

## ML HW6 Report

309551131 楊子脩

a. code with explanations:

Part1:

I. K-means:

First, we need to read image data from files. So I implemented a function “readImage” using PIL package to read image in:

```
def readImage():  
    image = Image.open(args.input_image)  
    data = np.array(image.getdata())  
    return data
```

Next, we need to compute the kernel using the formula provided in HW spec, that is,

$$k(x, x') = e^{-\gamma_s \|S(x) - S(x')\|^2} \times e^{-\gamma_c \|C(x) - C(x')\|^2}$$

according to this formula, I defined a function to compute kernel:

```
def computeKernel(data_x):  
    """  
    Kernel formula according to HW spec  
    """  
    #color distance  
    color_dist = cdist(data_x, data_x, 'sqeuclidean')  
    grid = np.indices((100,100)).reshape(2,10000,1)  
  
    #pixel coordinate  
    coordinate_repre = np.hstack((grid[0], grid[1]))  
  
    #spatial distance  
    spatial_dist = cdist(coordinate_repre, coordinate_repre, 'sqeuclidean')  
    gram_matrix = np.multiply(np.exp(-args.gamma_s * spatial_dist), np.exp(-args.gamma_c * color_dist))  
  
    return gram_matrix
```

the formula needs color distance and spatial distance, so we need to calculate them first, then combined with hyperparameters we provide( I use 0.0001 for gamma s, 0.001 for gamma c) to obtain the kernel matrix.

After getting kernel matrix, we need to do the initial clustering.

```
def initialCluster(x,num_of_cluster, kernel, mode):
    """
    initialize cluster
    mode 0 for first randomly choose centers
    mode 1 for k-means++
    """
    images = []
    centers= chooseCenters(x, mode, kernel)
    cluster = np.zeros(num_of_points, dtype = int)

    for i in range(num_of_points):
        min_dist = np.full(num_of_cluster, np.inf)
        for j in range(num_of_cluster):
            min_dist[j] = distance(i, centers[j], gram_matrix)
        cluster[i] = np.argmin(min_dist)
    images.append(outputImage(cluster,"kmeans",0))
    return cluster, images
```

From the above picture, when initializing, we need to first have some centers to perform the first clustering(the number is decided by the provided argument, default is 3). While choosing centers, there are two ways to do so. Default is to choose randomly:

```
def chooseCenters(x,mode, kernel):
    if 0 == mode:
        return np.random.choice(10000, (args.k,1))
```

Another method will be discussed later. After centers are chosen, we can start performing clustering, that is the nested for loop in the picture above. As for calculate the distance between data and cluster centers, the formula I used is:

$$||\phi(X_n) - \phi(\mu_n)|| = k(x_n, x_n) + k(\mu_k, \mu_k) - 2k(x_n, \mu_k).$$

```
def distance(x, y, gram_matrix):
    return gram_matrix[x,x] + gram_matrix[y,y] -2* gram_matrix[x,y]
```

So far we can have the initial clusters. The rest is to do the iterative procedure.

```

def KernelKmeans(num_of_cluster, cluster, kernel, images):
    current_cluster = cluster.copy()
    for i in range(1, 100):
        print("Iteration #{}".format(i))
        new_cluster = np.zeros(num_of_points, dtype=int)

        _, cluster_size = np.unique(current_cluster, return_counts=True)

        k_pq = np.zeros(num_of_cluster)
        # if xn belongs to cluster k then akn = 1, otherwise 0
        for k in range(num_of_cluster):
            temp = kernel.copy()
            for j in range(num_of_points):
                if current_cluster[j] != k:
                    temp[j, :] = 0
                    temp[:, j] = 0
            k_pq[k] = np.sum(temp)

        # compute new clusters
        for j in range(num_of_points):
            min_dist = np.zeros(num_of_cluster)
            for k in range(num_of_cluster):
                k_jn = np.sum(kernel[j, :][np.where(current_cluster == k)])

                min_dist[k] = kernel[j, j] - 2/cluster_size[k] * \
                    k_jn + (k_pq[k]/cluster_size[k]**2)
            new_cluster[j] = np.argmin(min_dist)

        images.append(outputImage(new_cluster, "kmeans", i))
        if np.linalg.norm((new_cluster - current_cluster), ord=2) < 1e-3:
            break
        current_cluster = new_cluster.copy()

    createGIF(images, "kmeans", i)
    return

```

According to the formula in teacher's slide:

$$\begin{aligned}
 \left\| \phi(x_j) - \mu_k^\phi \right\| &= \left\| \phi(x_j) - \sum_{n=1}^N \alpha_{kn} \phi(x_n) \right\| \\
 &= \mathbf{k}(x_j, x_j) - \frac{2}{|C_k|} \sum_n \alpha_{kn} \mathbf{k}(x_j, x_n) + \frac{1}{|C_k|^2} \sum_p \sum_q \alpha_{kp} \alpha_{kq} \mathbf{k}(x_p, x_q)
 \end{aligned}$$

**Gram matrix!**

In this formula, there are three parts. So in the function I first calculate the last part of the formula because it's independent to the data, that is variable  $k_{pq}$  in the function. Then we can start performing a new round of clustering by the distance formula shown above to determine the data belonging to which cluster. After clustering, I capture the current clustering status and make an image for later GIF production.

```
def outputImage(cluster, img_type, num_of_img):
    colors = np.array([[255, 0, 0], [0, 255, 0], [0, 0, 255], [0, 215, 175], [
        95, 0, 135], [255, 255, 0], [255, 175, 0]])
    point_color = np.zeros((num_of_points, 3))
    for i in range(num_of_points):
        point_color[i, :] = colors[cluster[i], :]

    image = point_color.reshape((100, 100, 3))
    image = Image.fromarray(np.uint8(image))
    #image.save(os.path.join(args.output_image, img_type+str(num_of_img)+".png"))
    return image
```

This iterative procedure will continue until the clusters converge.  
After all the process, then output a GIF by the images we stored in the procedure:

```
def createGIF(images, gif_name, iter):
    images[0].save("output/kmeans.gif", save_all=True,
        append_images=images[1:], optimize=False, loop=0, duration=100)
    return
```

## II. Spectral Clustering

Same as K-means, we need to read images and compute kernel matrix. The process is identical to the images in K-means so I'll skip them here.

The rest procedure is according to the algorithm in slides,

Ratio cut:

### Unnormalized spectral clustering

Input: Similarity matrix  $S \in \mathbb{R}^{n \times n}$ , number  $k$  of clusters to construct.

- Construct a similarity graph by one of the ways described in Section 2. Let  $W$  be its weighted adjacency matrix.
- Compute the unnormalized Laplacian  $L$ .
- Compute the first  $k$  eigenvectors  $u_1, \dots, u_k$  of  $L$ .
- Let  $U \in \mathbb{R}^{n \times k}$  be the matrix containing the vectors  $u_1, \dots, u_k$  as columns.
- For  $i = 1, \dots, n$ , let  $y_i \in \mathbb{R}^k$  be the vector corresponding to the  $i$ -th row of  $U$ .
- Cluster the points  $(y_i)_{i=1, \dots, n}$  in  $\mathbb{R}^k$  with the  $k$ -means algorithm into clusters  $C_1, \dots, C_k$ .

Output: Clusters  $A_1, \dots, A_k$  with  $A_i = \{j \mid y_j \in C_i\}$ .

### Normalized cut:

#### Normalized spectral clustering according to Ng, Jordan, and Weiss (2002)

Input: Similarity matrix  $S \in \mathbb{R}^{n \times n}$ , number  $k$  of clusters to construct.

- Construct a similarity graph by one of the ways described in Section 2. Let  $W$  be its weighted adjacency matrix.
- Compute the normalized Laplacian  $L_{\text{sym}} D^{-1/2} L D^{-1/2}$
- Compute the first  $k$  eigenvectors  $u_1, \dots, u_k$  of  $L_{\text{sym}}$ .
- Let  $U \in \mathbb{R}^{n \times k}$  be the matrix containing the vectors  $u_1, \dots, u_k$  as columns.
- Form the matrix  $T \in \mathbb{R}^{n \times k}$  from  $U$  by normalizing the rows to norm 1, that is set  $t_{ij} = u_{ij} / (\sum_k u_{ik}^2)^{1/2}$ .

- For  $i = 1, \dots, n$ , let  $y_i \in \mathbb{R}^k$  be the vector corresponding to the  $i$ -th row of  $T$ .
- Cluster the points  $(y_i)_{i=1, \dots, n}$  with the  $k$ -means algorithm into clusters  $C_1, \dots, C_k$ .

Output: Clusters  $A_1, \dots, A_k$  with  $A_i = \{j \mid y_j \in C_i\}$ .

The overall process is like this:

```
def spectralClustering(kernel, num_of_cluster, cut_type, method):
    matrix_U = computeMatrixU(kernel, cut_type, num_of_cluster)
    U_centers = chooseCenters(matrix_U, num_of_cluster, method)
    clusters, images = kmeans(matrix_U, U_centers, num_of_cluster, cut_type)
    if num_of_cluster == 2:
        printEigen(matrix_U, clusters)
    return images
```

First function call is computeMatrixU:

```
def computeMatrixU(matrix_W, cut_type, num_of_cluster):
    """
    matrix_w: weight matrix
    """
    if cut_type == 1:
        cut = "ratio"
    else:
        cut = "normalized"
    # Graph Laplacian L and degree matrix D
    matrix_D, matrix_L = Laplacian(matrix_W)
    if cut_type == 2:
        matrix_D_sym = np.zeros((matrix_D.shape))
        for i in range(len(matrix_D)):
            matrix_D_sym[i,i] = 1.0 / np.sqrt(matrix_D[i,i])
        matrix_L = matrix_D_sym.dot(matrix_L).dot(matrix_D_sym)

    eigenvalues, eigenvectors = np.linalg.eig(matrix_L)
    # eigenvectors = eigenvectors.T
    # np.save(cut+"eigenvalues"+str(pic)+".npy", eigenvalues)
    # np.save(cut+"eigenvectors"+str(pic)+".npy", eigenvectors)
    eigenvalues = np.load(cut+"eigenvalues"+str(pic)+".npy")
    eigenvectors = np.load(cut+"eigenvectors"+str(pic)+".npy")

    # sort eigenvalues and find indices of nonzero eigenvalues
    sort_idx = np.argsort(eigenvalues)
    mask = eigenvalues[sort_idx] > 0
    idx = sort_idx[mask][0:num_of_cluster]
    matrix_U = eigenvectors[idx].T
    if cut_type == 2:
        T = matrix_U.copy()
        temp = np.sum(matrix_U, axis=1)
        for i in range(len(T)):
            T[i] /= temp[i]
        matrix_U = T
    # return eigenvectors[sort_idx[:num_of_cluster]].T
    return matrix_U
```

Accordingly, we need to compute Graph Laplacian by the formula:

define a special matrix:  $L = D - W$

**Graph Laplacian**

where W is our precomputed kernel matrix. D is the degree matrix.

```
def Laplacian(matrix_W):
    matrix_D = np.zeros_like(matrix_W)
    for idx, row in enumerate(matrix_W):
        matrix_D[idx, idx] += np.sum(row)
    matrix_L = matrix_D - matrix_W
    return matrix_D, matrix_L
```

After getting matrix\_L (Laplacian), if normalized cut, we need to normalize the matrix\_L. The formula is:

$$L_{\text{sym}} := D^{-1/2} L D^{-1/2}$$

After that, we calculate the eigenvalues and eigenvectors of matrix\_L or normalized matrix\_L. Then sort eigenvalues and select first k corresponding eigenvectors, make them matrix\_U. If normalized cut, we then need to perform normalizing on matrix\_U to make row normalized to norm 1.

We can start performing k-means now, but we need to choose initial centers. Also there are two ways, default is random method.

```
def chooseCenters(matrix_U, num_of_cluster, method):
    if 0 == method:
        return matrix_U[np.random.choice(10000, args.k)]
```

Then we can start the iterative procedure of k-means until converge.

```
def kmeans(matrix_U, U_centers, num_of_cluster, cut_type):
    images = []
    old_centers = U_centers.copy()
    new_clusters = np.zeros(num_of_points, dtype=int)
    for i in range(100):
        print("Iteration #{}".format(i))
        new_clusters = kmeansClustering(matrix_U, old_centers, num_of_cluster)
        new_centers = kmeansCenters(matrix_U, new_clusters, num_of_cluster)
        image = outputImage(new_clusters, "spectral_clustering", i)
        images.append(image)
        if np.linalg.norm((new_centers - old_centers), ord=2) < 1e-2:
            break
        old_centers = new_centers.copy()
    return new_clusters, images
```

K-means procedure is to compute the distance between data and center of cluster, then assign the data to the nearest cluster:

```
def kmeansClustering(matrix_U, U_centers, num_of_cluster):
    new_clusters = np.zeros(num_of_points, dtype=int)
    for i in range(num_of_points):
        dist = np.zeros(num_of_cluster)
        for j, cen in enumerate(U_centers):
            dist[j] = np.linalg.norm((matrix_U[i]-cen), ord=2)
        new_clusters[i] = np.argmin(dist)
    return new_clusters
```

After all datas are assigned, compute new centers of clusters:

```
def kmeansCenters(matrix_U, new_clusters, num_of_cluster):
    """
    Recompute new centers
    """
    new_centers = []
    for i in range(num_of_cluster):
        points_in_c = matrix_U[new_clusters == i]
        new_center = np.average(points_in_c, axis=0)
        new_centers.append(new_center)
    return np.array(new_centers)
```

Also, I capture the status of clusters and make them an image, after converging, produce a GIF from those images. The process is identical to k-means, so I'll also skip here.

Part2.

In the argument, I can specify different k values.

I.II. K-means and Spectral Clustering

```
parser.add_argument("--k", type=int, default=3)
```

Part3.

When choosing initial centers of clusters, there are two ways to do so. The first one is randomly choose, which is already shown in the content above. Another way is called k-means++. The idea is that the further data point from current centers(start from 1 center) have higher chances to be selected as center of another cluster.

The process is as below:

## I. K-means:

```
centers = []
centers = list(random.sample(range(0, 10000), 1))
for number_center in range(1, args.k):
    min_dist = np.full(num_of_points, np.inf)
    for i in range(num_of_points):
        for j in range(number_center):
            #dist = distance(i, centers[j], kernel)
            dist = np.linalg.norm(i - centers[j])
            if dist < min_dist[i]:
                min_dist[i] = dist
    min_dist /= np.sum(min_dist)
    centers.append(np.random.choice(
        np.arange(10000), 1, p=min_dist)[0])
return centers
```

## II. Spectral clustering:

```
centers = []
centers = list(random.sample(range(0, 10000), 1))
for number_center in range(1, args.k):
    min_dist = np.full(num_of_points, np.inf)
    for i in range(num_of_points):
        for j in range(number_center):
            dist = np.linalg.norm(i - centers[j])
            if dist < min_dist[i]:
                min_dist[i] = dist
    min_dist /= np.sum(min_dist)
    centers.append(np.random.choice(
        np.arange(10000), 1, p=min_dist)[0])
U_centers = []
for i in range(num_of_cluster):
    U_centers.append(matrix_U[centers[i]])
U_centers = np.array(U_centers)
return U_centers
```

## Part4.

### II. Spectral Clustering

I implemented a function called printEigen to print the eigenvectors coordinate. The coordinate in the same cluster will have same color:



```

def printEigen(matrix_U, clusters):
    if args.cut_type == 1:
        cut = "ratio"
    else:
        cut = "normalized"
    if args.mode == 1:
        mode = "random"
    else:
        mode = "kmeans++"

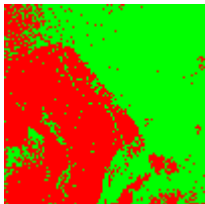

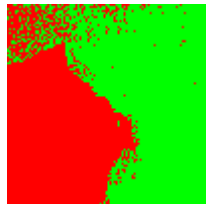
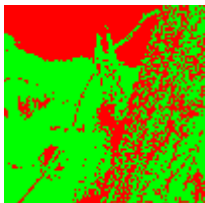
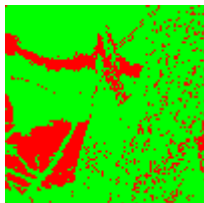
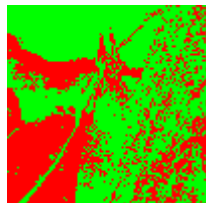
    matrix_U = np.real(matrix_U)
    colors = ["r", "g", "b"]
    plt.clf()
    for idx in range(len(matrix_U)):
        plt.scatter(matrix_U[idx, 0], matrix_U[idx, 1],
                    color=colors[clusters[idx]])
        if idx % 1000 == 0:
            print("hihi")
    plt.savefig(str(pic)+"_eigen_coordinate_"+str(cut)+"_"+str(mode)+".png")
    return

```

## b. Experiments setting and results

Part1.

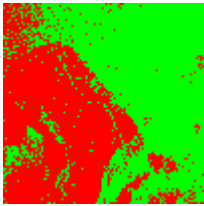
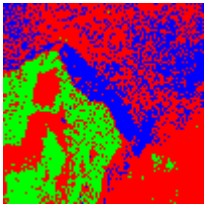
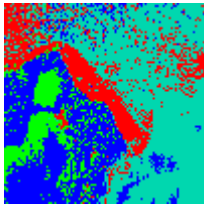
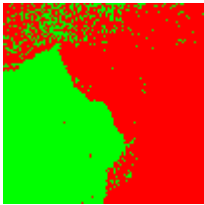
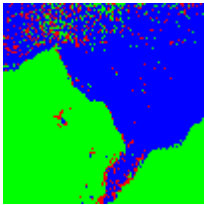
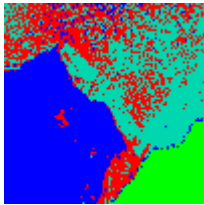
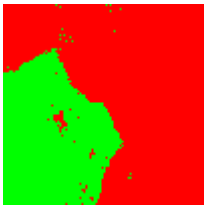
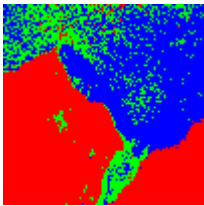
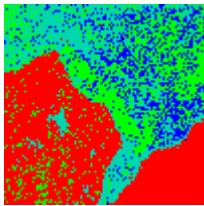
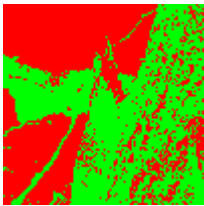
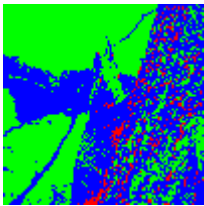
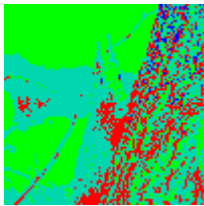
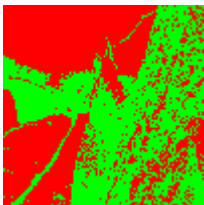
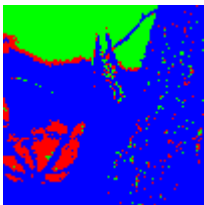
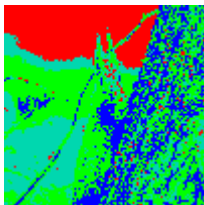
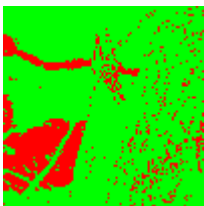
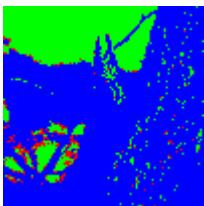
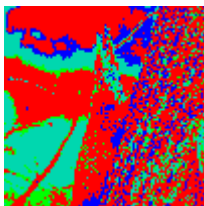
K = 2, random choose center method

	Kernel K means	Spectral Clustering- Ratio Cut	Spectral Clustering- Normalized Cut
Image 1			
Image 2			

Part2.

K = 2, 3, 4

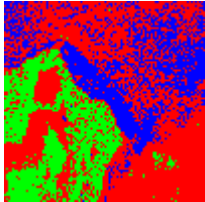
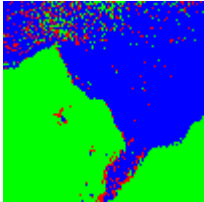
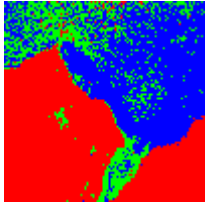
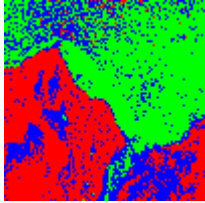
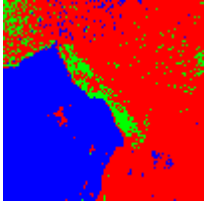
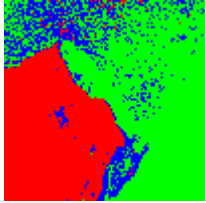
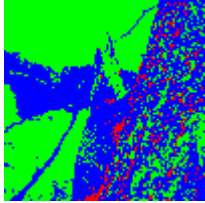
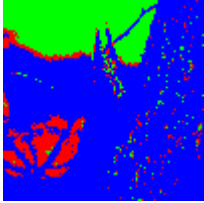
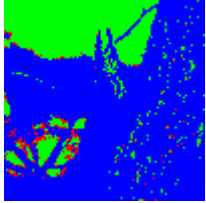
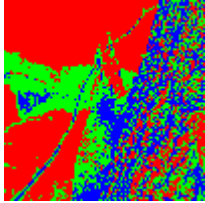
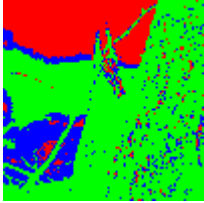
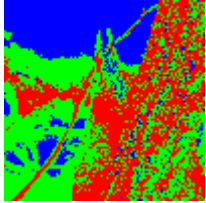
random choose center method

K	2	3	4
Kernel K means image1			
Spectral Clustering- Ratio Cut image1			
Spectral Clustering- Normalized Cut image1			
Kernel K means image 2			
Spectral Clustering-Ratio Cut image2			
Spectral Clustering- Normalized Cut image2			

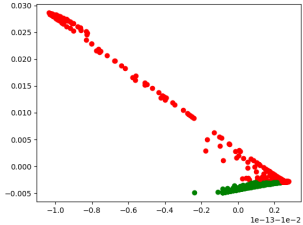
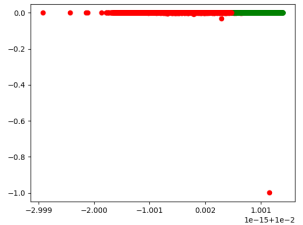
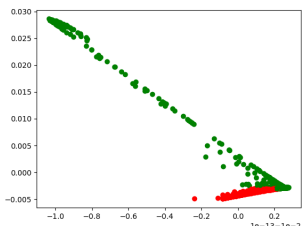
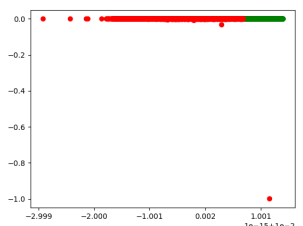
Part3.

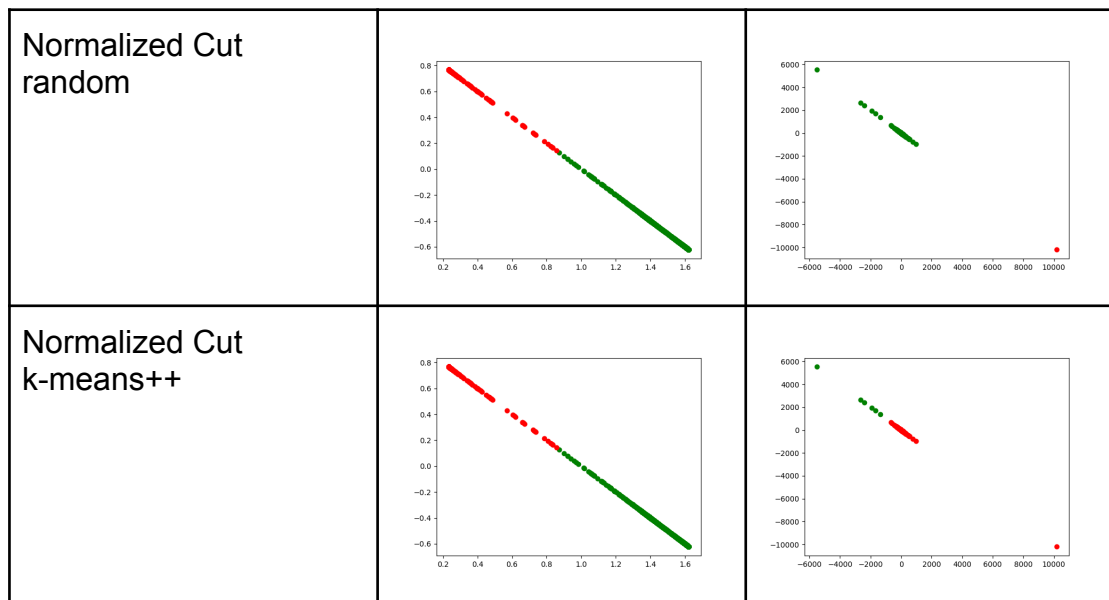
K = 3

	Kernel K means	Spectral Clustering- Ratio Cut	Spectral Clustering- Normalized Cut
--	----------------	--------------------------------------	---

random image1			
k-means++ image1			
random image2			
k-means++ image2			

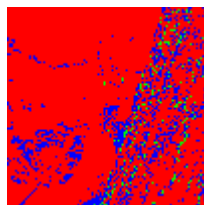
Part4.  
K=2

	image1	image2
Ratio Cut random		
Ratio Cut k-means++		



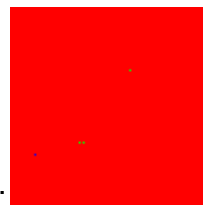
### c. Observations and Discussions

- We can see the results are not bad. The shape of land in image 1, and the rough shape of rabbit in image2 can be observed from the results.
- The bottom right corner of image1 sometimes will be classified as the same cluster of the land, due to the color is deeper than other part of the sea.
- Random method needs more iteration to converge than the k-means++ method.
- The proper value of  $k$  (the number of clusters) differs from different pictures, which may have different numbers of color blocks.
- When  $k > 5$ , some color dots in image1 may only appear at very few locations.
- The average number of iterations of spectral clustering is less than kernel k-means, but calculating eigenvectors and eigenvalues takes lots of time, so the total cost time is higher.
- Different initial choosing methods will affect initial result, but after iterations, the difference will narrow.
- Sometimes, after iterations, the results seem to be worse than initial.
- The normalized cut seems to have worse performance than ratio cut. All data will eventually be classified into the same cluster.



E.g. iter = 1:

iter = 19:



- Data points in the same cluster have the same coordinate in the eigenspace of Graph Laplacian.