

a. code with detailed explanations

- Part1
 - Load data & calculate gram matrix

```
1 img = cv2.imread('ML_HW06/image2.png')
2 all_point = []
3 for i in range(100):
4     for j in range(100):
5         each_point = np.append(img[i][j], [i, j])
6         all_point.append(each_point)
7 all_point = np.array(all_point)

1 def cal_gram(data):
2
3
4     tmpC = pdist(data[:, :3])
5     tmpC = squareform(tmpC)
6
7     tmpS = pdist(data[:, 3:])
8     tmpS = squareform(tmpS)
9
10    cal_gram = np.exp(-1/100000*tmpC**2) * np.exp(-1/100000*tmpS**2)
11
12    return cal_gram
```

Use cv2 package to load image, and put the corresponding coordinate in the array, which name is “all_point”. The first three number in each row mean this data point’s colors (RGB values), and the last two numbers mean this data point’s coordinate.

Use “cal_gram” function to calculate gram matrix. In our experiment, we set gamma_s and gamma_c equal to 1/100000.

- spectral clustering

```
1 def spec_pre(n_cut):
2
3     global global_U
4
5     K = 2
6
7     old_gram = cal_gram(all_point)
8     W = np.copy(old_gram)
9     tmp_d = np.sum(W, axis=0)
10    D = np.zeros((10000, 10000))
11    np.fill_diagonal(D, tmp_d)
12    L = np.array([])
13    if n_cut == 0:
14        L = np.subtract(D, W)
15    elif n_cut == 1:
16        sqrtD = np.sqrt(D)
17        L = np.linalg.inv(sqrtD) @ np.subtract(D, W) @ np.linalg.inv(sqrtD)
18    e_value, e_vector = np.linalg.eigh(L)
19    U = np.copy(e_vector[:, :K])
20
21
22
23    if n_cut == 1:
24        norm = np.linalg.norm(U, axis=1)
25        for i in range(len(U)):
26            U[i] = U[i]/norm[i]
27
28    global_U = np.copy(U)
29    print(U)
30
31    tmp_dis = pdist(U, metric='euclidean')
32    new_gram = squareform(tmp_dis)
33    return new_gram
```

In spectral clustering, we need to build similarity graph and mapping to eigenspace of graph Laplacian. The input parameter “n_cut” is a bool variable (n_cut==0 mean ratio cut, n_cut==1 mean normalized cut). In the code showed above, variable L represents Laplacian. Note that in ratio cut $L = D - W$, and in normalized cut, Normalized Laplacian = $D^{-1/2} L D^{-1/2}$. U is an 10000*K matrix which include the first K eigen vector of Normalized Laplacian.

Finally, the “spec_pre” function will return a variable “new_gram”, which will be used in the k-mean.

- kernel k-means

```

1 def kernel_kmeans(GRAM, num_iter, plusplus=0):
2
3     K = 2
4
5     all_result = []
6     cluster = np.zeros((K,10000))
7     new_Cluster = np.zeros((K,10000))
8
9     if plusplus == 0:
10         for i in range(10000):
11             if i % 100 > 100/K-1:
12                 cluster[1][i] = 1
13             else:
14                 cluster[0][i] = 1
15     else:
16         center = np.zeros(K)
17         center[0] = np.random.randint(10000)
18         p = np.zeros(10000)
19
20         for i in range(len(p)):
21             p[i] = GRAM[int(center[0])][i]
22         p_sum = np.sum(p)
23
24         rand_num = np.random.rand() * p_sum
25
26         for i in range(10000):
27             if rand_num < p[i]:
28                 center[1] = i
29                 break
30             rand_num -= p[i]
31         for i in range(10000):
32             if GRAM[int(center[0])][i] > GRAM[int(center[1])][i]:
33                 cluster[0][i] = 1
34             else:
35                 cluster[1][i] = 1
36
37

```

```

46     for e in range(num_iter):
47
48         print(e)
49
50         clusterCount = np.zeros(K)
51         for i in range(K):
52             clusterCount[i] = np.sum(cluster[i])
53
54         print(clusterCount)
55
56         right_part = np.zeros(K)
57         for c in range(K):
58
59             row_c = np.array([cluster[c]])
60             index_array = row_c.T @ row_c
61             for i in range(10000):
62                 for j in range(10000):
63                     if index_array[i][j] == 1:
64                         right_part[c] += GRAM[i][j]
65
66             right_part = right_part/(clusterCount[c]**2)
67
68
69         for pixel_index in range(10000):
70
71             middle_part = 0
72             row_GRAM = np.array([GRAM[pixel_index]])
73             middle_part = row_c.T @ row_GRAM.T
74             middle_part *= -2 / clusterCount[c]
75
76             if 1 + middle_part + right_part[c] < pixel_min[pixel_index]:
77                 pixel_min[pixel_index] = 1 + middle_part + right_part[c]
78
79             for i in range(K):
80                 new_Cluster[i][pixel_index] = 0
81
82                 new_Cluster[c][pixel_index]=1
83
84         cluster = np.copy(new_Cluster)
85         all_result.append(cluster)
86
87     return all_result
88

```

The variable “cluster” store the cluster of every data point (cluster[k][i] == 1 if data i belongs to cluster k). We use two ways to initial cluster. The first way is just set the data point which is in the left part of the whole image belongs to one cluster, and the data point in the right part of image belongs to another cluster. The second way to initial cluster is k-mean++, which I will introduce later in Part3.

The code in right hand side show how we minimized the objective function:

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

The variable “right_part” represents the right part of above function. The variable “middle_part” represents the middle part of above function. After calculate each middle part, we will check whether the objective function will be smaller if we assign the new cluster to the data point (line 76). If yes, we set data point belongs to new cluster.

- Whole clustering process

```

1  def Clustering(k_mean=0, n_cut = 1, kpp = 0):
2
3      GRAM = np.zeros((10000,10000))
4
5      if k_mean==0:
6          GRAM = cal_gram(all_point)
7          num_iter = 5
8      else:
9          GRAM = spec_pre(n_cut)
10         num_iter = 2
11         if kpp == 1:
12             num_iter-=1
13
14     all_result = kernel_kmeans(GRAM, num_iter, kpp)
15     for i in range(num_iter+1):
16         draw_img(all_result, i)
17     compose_gif(num_iter+1)
18
19     return all_result

```

The whole clustering process are shown above. There are three parameters we will in put:

k-mean: 0 when k-mean, 1 when spectral clustering.

n_cut: 0 when ratio cut, 1 when normalized cut.

kpp: 0 when we use our own initialize method, 1 when use k-mean++.

- Part3

In this part, I implement k-mean++, which is used in both k-means clustering and spectral clustering, and compare the result with using original method.

```
center = np.zeros(K)
center[0] = np.random.randint(10000)
p = np.zeros(10000)

for i in range(len(p)):
    p[i] = GRAM[int(center[0])][i]
p_sum = np.sum(p)

rand_num = np.random.rand() * p_sum

for i in range(10000):
    if rand_num < p[i]:
        center[1] = i
        break
    rand_num -= p[i]
for i in range(10000):
    if GRAM[int(center[0])][i] > GRAM[int(center[1])][i]:
        cluster[0][i] = 1
    else:
        cluster[1][i] = 1
```

In k-mean++, I first random pick a center, and then pick another center base on the “distance” between the first center (the far the distance is, the more possible the point will be chosen to be the second center). After that, we decide all data point belongs to which cluster by which cluster center is closer to each data point. That’s how we initialize all data point cluster.

- Part4

The matrix U (a 10000*K matrix) which we had calculated in function “spec_pre” are the coordinates in the eigenspace of graph Laplacian. So we visualize these 10000 data points and see if the data points which we consider in the same cluster have the same (or near) coordinate.

```
1 cluster_1_x = np.array([])
2 cluster_1_y = np.array([])
3 cluster_2_x = np.array([])
4 cluster_2_y = np.array([])
5 for i in range(10000):
6     if all_result[2][0][i] == 1:
7         cluster_1_x = np.append(cluster_1_x, global_U[i][0])
8         cluster_1_y = np.append(cluster_1_y, global_U[i][1])
9     else:
10        cluster_2_x = np.append(cluster_2_x, global_U[i][0])
11        cluster_2_y = np.append(cluster_2_y, global_U[i][1])

1 plt.scatter(cluster_1_x, cluster_1_y,color='red')
2 plt.scatter(cluster_2_x, cluster_2_y,color='blue')
3 plt.show()
```

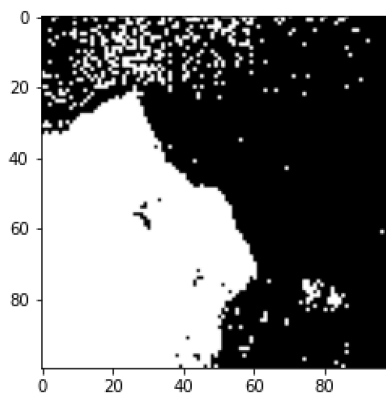
In above code, cluster_1_x, cluster_1_y, cluster_2_x, cluster_2_y represents the x and y coordinates in two cluster.

b. experiments settings and results & discussion

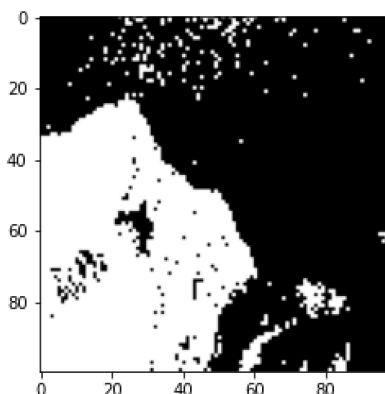
- Part1
 - Setting & Results:

In k-mean clustering, we train our model 5 iteration, while we train spectral clustering 3 iteration (both ratio cut and normalize cut).

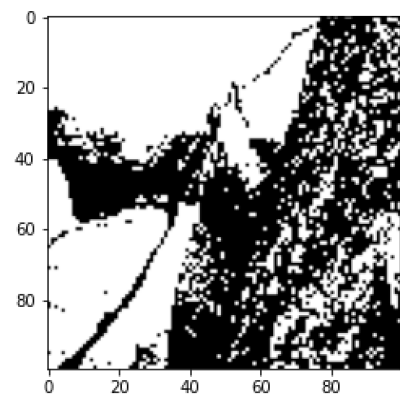
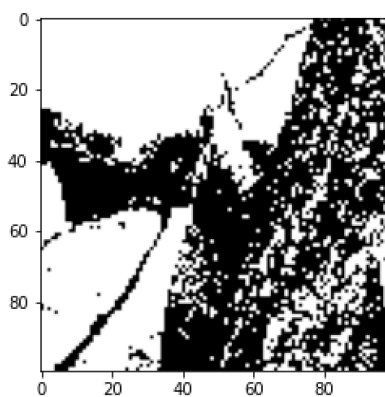
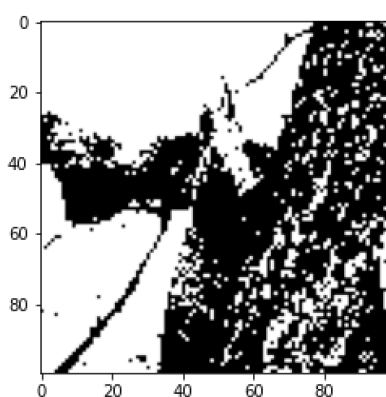
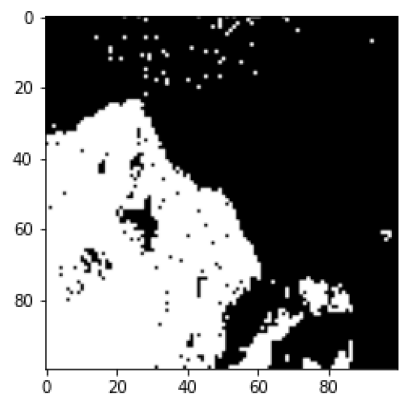
k-mean



spectral clustering
(ratio cut)



spectral clustering
(normalize cut)



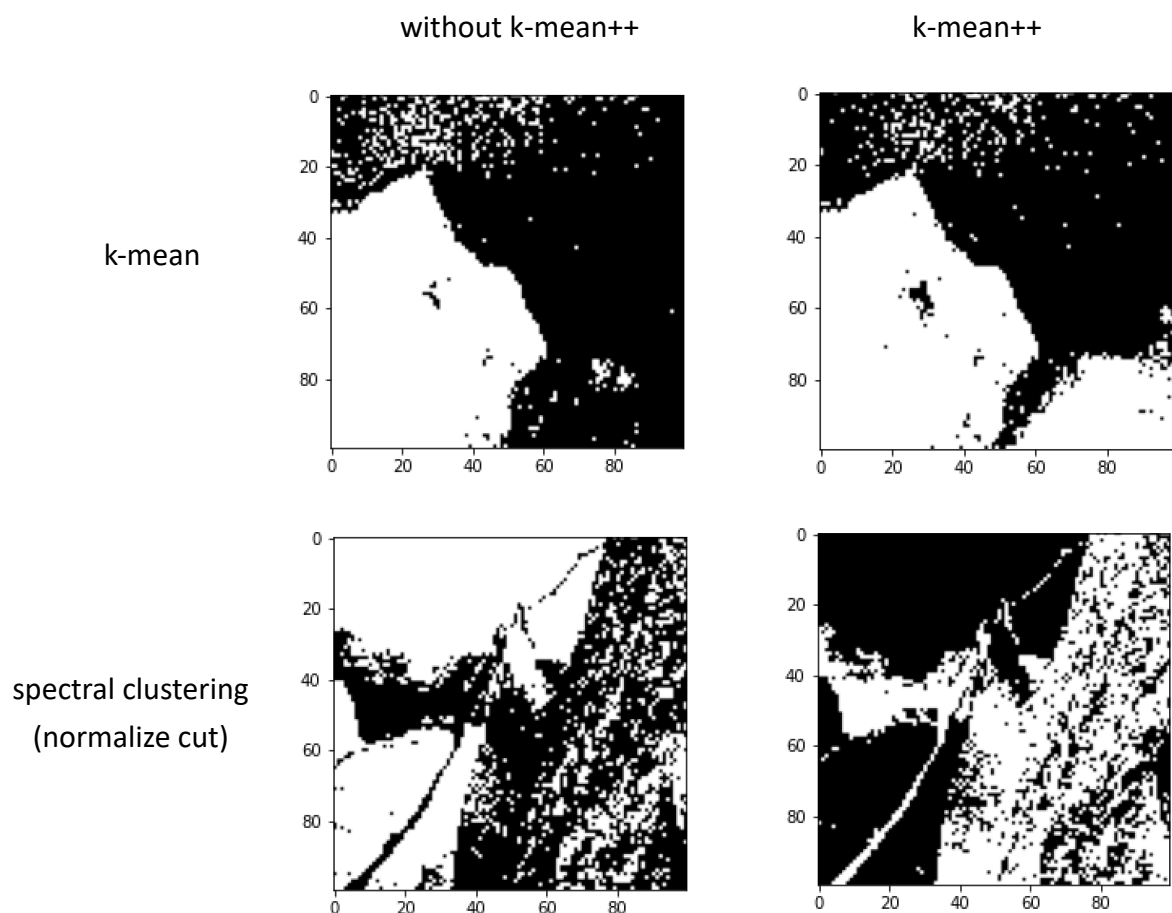
- Discussion:

According to the results, there is no specific different between those clustering methods. Specially in image2, three results are almost same. The performance of the results is spectral clustering (normalize cut) a little bit better than spectral clustering (ratio cut) and better than k-mean. However, if we consider the training time, spectral clustering may be the better clustering method, because it spends less time than k-mean. (3 iteration vs 5 iteration)

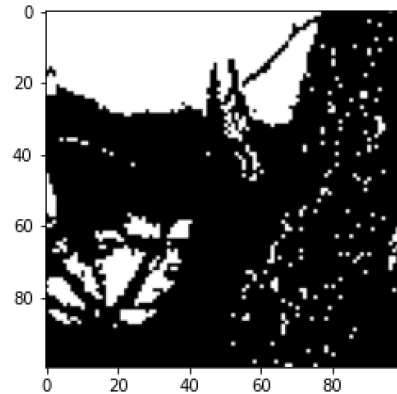
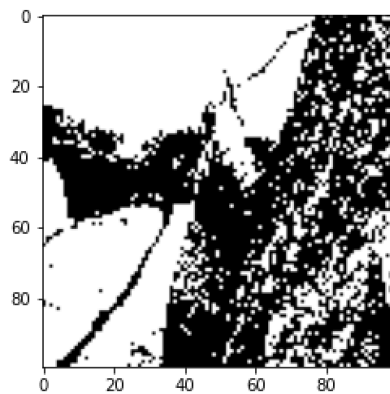
- Part3

- Setting & Results:

In this part, we use k-mean++ to initialize the all-data points' cluster, and then compare the results with our own initialize method (the left-part of image is considered cluster1, the right-part of image is considered cluster2). We train 5 iteration of k-mean and 2 iteration of spectral clustering.



spectral clustering
(ratio cut)



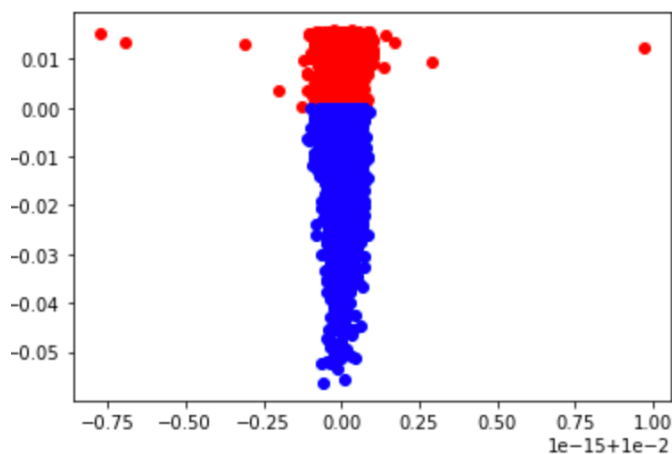
○ Discussion:

According to the results shown above, we can get equal or even better results if we use k-mean++. Also, if we check GIF, we could find that the clusters of all data points slightly change after iteration 2. Moreover, in spectral clustering, the iteration is lower than training without k-mean++. That's mean k-mean++ can make our model converge easier and decrease our training time.

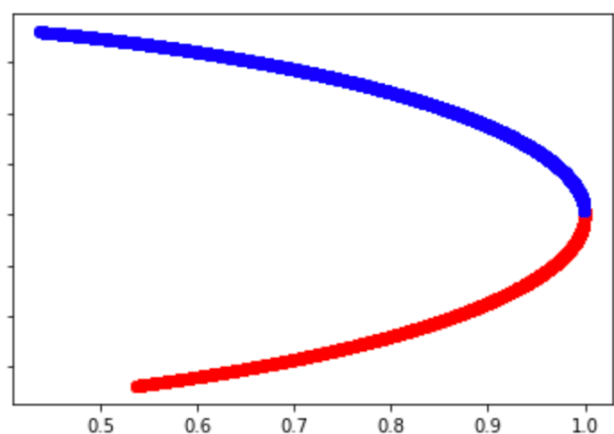
• Part4

We use the code mentioned before to draw a scatter plot of the coordinates in the eigenspace of graph Laplacian, and check whether the same cluster data points have same or near coordinate.

Ratio cut



Normalized cut

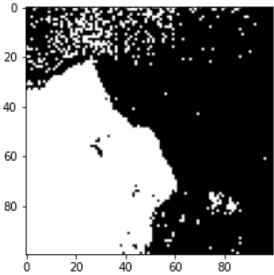
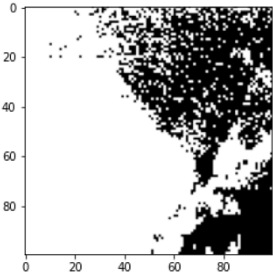
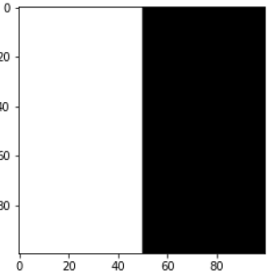


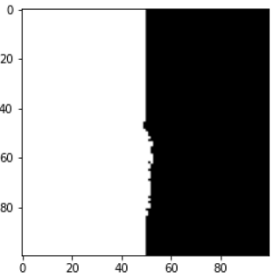
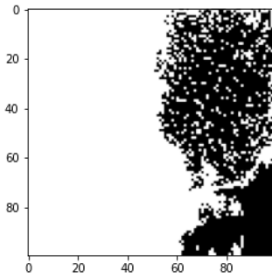
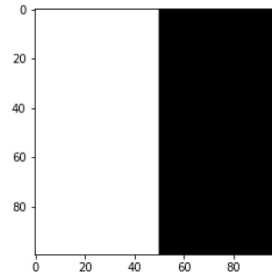
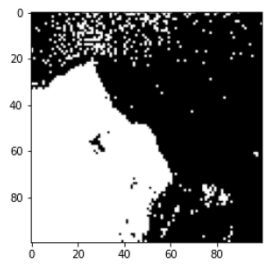
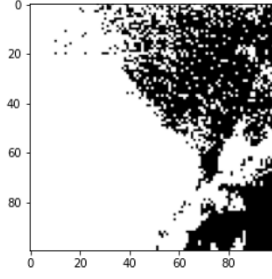
In above charts, the red plot means the data point is cluster1, blue plot means the data point is cluster2. The x-axis and the y-axis of the chart represents two coordinates of data point (because k is 2 in our experiments, so each point has two coordinates in eigensapce).

Obviosly, the data points in same cluster are very close, and connect with each other. That's mean the data points in same cluster have the same coordinates in the eigenspace of graph Laplacian.

c. observations and discussion

In above experiments, we set γ_c and γ_s equal to $1/100000$. In following part, I want to test k-mean clustering with different γ_c and γ_s . We set γ_c and $\gamma_s = 1/100000, 1/100$ and 0 individually, and compare their results.

		γ_c		
		$1/100000$	$1/100$	0
γ_s	$1/100000$			

	1/100			
	0			NaN

According to our total eight experiments, when we set $\gamma_c = \gamma_s = 1/100000$, the results are the best. Also, it's not surprising that the information of "color" is more important than "space". If γ_s is bigger than γ_c (for example, $\gamma_s = 1/100$ and $\gamma_c = 1/100000$), the performance of the result also looks poor. Moreover, if we totally ignore color information, the cluster of data points will be the same as we initial (of course).