.zeros((n, n))
    # Weight
    for index, point in enumerate(dataIn):
        dist = np.linalg.norm((dataIn - point), axis=1).reshape(1, -1)
        result[index] = np.exp((-1 * dist**2) / (2 * sigma**2))
    return result

def degree(similarityMatrix):
    diag = similarityMatrix.sum(axis=1)
    result = np.diag(diag)
    return result

def spectrum(laplacian, K=2):
    eig_vals, eig_vecs = np.linalg.eig(laplacian)
    ind = eig_vals.real.argsort()[:K]
    result = np.ndarray(shape=(laplacian.shape[0],0))
    for i in range(1, ind.shape[0]):
        cor_e_vec = np.transpose(np.matrix(eig_vecs[:,np.asscalar(ind[i])]))
        result = np.concatenate((result, cor_e_vec), axis=1)
    return result

W = similarity(data)
D = degree(W)
L = D-W
transformedData = spectrum(L)
```

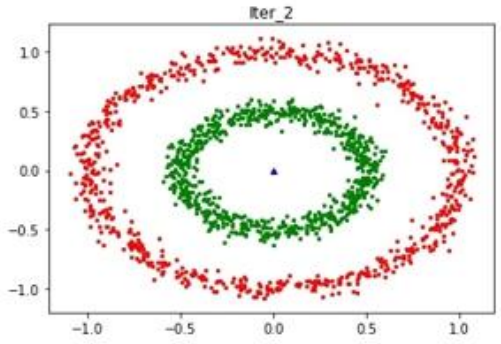
The picture above, I calculate three different matrix. One is W, which build a similarity matrix with RBF kernel by similarity function. Another is D, which is degree matrix and count the similar data by degree function. The other is spectrum function, which is to calculate Laplacian matrix (Graph Laplacian). We should find out K smallest eigenvalue, and the eigenspace corresponding to 0 is spanned by k mutually orthogonal vectors. The vectors are indicator vectors of graph's connected components.

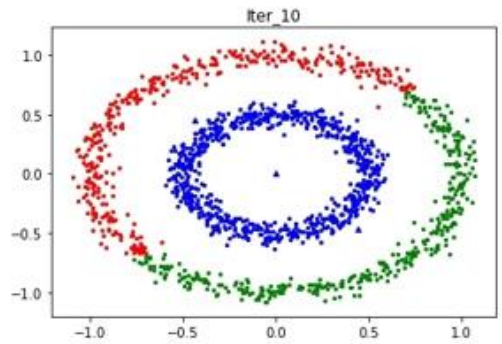
The things I do, is to transfer origin dataset into Graph Laplacian and find the eigenvalue and eigenvectors, after that I take my transformedData to do the same method to find their own groups. The different between spectral clustering and kernel k means is that spectral clustering can quickly find out the clusters with few iterations. (Always in one iter)

```python
def sepctral_clustering(dataset, transformedData, centers):
    nCluster = centers.shape[0]
    global k
    times = 0
    while(True):
        times +=1
        dists = np.zeros((transformedData.shape[0], 0))
        for i in range(nCluster):
            dist = np.linalg.norm(transformedData - centers[i], axis=1).reshape(-1, 1)
            dists = np.concatenate((dists, dist), axis=1)

        clusters = np.ravel(np.argmin(np.matrix(dists), axis=1))
        cluster_mem = []
        cluster_data = []
        for i in range(nCluster):
            cluster_mem.append(transformedData[np.where(clusters == i)])
            cluster_data.append(dataset[np.where(clusters == i)])

        new_centers = np.ndarray(shape=(0, centers.shape[1]))
        new_data = np.ndarray(shape=(0, dataset.shape[1]))

        for i in range(nCluster):
            centroidClusterTransf = np.asmatrix(cluster_mem[i]).mean(axis=0)
            centroidClusterOri = np.asmatrix(cluster_data[i]).mean(axis=0)
            new_centers = np.concatenate((new_centers, centroidClusterTransf), axis=0)
            new_data = np.concatenate((new_data, centroidClusterOri), axis=0)

        if((centers == new_centers).all()):
            break

        centers = new_centers
        color = ['r', 'g', 'b']
        for i in range(nCluster):
            plt.scatter(cluster_data[i][:, 0], cluster_data[i][:,1], c=color[i], s=5)
        center = np.array(new_data)
        plt.scatter(center[:,0], center[:,1], c='b', s=15, marker='^')
        plt.title("Iter_{}".format(times))
        plt.show()
```



***4. DBSCAN***

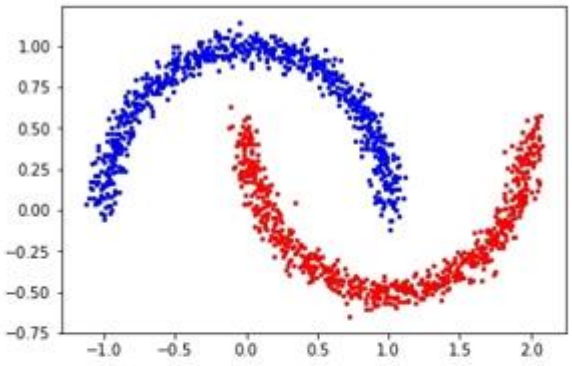
First , I initial my label in zero, which means it doesn't find the clustering yet. Next, I pick the data in order, and calculate each data point has how many neighbors nearby it within radius. If the data point's neighbors larger than MinPts we will give the data point in a cluster and also find out it's each sub data point in the neighbors. However, if the data point's neighbors is smaller than MinPts, then it will be labeled as -1 (noise)

```
def DBSCAN(Dataset, radius, MinPts):
    labels = [0] * len(Dataset)
    cluster_num = 0
    for index in range(len(Dataset)):
        if not (labels[index] == 0):
            continue
        NeighborPts = getNeighbors(Dataset, index, radius)
        if len(NeighborPts) < MinPts:
            labels[index] = -1
        else:
            cluster_num += 1
            update(Dataset, labels, index, NeighborPts, cluster_num, radius, MinPts)
    return labels
```

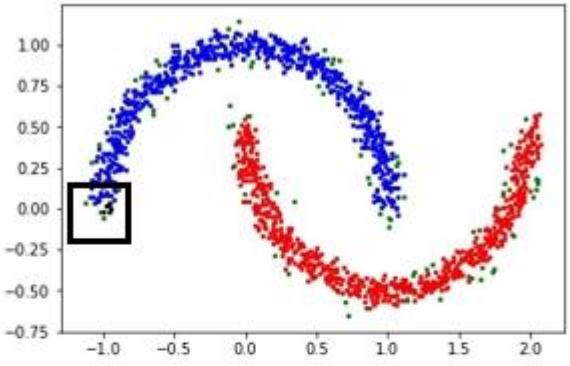
The picture below, has two function, update function is to label each data point in neighbor points, if the data point's neighbors > MinPts , it will be labeled as same as the neighbor's label, otherwise it will be -1 for noise. getNeighbors is to find out the point n's neighbors.

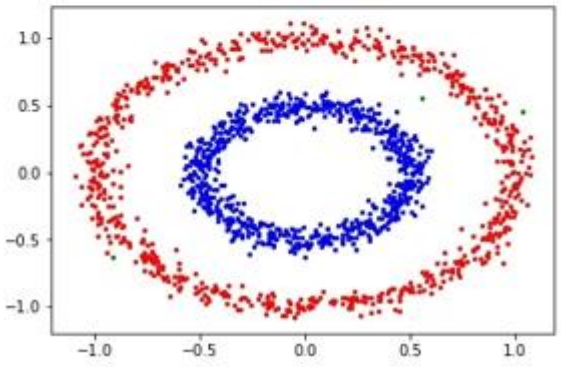
```
def update(Dataset, labels, index, NeighborPts, cluster_num, radius, MinPts):
    labels[index] = cluster_num
    i = 0
    while i < len(NeighborPts):
        neighbor_index = NeighborPts[i]
        if labels[neighbor_index] == -1:
            labels[neighbor_index] = cluster_num
        elif labels[neighbor_index] == 0:
            labels[neighbor_index] = cluster_num
            PnNeighborPts = getNeighbors(Dataset, neighbor_index, radius)
            if len(PnNeighborPts) >= MinPts:
                NeighborPts = NeighborPts + PnNeighborPts
        i += 1


def getNeighbors(Dataset, index, radius):
    neighbors = []
    for node in range(len(Dataset)):
        if np.linalg.norm(Dataset[index] - Dataset[node]) < radius:
            neighbors.append(node)
    return neighbors
```

DBSCAN(data, 0.2, 5)

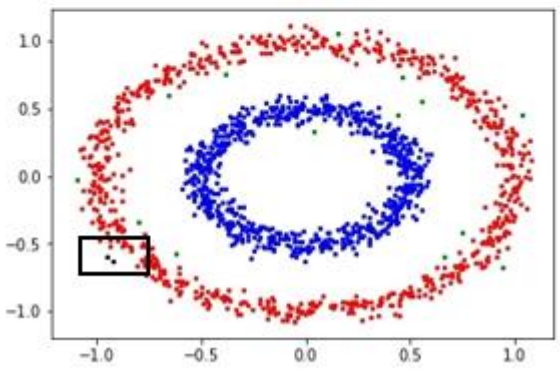



DBSCAN(data, 0.05, 6)




DBSCAN(data, 0.1, 5)




DBSCAN(data, 0.07, 2)

***K means++***

For the k-means++, first I randomly pick one point, and then calculate each distance between the point and all the data. Second I give each point different weight, the far distance get the higher weight, it means it will have higher chance to be select in next cluster center.

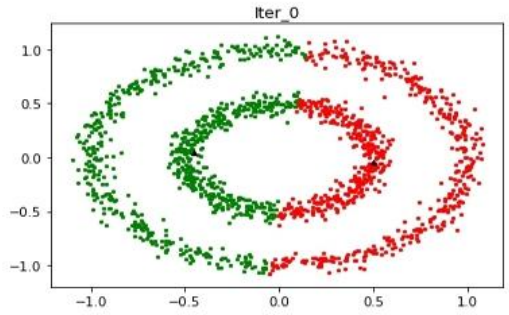
```python
def kpp(dataset, K=2):
    centers = np.zeros((K, dataset.shape[1]))
    pick_num = np.random.choice(len(dataset), 1)[0]
    print("pick_num = {}".format(pick_num))
    centers[0] = dataset[pick_num]

    dist = np.linalg.norm((dataset - centers[0]), axis=1)**2
    p = dist/dist.sum()
    another = np.random.choice(len(dataset), 1, p=p)[0]
    print("pick_num = {}".format(another))
    centers[1] = dataset[another]

    return centers
```

K means(with k means++)

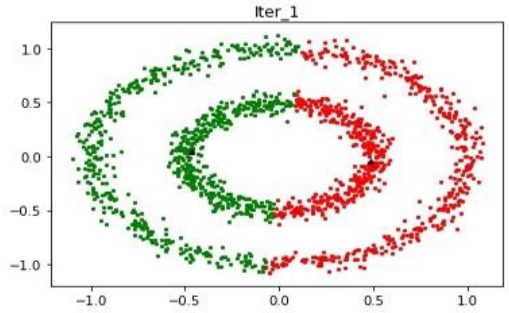
```python
def Kmeans(dataset, K=2):
    color = ['r', 'g', 'b']
    last = np.zeros((K,dataset.shape[1]))
    # init center point
#     centers = np.zeros((K, dataset.shape[1]))
#     for i in range(K):
#         centers[i] = dataset[i]
    centers = kpp(dataset)

    for times in range(15):
        dists = []
        for i in range(K):
            dist = (dataset - centers[i])*(dataset - centers[i])
            dists.append(np.sqrt(np.sum(dist, axis=1)))

        ans = np.array(dists)
        result = np.argmin(ans, axis=0)

        # 收入相同的Cluster
        clusters = []
        for i in range(K):
            clusters.append(dataset[np.where(result == i)])
            centers[i] = np.mean(np.array(clusters[i]), axis=0)
#             print("Cluster_{}: {}".format(i, clusters[i]))
            print("Center_{}: {}".format(i, centers[i]))
            plt.scatter(clusters[i][:, 0], clusters[i][:, 1], c=color[i], s=5)
        plt.scatter(centers[:, 0], centers[:,1], marker='^', s=15, c='k')
        plt.title("Iter_{}".format(times))
        plt.savefig("moon_iter{}.png".format(times))
        plt.show()
        if (last == centers).all():
            break
#             print("OK")
        last = centers
```

```
Kmeans(data,2)
```

```
pick_num = 548
pick_num = 556
Center_0: [ 0.49514868 -0.04928731]
Center_1: [-0.45416236  0.04400233]
```

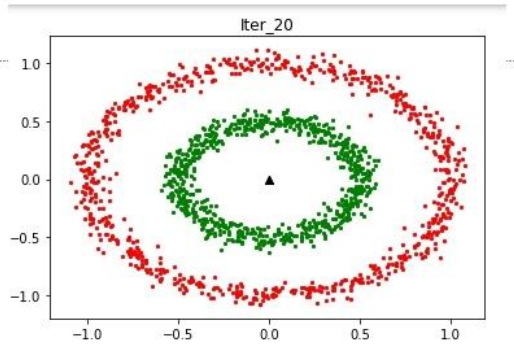



```
Center_0: [ 0.48519751 -0.04686608]
Center_1: [-0.46445586  0.04363086]
```



Kernel k means(with k means++)

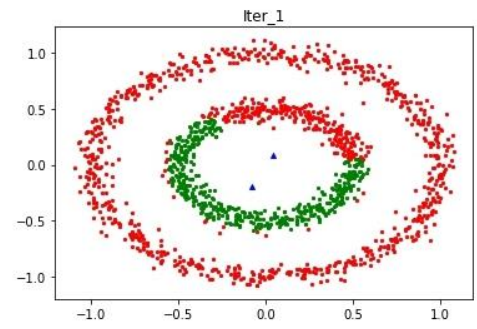
```python
def kernel_KMeans(dataset, sigma, K=2):
    n = len(dataset)
    gram = np.zeros((n, n))
    # Gram matrix
    for index, point in enumerate(dataset):
        dist = np.linalg.norm((dataset - point), axis=1).reshape(1, -1)
        gram[index] = np.exp((-1 * dist**2) / (2 * sigma**2))

    # Centers
#     centers = np.zeros((K, gram.shape[1]))
#     for i in range(K):
#         centers[i] = gram[i]
    centers = kpp(gram)

    for _ in range(30):
        # Distance from each center
        dists = np.zeros((K, gram.shape[1]))
        for i in range(K):
            dist = np.linalg.norm(gram - centers[i], axis=1)
            dists[i] = dist
        result = np.argmin(dists, axis=0)
        clusters = []
        t = []
        color = ['r', 'g', 'b']
        for i in range(K):
            clusters.append(gram[np.where(result == i)])
            t.append(data[np.where(result == i)])
            centers[i] = np.mean(np.array(clusters[i]), axis=0)
            print(np.mean(t[i], axis=0))
            print("Center_{}: {}".format(i, centers[i]))
            plt.scatter(t[i][:, 0], t[i][:, 1], c=color[i], s=5)
            plt.scatter(np.mean(t[i], axis=0)[0], np.mean(t[i], axis=0)[1], marker='^', c='k')
            plt.title("Iter_{}".format(_))
#         plt.savefig("kernel_moon_3{}.png".format(_))
        plt.show()
```

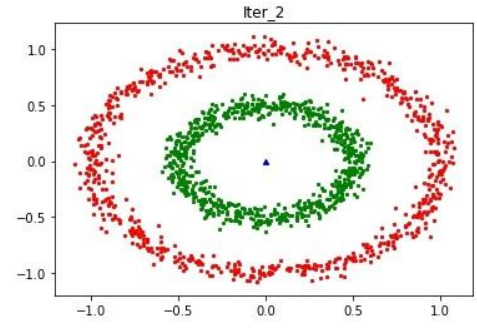



Spectral clustering (with k means++)

```python
def sepctral_clustering(dataset, transformedData, centers):
    nCluster = centers.shape[0]
    global k
    times = 0
    while(True):
        times +=1
        dists = np.zeros((transformedData.shape[0], 0))
        for i in range(nCluster):
            dist = np.linalg.norm(transformedData - centers[i], axis=1).reshape(-1, 1)
            dists = np.concatenate((dists, dist), axis=1)

        clusters = np.ravel(np.argmin(np.matrix(dists), axis=1))
        cluster_mem = []
        cluster_data = []
        for i in range(nCluster):
            cluster_mem.append(transformedData[np.where(clusters == i)])
            cluster_data.append(dataset[np.where(clusters == i)])

        new_centers = np.ndarray(shape=(0, centers.shape[1]))
        new_data = np.ndarray(shape=(0, dataset.shape[1]))

        for i in range(nCluster):
            centroidClusterTransf = np.asmatrix(cluster_mem[i]).mean(axis=0)
            centroidClusterOri = np.asmatrix(cluster_data[i]).mean(axis=0)
            new_centers = np.concatenate((new_centers, centroidClusterTransf), axis=0)
            new_data = np.concatenate((new_data, centroidClusterOri), axis=0)

        if((centers == new_centers).all()):
            break

        centers = new_centers
        color = ['r', 'g', 'b']
        for i in range(nCluster):
            plt.scatter(cluster_data[i][:, 0], cluster_data[i][:,1], c=color[i], s=5)
        center = np.array(new_data)
        plt.scatter(center[:,0], center[:,1], c='b', s=15, marker='^')
        plt.title("Iter_{}".format(times))
        plt.show()


# centers = initCentroid(transformedData, initCentroidMethod, k)
centroidInit = kpp(transformedData)
sepctral_clustering(data, transformedData, centers)
```

```
pick_num = 964
pick_num = 1128
```




DBSCAN no k means++

GIF in file

I make some gif in the file. DBSCAN just in 1 iter so I didn’t make the gif.