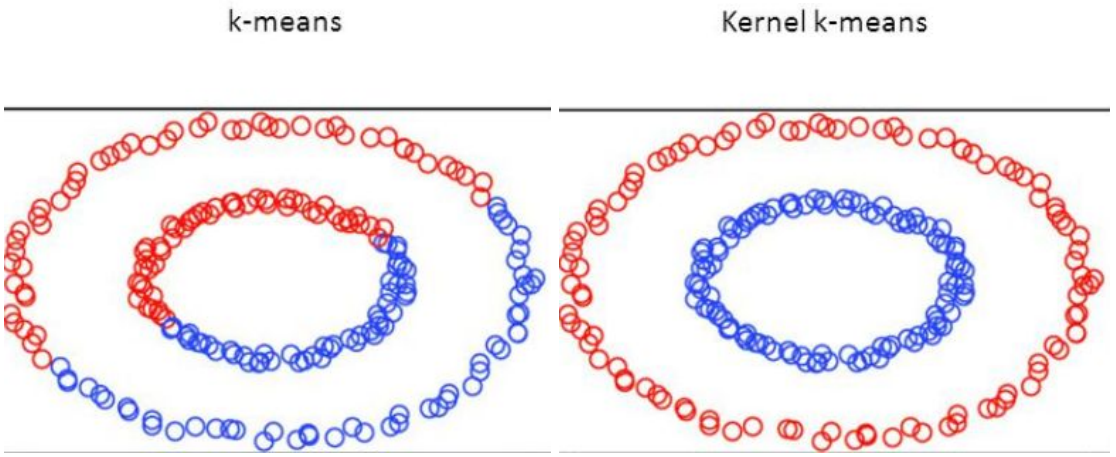
ML HW6 - Kernel K-means & Spectral Clustering

多工所 王彥儒 0856621

Kernel K-means

Introuduction

Kernel K-means is an approach that based on K-means algorithm, but the difference is the feature space. The original K-means merely calculates the Euclidean distance among data points to data points. Yet, Kernel K-means uses the kernel to mapping the features into higher dimensions, to solve some linearly inseparable data distributions. As a result, Kernel K-means can divided the data points into more similar clusters.



Implementation Procedure & Discuss

1-1 Initialization

As same as k-means, the initial methods is an essential part to kernel k-means. There're servel initialization method such as **modK**, the initial pixel labels would be like **[0,1,2,0,1,2,....]** if the k (number of clusters is 3). The second one is **equal**, the initial pixel labels would become **[0,0,0,....1,1,1,....2,2,2,..]** if k=3. These two methods are to make the label distribution uniform. The last one is **random**, so the pixel labels are randomly selected.

Initialization (centers & clusters)

```
def Initial_Kmeans(data, k, mode):
    center_x = np.random.randint(0, h, k)
    center_y = np.random.randint(0, h, k)

    if mode == 'modK':
        prev = []
        for i in range(h*w):
            prev.append(i%k)
        return center_x, center_y, np.array(prev)
    elif mode == 'equal':
        prev = []
        for i in range(h*w):
            prev.append(int(i / (h*w/k)))
        return center_x, center_y, np.array(prev)
    else:
        prev = np.random.randint(0, k, h*w)
        return center_x, center_y, prev
```

1-2 Kernel

In this part, we have to calculate the defined kernel (RBF kernels combination)

The RBF kernel calculation has been mentioned in the last homework (HW5_GP&SVM)

Although the *scipy.spatial.distance.pdist* is the slowest method, the result is as same as the *sklearn.metrics.pairwise.rbf_kernel*. And the numpy method I used in HW5 is a little bit difference from the above two. Thus, I used scipy to calculate the RBF kernel as convenience.

The interesting thing in this part is that we combined the spatial similarity and color similarity into one kernel to get more information. Additionally, the experiment of the hyper-parameter gamma (spatial & color) will have a discussion in the following pages.

Kernel

For both kernel k-means and spectral clustering, please use the new kernel defined below to compute the Gram matrix.

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

```python
from scipy.spatial.distance import pdist, squareform

gamma_c = 1/(255*255)
gamma_s = 1/(100*100)

def kernel(color, coord):

    spatial_K = np.exp(-gamma_s * squareform(pdist(coord, 'sqeuclidean')))
    color_K = np.exp(-gamma_c * squareform(pdist(color, 'sqeuclidean')))

    return spatial_K * color_K
```

1-3 Loss (Difference)

According to the teaching material, the loss (difference) calculation formula is as following:

Loss

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

The main idea here is to calculate the distance of **k cluster** and x_j (#j data point, x:kernel vector), so each cluster will have a loss for each data points. Assume that we are dealing with the pixel 1 (10000,) , the first term **k(**x_j, x_j) will get the same value in every cluster, so we can ignore the first term calculation.

The $|C_k|$ is the total number of the data points in the same cluster.

The second term is calculating the same objective point with different points' kernel value sum, which are in the same cluster.

The third term is calculating the same cluster with different data points, and sum up their kernel value.

I think this part is the most important in the task. If we implement the formula with for loop, the execution time would become very long. **The input pixel is 10^4, the gram matrix computation is $10^4 \times 10^4$. In my early experiment, the converge time of one 100x100 image with 3 clusters is about 30 minute. After vectorizing the matrice, the execution time is within 1 minute.**

```
def second_loss(kernel_data, classification, idx, cluster):
    cluster_sum = 0
    kernel_sum = 0
    cluster_sum = len(classification[classification == cluster]) # correct cluster num |Ck|
    kernel_sum = np.sum(kernel_data[idx][classification == cluster]) # correct cluster kernel value sum (10000,10000)


    if cluster_sum == 0:
        cluster_sum = 1 # prvent from divided zero

    return (-2) / cluster_sum * kernel_sum

def third_loss(kernel_data, classification, k):
    cluster_sum = np.zeros(k, dtype='int')
    kernel_sum = np.zeros(k)

    for i in range(k):
        cluster_sum[i] = np.count_nonzero(classification==i) # correct cluster num |Ck|


    for cluster in range(k):
        same_row = kernel_data[classification == cluster] # row prediction equal to cluster
        same_col = same_row[:, classification == cluster] # col prediction equal to cluster
        kernel_sum[cluster] = np.sum(same_col) # same cluster kernel value sum

    for i in range(k):
        if cluster_sum[i] == 0:
            cluster_sum[i] = 1 # prvent from divided zero
        kernel_sum[i] = 1 / (cluster_sum[i]**2) * kernel_sum[i]

    return kernel_sum

def total_loss(data, kernel_data, classification, k):
    new_classification = np.zeros(data.shape[0], dtype='int') # New loss for each point
    loss_3 = third_loss(kernel_data, classification, k) # 3 terms for same cluster points relation

    for idx in range(data.shape[0]): # Calculate each point's loss (10000)
        loss_for_class = np.zeros(k)
        for cluster in range(k):
            loss_for_class[cluster] = second_loss(kernel_data, classification, idx, cluster) + loss_3[cluster]
        new_classification[idx] = np.argmin(loss_for_class)
```

1-4 Visualization

Last part is to visualize the pixel value distribution and display the clustering procedure with GIF images.

Visualization

```
color_select = ['#377eb8', '#ff7f00', '#4daf4a','#f781bf', '#a65628', '#984ea3','#999999', '#e41a1c', '#dede00']
color_rgb = []
for c in color_select:
    color_rgb.append(ImageColor.getrgb(c))


def Kernel_K_Means(data, coord, k, initial, file_name, max_iteration=20, color_rgb=color_rgb):


    Cx, Cy, classification = Initial_Kmeans(data, k, initial)
    kernel_data = kernel(data, coord)

    iteration = 0
    prev_error = -np.inf
    print(f'classification shape = {classification.shape}')

    while(iteration < max_iteration):
        print(f'iteration = {iteration}')
        prev_classification = classification
        visualize(data, classification, color_rgb, iteration, initial, file_name)
        classification = total_loss(data, kernel_data, classification, k)
        error = np.sum(np.absolute(classification - prev_classification))
        print(f'Error = {error}')

        if error == prev_error:
            break
        prev_error = error
        iteration += 1

# initialization = ['random', 'modK', 'equal-divide']
Kernel_K_Means(img_array, img_coord, k=3, initial='equal', file_name='coast')
```

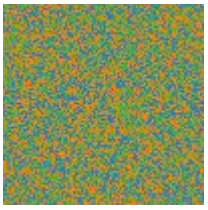
```
def visualize(img_array, classification, color, iteration, initial, file_name):
    h_w, c = img_array.shape
    h = w = int(h_w**0.5)
    img = img_array.reshape(h, w, c)
    for i in range(h):
        for j in range(w):
            img[i,j] = color[classification[i*h + j]]

    plt.axis('off')
    plt.imshow(img)
    plt.show()
    result = Image.fromarray((img).astype('uint8'))
    result.save(f'{file_name}_{initial}_{iteration:02}.png')
```

GIF

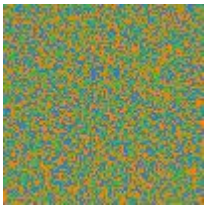
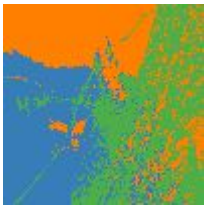
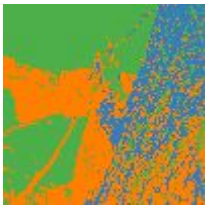
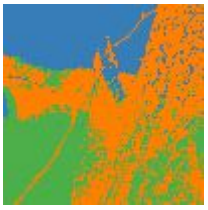
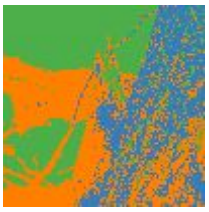
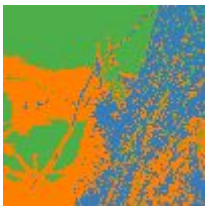
```
import imageio as io
import os



file_names = sorted((fn for fn in os.listdir('.') if fn.startswith('coast_mod')))

#making animation
with io.get_writer('coast_modK.gif', mode='I', duration=0.5) as writer:
    for filename in file_names:
        print(filename)
        image = io.imread(filename)
        writer.append_data(image)
writer.close()
```

Experiment & Result

- Initialization (K=3, max_iteration=20, gamma_c=1/(255x255) gamma_s=1/(100x100))
  - Island

Iteration	Random	modK	equal
0			
3			
6			
9			
Final			 (13)

➢ Rabbit

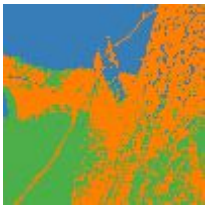
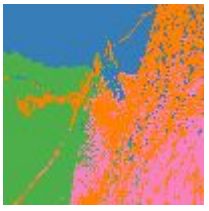
Iteration	Random	modK	equal
0			
3			
6			
9			
Final	 (14)		 (17)

In the experiment, we found that the final iteration (20) image are the same in each initialization. However, equal divided is the first one to converge (difference=0) in the most of cases.

- K cluster (max_iteration=20, init='equal', gamma_c=1/(255x255) gamma_s=1/(100x100))
  - Island

K cluster / Iteration	2	3	4
0			
3			
6			
9			
Final	 (12)		 (12)

➢ Rabbit

K cluster / Iteration	2	3	4
0			
3			
6			
9			
Final	 (10)	 (17)	

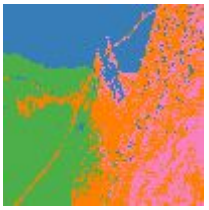
❖ Gamma_color (max_iteration=20, init='random')

➢ Island

K cluster γ_color base color: 1/(255*255)	2	3	4
0.01			
1			
100			

❖ Gamma_spatial (max_iteration=20, init='random')

➢ Rabbit

K cluster γ base spatial: 1/(100*100)	2	3	4
0.01			
1			
100			

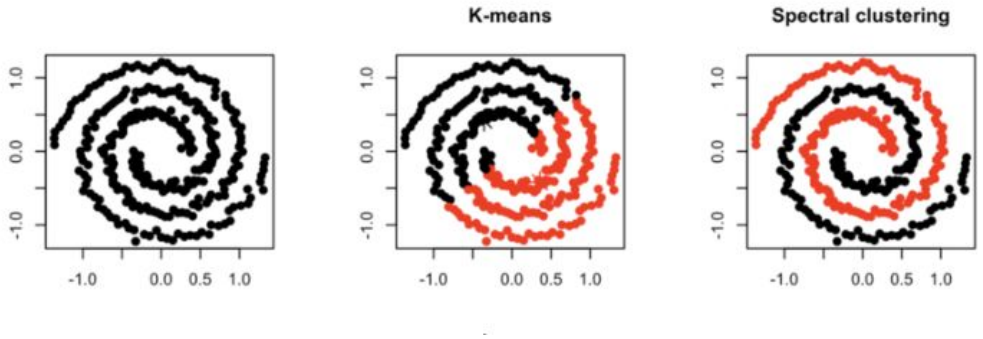
In the gamma hyper parameters experiment, the spatial and color are complementary. Small spatial gamma equal to large color gamma, which results in focusing on the color features of image. On the contrary, small color gamma equal to large spatial gamma, which cause the divided areas into squares or circles space.

Spectral Clustering

Introuduction

Spectral clustering uses the spectrum (eigenvalues) of the similarity matrix of the data to perform dimensionality reduction before clustering in fewer dimensions. The similarity matrix is provided as an input and consists of a quantitative assessment of the relative similarity of each pair of points in the dataset. To partition the two sets, there are two methods “Normalized Cut” and “Ratio Cut”. In the normalize case, it cares about the weight (degree) of data. On the other hand, ratio cut concerns about the number of data. In our class lecture, we sum up that **the spectral methods are special case of kernel k-means**.

The difference between the 2 can easily be shown by this illustration:



Implementation Procedure & Discuss

2-1 Initialization

In the spectral clustering initialization, the initial methods are quite different to kernel k-means. We have to deal with the centers of clusters. The first one random_pick is randomly select k points as cluster centers. Another new method is called kmean++. The main idea of kmean++ is making the initial centers as far as possible between each others. The last one default methods is random from data, which calculating the mean and the standard deviation of data, and randomly create the center value from these two variables.

Initialization (centers & clusters)

```
def Initialization(data, k, method):
    num, feature = data.shape
    cluster = np.zeros([k,feature]) # (k,3)

    classification = np.random.randint(k, size=num)
    if method=='random_pick':
        idx = np.random.randint(low=0, high=num, size=k)
        cluster=data[idx,:]

    elif method == 'kmean++':
        idx_first_pts = np.random.randint(low=0, high=num)
        cluster[0]=data[idx_first_pts]
        for i in range(1,k):
            D = np.zeros([num, i])
            for j in range(i):
                for k in range(num):
                    D[k,j] += np.sum(np.sqrt((data[k]-cluster[j])**2))

            D_min = np.min(D, axis=1) # (10000,) closest distance of point(k) between each cluster
            sum_value = np.sum(D_min)*np.random.rand() # random value * closest distance
            for l in range(num):
                sum_value -= D_min[l]
                if sum_value <= 0: # minus until sum value less than or equal to zero
                    cluster[i] = data[l]
                    break
    else:
        mean = np.mean(data, axis=0)
        std = np.std(data, axis=0)
        for i in range(feature):
            cluster[:,i] = np.random.normal(mean[i], std[i], size=k)


    return cluster, classification
```

2-2 Kernel

Actually as same as kernel k-means kernel. Please refer to the 1-2 Kerenl in the kernel k-means methods.

2-3-1 Unnormalized Laplacian

According to the teaching material, unnormalized method have to calculate the Laplacian L firstly. L contains two part, D (degree matrix) and W (similarity matrix). W is come from kernel method, including the correlation between each points. D is the sum of each points connections. After doing the Laplacian (L=D-W), we got L.

In the second step, we have to calculate the value and vector of Laplacian. The time of calculation is the most time consuming part of the whole task. Afterward, we want to choose the best k eigenvectors as the features from the smallest k eigenvalues. According to the Fiedler vector of graph laplacian, the first non-null eigenvalue is used as the first selected eigenvalue. Yet, in the experiment, the smallest eigenvalue is still non-null. In the selected eigenvector each row represents the coordinate of the data, and each column is the different eigenvectors. We only care about the selected eigenvector with its whole coordinate (U).

Unnormalized Laplacian

$L = D - W$

- Compute the unnormalized Laplacian L
- Compute the first k generalized eigenvectors $u_1, \ldots, u_k$ of the generalized eigenproblem $Lu = \lambda Du$
- Let $U \in \mathbb{R}^{n \times k}$ be the matrix containing the vectors $u_1, \ldots, u_k$ as columns.
- For $i = 1, \ldots, n$, let $y_i \in \mathbb{R}^k$ be the vector corresponding to the i -th row of U
- Cluster the points $(y_i)_{i=1,\ldots,n}$ in $\mathbb{R}^k$ with the k -means algorithm into clusters $C_1, \ldots, C_k$
- Output: Clusters $A_1, \ldots, A_k$ with $A_i = \{j | y_j \in C_i\}$

```
W = kernel(img_array, img_coord) # Similarity matrix
D = np.diag(np.sum(W,axis=1)) # Degree matrix (same row col summation)
L = D - W # Laplacian
L # (10000,10000)
```

```
array([[ 4.11994801e+03, -9.98947075e-01, -9.95412171e-01, ...,
        -8.75557937e-02, -3.05366534e-02, -7.51213408e-02],
       [-9.98947075e-01,  4.08442599e+03, -9.98747383e-01, ...,
        -8.57211420e-02, -2.90068469e-02, -7.31862025e-02],
       [-9.95412171e-01, -9.98747383e-01,  4.03436484e+03, ...,
        -8.33999881e-02, -2.73014429e-02, -7.07504008e-02],
       ...,
       [-8.75557937e-02, -8.57211420e-02, -8.33999881e-02, ...,
         4.64853421e+03, -7.56231329e-01, -9.53576414e-01],
       [-3.05366534e-02, -2.90068469e-02, -2.73014429e-02, ...,
        -7.56231329e-01,  3.57547187e+03, -7.65557361e-01],
       [-7.51213408e-02, -7.31862025e-02, -7.07504008e-02, ...,
        -9.53576414e-01, -7.65557361e-01,  4.44488571e+03]])
```

```
eigenvalue_unnorm, eigenvector_unnorm = np.linalg.eig(L)
```

```
def choose_eigenvector(value, vector, k):

    # eigenvalue = 0 represents a connected component
    idx = np.argsort(value)

    # col->eigenvector row -> coordinate
    # choose top k points
    # ignore the smallest eigenvalue (eigenvalue=0, eigenvectors are the same)
    U = vector[:,idx[1:k+1]]

    return U.real.astype(np.float32)
```

2-3-2 Normalized Laplacian

In the normalized clustering, the first Laplacian calculation is the same as the unnormalized. The main difference is the normalized term, which is the inverse of degree matrix square. The degree matrix is a diagonal matrix, so we have to take care of the divided zero problem. After normalizing Laplacian with degree matrix, we got our new symmetric Laplacian (Lsym).

The following process is same as unnormalized. Choose the k smallest eigenvalue's eigenvectors as the features. (U_norm).
Nevertheless, there's one last thing to do, normalizing the rows norm to 1. Calculating each rows total distance, we divided U_norm rows by rows (T).

Normalized Laplacian

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

- 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_{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 $t_{ij} = u_{ij} / \left(\sum_k u_{ik}^2\right)^{1/2}$
- 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 $A_i = \{j | y_j \in C_i\}$

```
W = kernel(img_array, img_coord) # Similarity matrix
D = np.diag(np.sum(W, axis=1)) # Degree matrix (same row col summation)
L = D - W # Laplacian

D_inv_sqrt = np.zeros_like(D) # (10000,10000)

for i in range(len(D)):
    D_inv_sqrt[i,i] = 1/np.sqrt(D[i,i])
L_sym = D_inv_sqrt @ L @ D_inv_sqrt
```

```
eigenvalue_norm, eigenvector_norm = np.linalg.eig(L_sym)
```

```
k = 3
U_norm = choose_eigenvector(np.real(eigenvalue_norm), np.real(eigenvector_norm), k)
T = np.zeros_like(U)
T = U_norm / (np.sum(U_norm**2, axis=1)**0.5).reshape(-1,1)
new_classification, new_mean = k_means(T, k, initial='kmean++', file_name='rabbit')
```

2-4 K-means

The finally step is to do k-means on the features we extracted from above method. (U:unnorm, T:norm)
There are two steps in k-means, E (Expectation) and M (Maximize). E step is calculating the distance between each points. M step is calculating each clusters' new center points.

K-means with eigenvalue U input

E-step

$$r_{nk} = \begin{cases} 1 & \text{if } k = \underset{k}{\operatorname{argmin}} \|x_n - \mu_k\| \\ 0 & \text{otherwise} \end{cases}$$

M-step

$$\frac{\partial J}{\partial \mu_k} = 0 \Rightarrow 2\sum_{n=1}^{N} r_{nk}(x_n - \mu_k) = 0 \Rightarrow \mu_k = \frac{\sum_n r_{nk} x_n}{\sum_n r_{nk}}$$

```
color_select = ['#377eb8', '#ff7f00', '#4daf4a','#f781bf', '#a65628', '#984ea3','#999999', '#e41a1c', '#dede00']
color_rgb = []
for c in color_select:
    color_rgb.append(ImageColor.getrgb(c))

def k_means(data, k, initial, file_name, max_iteration=20, color_rgb=color_rgb):

    # mean comes from U (k,k) are n points with n coordinates
    mean, classification = Initialization(data, k,'kmean++')
    iteration = 0
    prev_error = -np.inf
    print(f'Mean:\n{mean}')

    while(iteration < max_iteration):
        print(f'Iteration = {iteration}')
        print(f'Current mean:\n{mean}')
        prev_classification = classification
        visualize(data, classification, color_rgb, iteration, initial, file_name)
        classification = E_step(data, mean, k)
        error = np.sum(np.absolute(classification - prev_classification))
        print(f'error = {error}')
        if error == prev_error:
            break
        prev_error = error
        mean = M_step(data, mean, classification, k)
        iteration += 1
    return classification
```

```python
def E_step(data, mean, k):
    new_classification = np.zeros(data.shape[0], dtype='int')

    for idx in range(data.shape[0]):
        distance = np.zeros(k)
        distance = np.sum((data[idx,:]-mean)**2, axis=1) # Distance between each point and cluster

        new_classification[idx] = np.argmin(distance)

    return new_classification

def M_step(data, mean, classification, k):
    # Find the mean of all clusters
    new_mean = np.zeros_like(mean, dtype='float')
    for cluster in range(k):
        new_mean[cluster] = np.sum(data[classification==cluster],axis=0)
        count = len(data[classification==cluster])
        if count == 0:
            count += 1

        new_mean[cluster] = new_mean[cluster] / count

    return new_mean


def visualize(img_array, classification, color, iteration, initial, file_name):
    h_w, c = img_array.shape
    h = w = int(h_w**0.5)
    img = np.zeros([h,w,3])
    for i in range(h):
        for j in range(w):
            img[i,j] = color[classification[i*h + j]]

    plt.axis('off')
    plt.imshow(img.astype('uint8'))
    plt.show()
    result = Image.fromarray((img).astype('uint8'))
```

```python
np.set_printoptions(precision=2)
k = 3
U = choose_eigenvector(np.real(eigenvalue_unnorm), np.real(eigenvector_unnorm), k)
new_classification = k_means(U, k, initial='kmean++', file_name='rabbit')
```

```python
k = 3
U_norm = choose_eigenvector(np.real(eigenvalue_norm), np.real(eigenvector_norm), k)
T = np.zeros_like(U)
T = U_norm / (np.sum(U_norm**2, axis=1)**0.5).reshape(-1,1)
new_classification, new_mean = k_means(T, k, initial='kmean++', file_name='rabbit')
```

2-5 Visualization

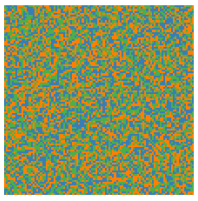
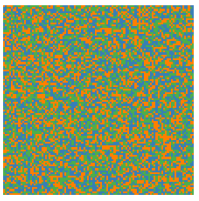
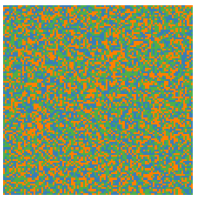
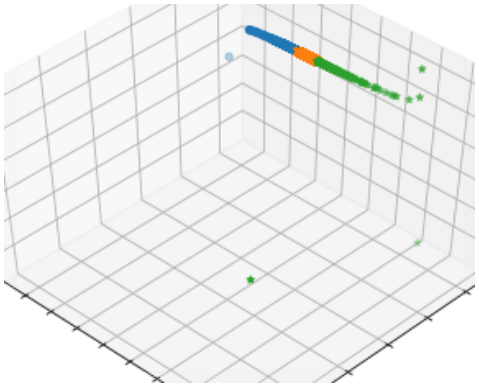
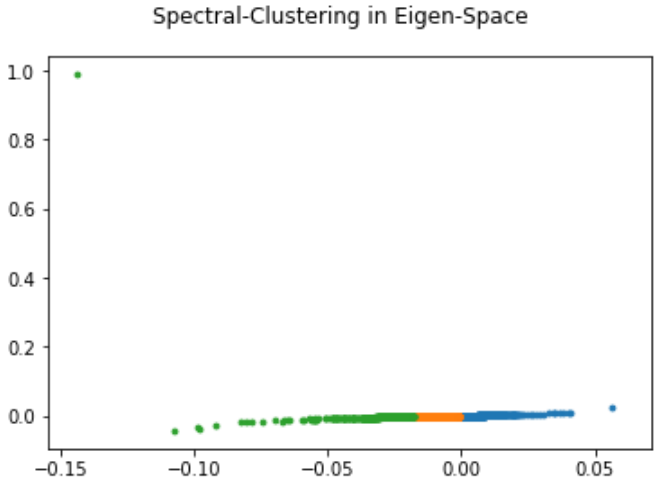
draw the features with 3D and 2D plot.

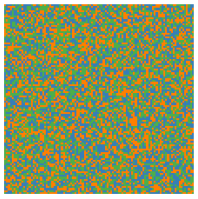
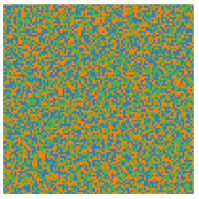
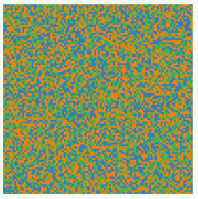
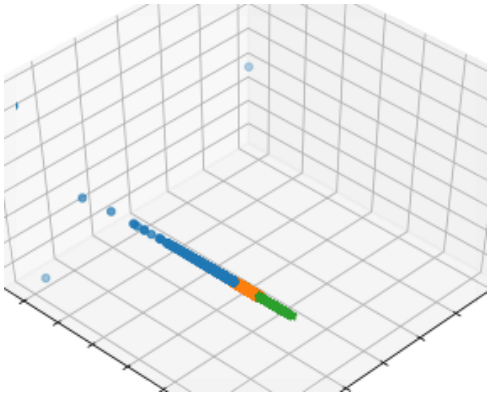
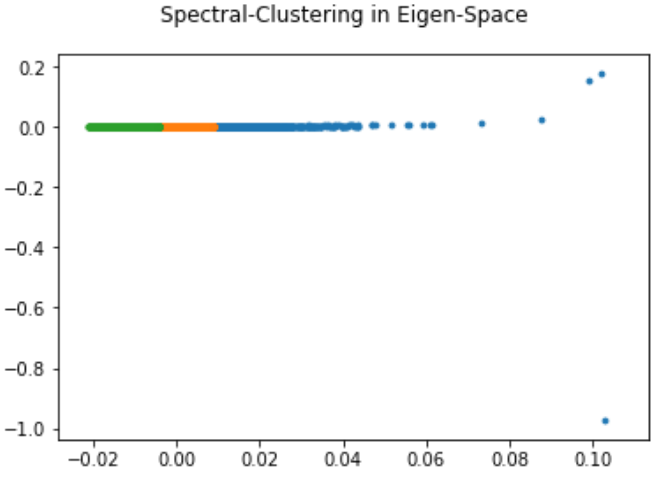
```
from mpl_toolkits.mplot3d import Axes3D

def plot_eigenvector(x, y, z, C)


    fig = plt.figure(1, figsize=(4, 3))
    ax = Axes3D(fig, rect=[0, 0, .95, 1], elev=48, azim=134)
    markers = ['o','x','*']
    for marker, i in zip(markers, np.arange(3)):
        ax.scatter(x[C==i],y[C==i],z[C==i], marker=marker)


    ax.w_xaxis.set_ticklabels([])
    ax.w_yaxis.set_ticklabels([])
    ax.w_zaxis.set_ticklabels([])

    ax.dist = 7

plot_eigenvector(U[:,0], U[:,1], U[:,2], new_classification)
```

```
def draw_eigenspace(classification, data, K):
    plt.clf()
    title = "Spectral-Clustering in Eigen-Space"
    plt.suptitle(title)
    for cluster in range(K):
        plt.scatter(data[classification==cluster][:,0], data[classification==cluster][:,1], s=8)


draw_eigenspace(new_classification, U, k)
```

Experiment & Result

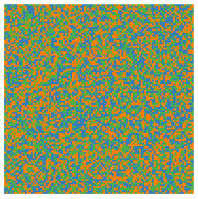
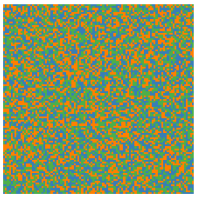
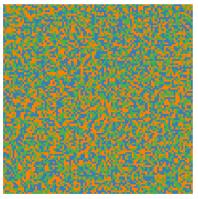
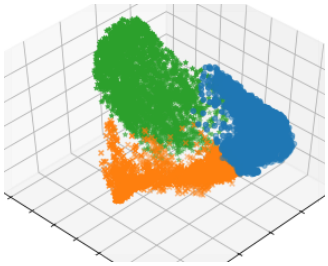
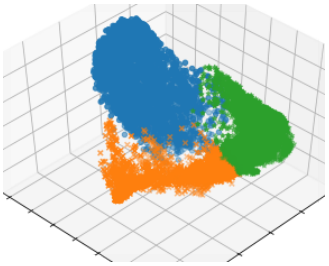
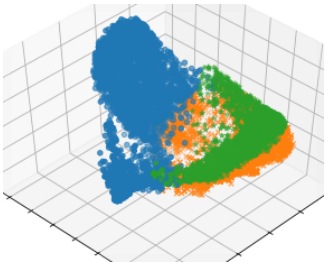
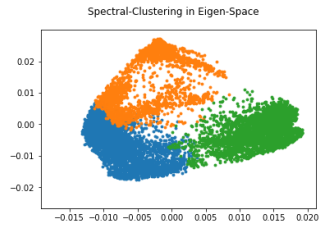
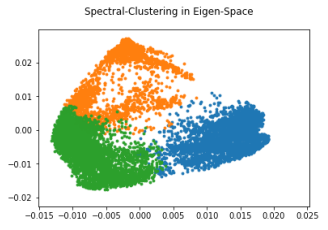
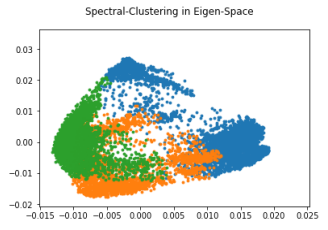
- Unnormalized Initialization (K=3, max_iteration=20, gamma_c=1/(255x255) gamma_s=1/(100x100))

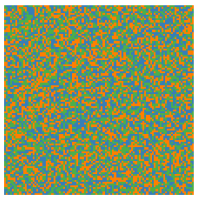
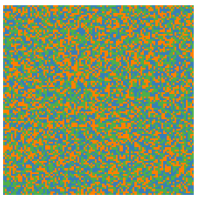
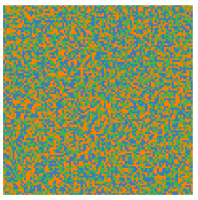
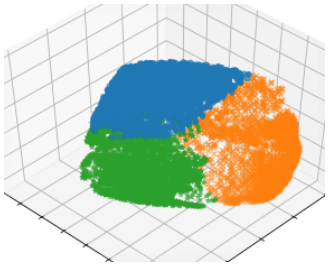
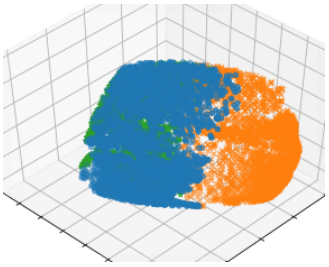
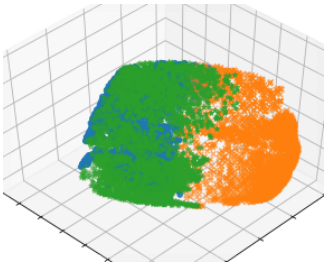
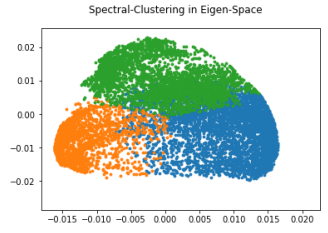
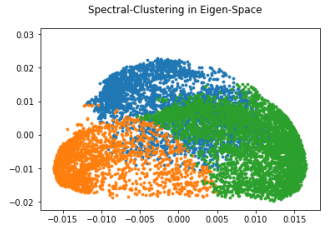
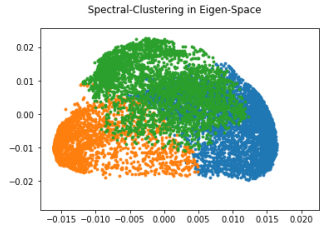
Iteration	random_pick	k-mean++	random_from_data
0			
Final			
			

Iteration	random_pick	k-mean++	random_from_data
0			
Final			
			

The results are quite the same in different initial methods.

❖ Normalized Initialization (K=3, max_iteration=20, gamma_c=1/(255x255) gamma_s=1/(100x100))

Iteration	random_pick	k-mean++	random_from_data
0			
Final			
			
			

Iteration	random_pick	k-mean++	random_from_data
0			
Final			
			
			

❖ K cluster (max_iteration=20, init='kmean++', gamma_c=1/(255x255) gamma_s=1/(100x100))

K cluster / Iteration	2	3	4
Final			

K cluster / Iteration	2	3	4
Final