ML Homework 6 Report

I. Code with detail explanations

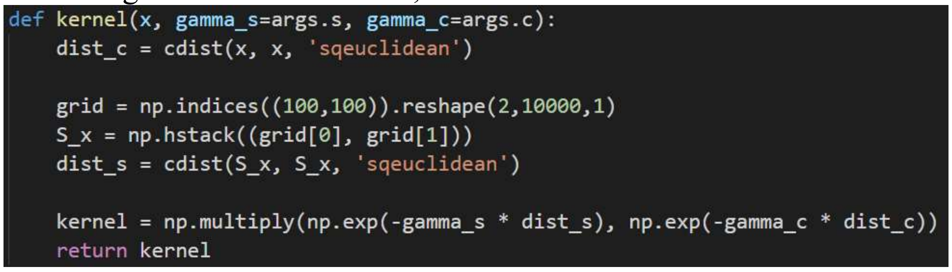
1. Kernel K-means

- Kernel

  Use the new kernel defined below:

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

where S(x) is the spatial information of data x , and C(x) is the color information of data x. Both γs and γc are hyper-parameters.
According to the formula above, we define the kernel:



```python
def kernel(x, gamma_s=args.s, gamma_c=args.c):
    dist_c = cdist(x, x, 'sqeuclidean')

    grid = np.indices((100,100)).reshape(2,10000,1)
    S_x = np.hstack((grid[0], grid[1]))
    dist_s = cdist(S_x, S_x, 'sqeuclidean')

    kernel = np.multiply(np.exp(-gamma_s * dist_s), np.exp(-gamma_c * dist_c))
    return kernel
```

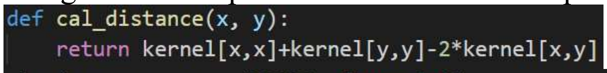
Which sqeuclidean is the Squared Euclidean distance.

- Initial clustering

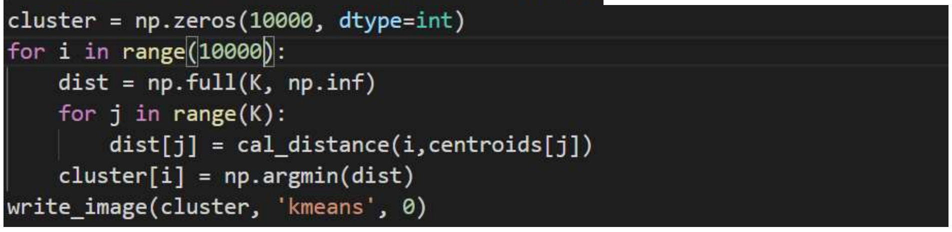
  Use the distance defined below:

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

Assign each data point to the closest center point.



```python
def cal_distance(x, y):
    return kernel[x,x]+kernel[y,y]-2*kernel[x,y]
```



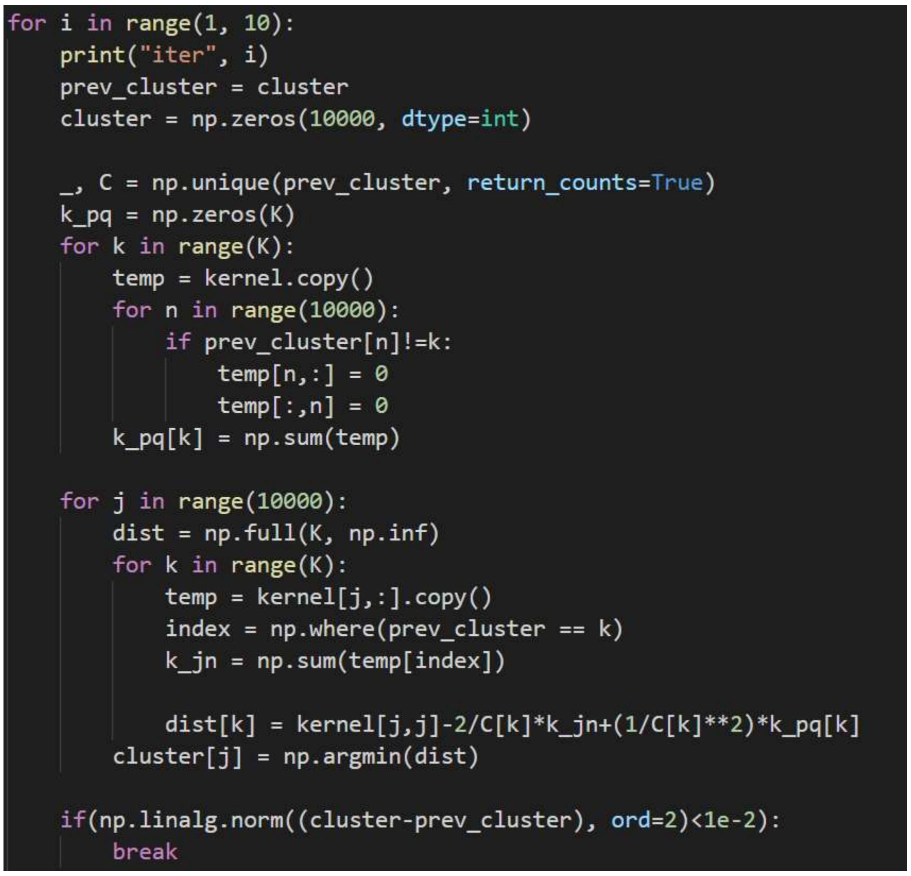
```python
cluster = np.zeros(10000, dtype=int)
for i in range(10000):
    dist = np.full(K, np.inf)
    for j in range(K):
        dist[j] = cal_distance(i,centroids[j])
    cluster[i] = np.argmin(dist)
write_image(cluster, 'kmeans', 0)
```

- Iterative Cluster Procedure

  Use the function defined below as distance:

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

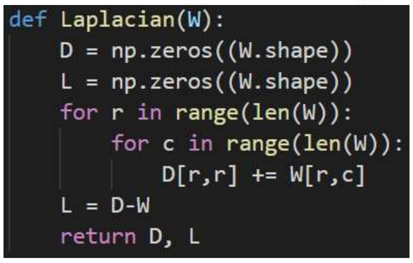
Assign to different cluster according to the minimum distance.
Recursively until the clustering result converges.



```python
for i in range(1, 10):
    print("iter", i)
    prev_cluster = cluster
    cluster = np.zeros(10000, dtype=int)

    _, C = np.unique(prev_cluster, return_counts=True)
    k_pq = np.zeros(K)
    for k in range(K):
        temp = kernel.copy()
        for n in range(10000):
            if prev_cluster[n]!=k:
                temp[n,:] = 0
                temp[:,n] = 0
        k_pq[k] = np.sum(temp)

    for j in range(10000):
        dist = np.full(K, np.inf)
        for k in range(K):
            temp = kernel[j,:].copy()
            index = np.where(prev_cluster == k)
            k_jn = np.sum(temp[index])

            dist[k] = kernel[j,j]-2/C[k]*k_jn+(1/C[k]**2)*k_pq[k]
        cluster[j] = np.argmin(dist)

    if(np.linalg.norm((cluster-prev_cluster), ord=2)<1e-2):
        break
```

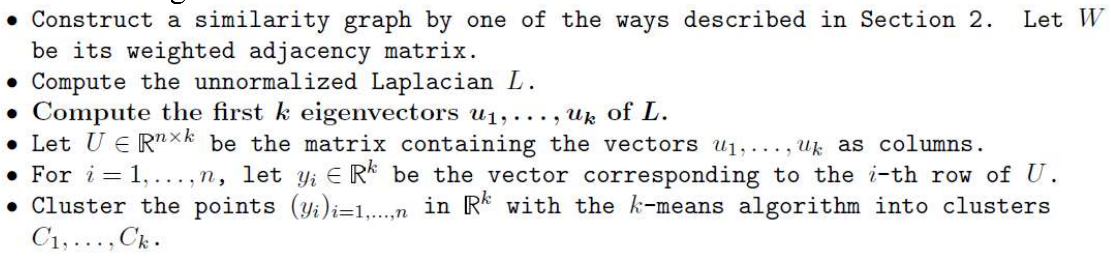
2. Spectral Clustering
   - Compute Laplacian
     First calculate the D(degree matrix of W), then calculate the Laplacian matrix according to the following formula:

$$L = D - A$$

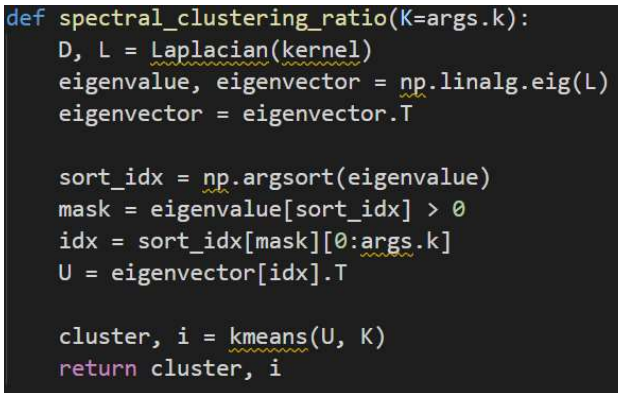


```python
def Laplacian(W):
    D = np.zeros((W.shape))
    L = np.zeros((W.shape))
    for r in range(len(W)):
        for c in range(len(W)):
            D[r,r] += W[r,c]
    L = D-W
    return D, L
```

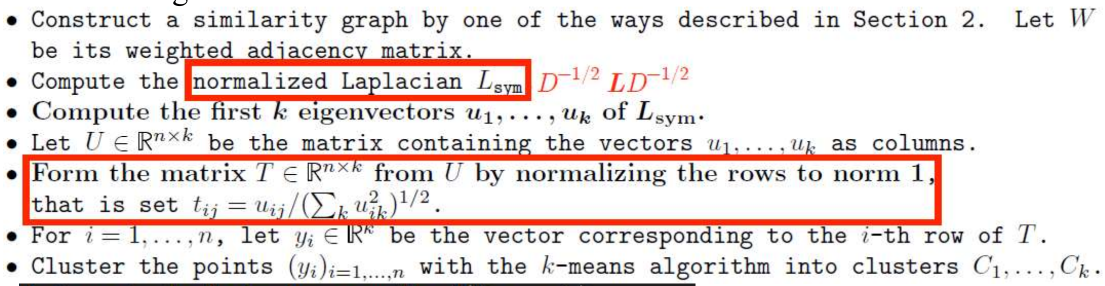
   - Ratio cut
     Follow the algorithm below:



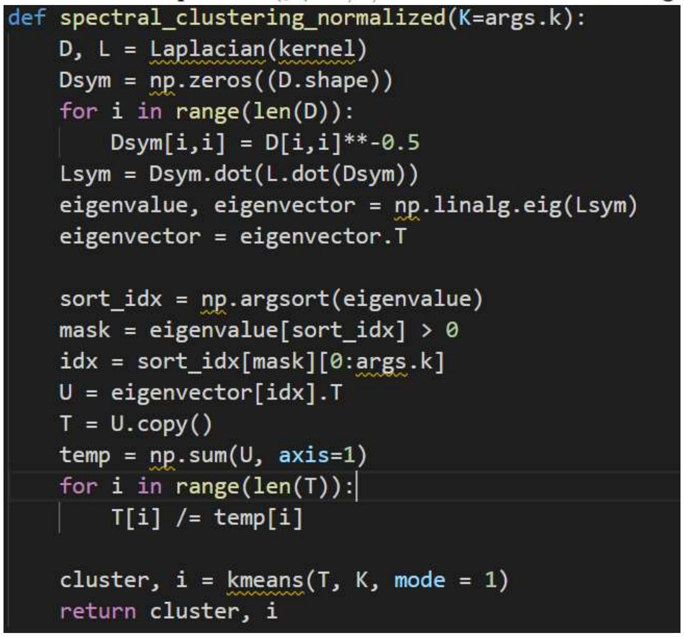
- Construct a similarity graph by one of the ways described in Section 2. Let W be its weighted adjacency matrix.
- Compute the unnormalized Laplacian L.
- **Compute the first k eigenvectors $u_1, \dots, u_k$ of L.**
- Let $U \in \mathbb{R}^{n \times k}$ be the matrix containing the vectors $u_1, \dots, u_k$ 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 $C_1, \dots, C_k$.


```python
def spectral_clustering_ratio(K=args.k):
    D, L = Laplacian(kernel)
    eigenvalue, eigenvector = np.linalg.eig(L)
    eigenvector = eigenvector.T

    sort_idx = np.argsort(eigenvalue)
    mask = eigenvalue[sort_idx] > 0
    idx = sort_idx[mask][0:args.k]
    U = eigenvector[idx].T

    cluster, i = kmeans(U, K)
    return cluster, i
```

- Normalized cut
  Follow the algorithm below:



- Construct a similarity graph by one of the ways described in Section 2. Let W be its weighted adjacency matrix.
- Compute the $\boxed{\text{normalized Laplacian } L_{\text{sym}}}$ $D^{-1/2} LD^{-1/2}$
- Compute the first k eigenvectors $u_1, \dots, u_k$ of L_{sym}.
- Let $U \in \mathbb{R}^{n \times k}$ be the matrix containing the vectors $u_1, \dots, u_k$ as columns.
- $\boxed{\text{Form the matrix } T \in \mathbb{R}^{n\times k} \text{ from } U \text{ by normalizing the rows to norm 1, that is set } t_{ij} = u_{ij}/(\sum_k u_{ik}^2)^{1/2}.}$
- 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 $C_1, \dots, C_k$.


```python
def spectral_clustering_normalized(K=args.k):
    D, L = Laplacian(kernel)
    Dsym = np.zeros((D.shape))
    for i in range(len(D)):
        Dsym[i,i] = D[i,i]**-0.5
    Lsym = Dsym.dot(L.dot(Dsym))
    eigenvalue, eigenvector = np.linalg.eig(Lsym)
    eigenvector = eigenvector.T

    sort_idx = np.argsort(eigenvalue)
    mask = eigenvalue[sort_idx] > 0
    idx = sort_idx[mask][0:args.k]
    U = eigenvector[idx].T
    T = U.copy()
    temp = np.sum(U, axis=1)
    for i in range(len(T)):
        T[i] /= temp[i]

    cluster, i = kmeans(T, K, mode = 1)
    return cluster, i
```

- K-means
  New centers is equal to average point of each clusters. Assign to different cluster according to the minimum distance. Recursively until the clustering result converges.

```python
for i in range(1, 10):
    print("iter ", i)
    prev_centers = centers
    centers = []
    for j in range(k):
        mask = cluster==j
        centers.append(np.sum(data[mask], axis=0) / len(data[mask]))
    centers = np.array(centers)
    if(np.linalg.norm((centers-prev_centers), ord=2)<1e-2):
        break

    cluster = clustering(data, centers, k)
    write_image(cluster, title, i)
return cluster, i
```

3. Part 2

Try more clusters with different value of argument parser k.

```python
parser.add_argument('--k', type=int, default=2)
```

4. Part 3: try different initial way

The simplest way: random pick central points.

```python
centroids = list(random.sample(range(0,10000), k))
```

I also use the K-means++ algorithm to select the initial center point of the cluster, which improves that the random selection of the initial center point may cause the clustering results to be very different from the actual distribution of the data.
The basic idea of the K-means++ algorithm is that the mutual distance between the initial cluster centers should be adjusted far.
The specific initial selection process is as follows: Randomly select a point from the input data point as the first center; for each point x in the data set, calculate the distance D(x) between it and the nearest center, and take D(x) as the weight to take the next center . Repeat until k centers are obtained and stop.

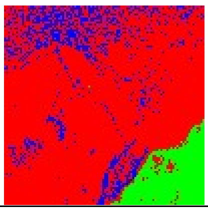
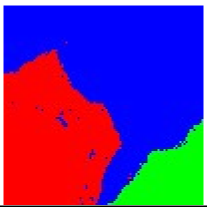
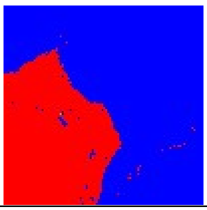
```python
centroids = []
centroids = list(random.sample(range(0,10000), 1))
for number_center in range(1, K):
    min_dist = np.full(10000, np.inf)
    for i in range(10000):
        for j in range(number_center):
            dist = cal_distance(i, centroids[j])
            if dist < min_dist[i]:
                min_dist[i] = dist
    min_dist /= np.sum(min_dist)
    centroids.append(np.random.choice(np.arange(10000), 1, p=min_dist)[0])
```

5. Part 4: plot eigenspace cluster

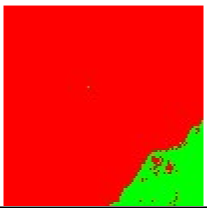
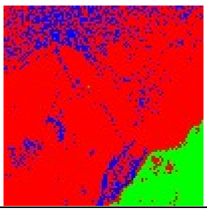
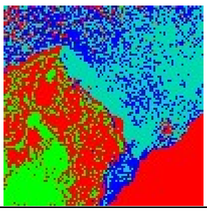
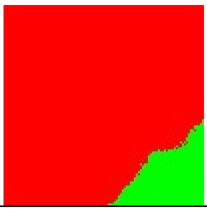
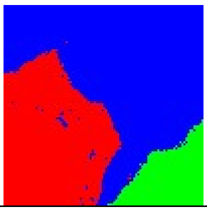
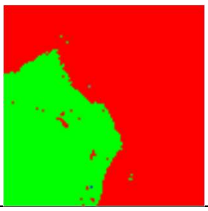
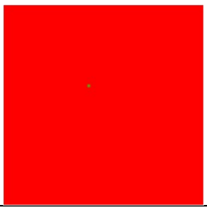
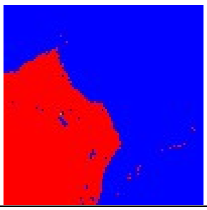
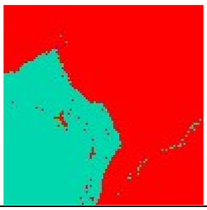
```python
def clustering(U, centers, k):
    cluster = np.zeros(10000, dtype=int)
    for i in range(10000):
        dist = np.full(k, np.inf)
        for j in range(k):
            dist[j] = cdist([U[i]], [centers[j]], 'sqeuclidean')
        cluster[i] = np.argmin(dist)
    return cluster
```

II. Experiments settings and results

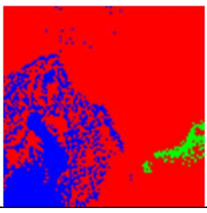
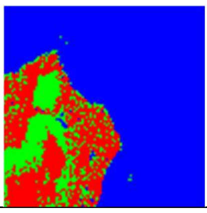
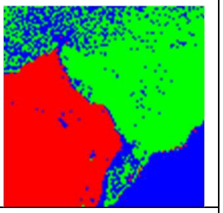
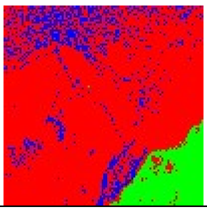
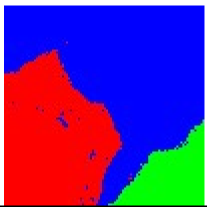
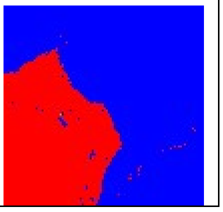
1. Part 1 (K=3)

Kernel-k-means	Ratio cut	Normalized cut
		

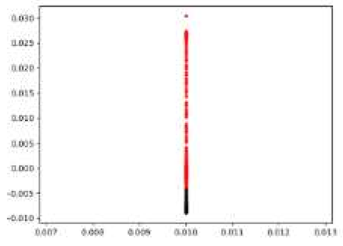
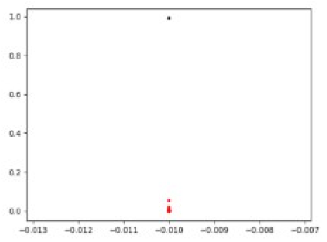
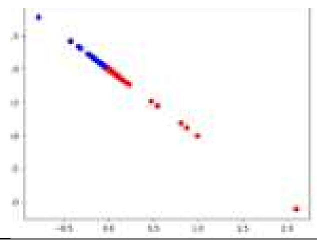
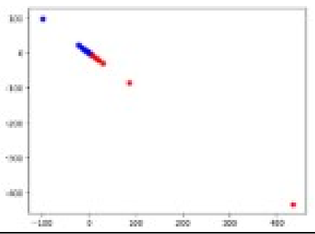
2. Part 2

K	2	3	4
Kernel-k-means			
Ratio cut			
Normalized cut			

3. Part 3

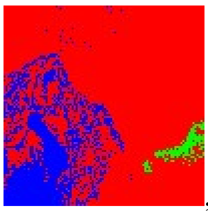
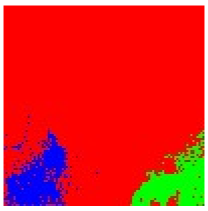
k-means++	Kernel-k-means	Ratio cut	Normalized cut
random			
Kmeans++			

4. Part 4

	Image1	Image2
Ratio cut		
Normalized cut		

III. Observations and Discussions

- From the results of grouping, we can see the junction of land and sea.
- From the results of part1, Ratio has the best clustering.
- Although k-means will converge with the number of iterations, the result may be worse than before. As can be seen from the two pictures below, although the results of grouping are getting more and more clustered, the outlines disappear.

Iteration = 0 , Iteration = 5 

- Generally speaking, the number of iterations of Spectral Clustering will be much less than that of Kernel K-means. But because of the large amount of calculation of eigenvector, it will take a lot more time than k-means.
- The clustering results can get better results when k = 3 or 4 on image1. When k>4, there may be only one or two dots with a few colors on the graph.
- The appropriate size of the k value depends on different pictures. When the picture contains more color blocks, the k value cannot be too small.
- Different initial clustering methods will greatly affect the initial results, but after iteration, the gap between random and k-means++

will narrow.

- Compared with k-means++, random requires more iterations to converge.
- Data points within the same cluster have the same coordinates in the eigenspace of graph Laplacian