

## Machine Learning Homework 6

### Kernel K-means and Spectral Clustering

資工所博士班  
周芝妤 0886004

#### ■ a. code with detailed explanations (40%)

Part 1: Here show the codes of visualization:

- ✓ Set a random colormap for data from different cluster.

```
# Visualization
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
from array2gif import write_gif

# predefine colormap
colormap = np.random.choice(range(256),size=(100,3))

def visual(C,k,H,W):
    """
    C: (10000) array, classes of each datapoints
    k: num of clusters
    H: image height
    W: image width
    return : (H,W,3) nrray
    """
    colors= colormap[:,k:]
    ans=np.zeros((H,W,3))
    for h in range(H):
        for w in range(W):
            ans[h,w,:]=colors[C[h*W+w]]

    return ans.astype(np.uint8)
```

- ✓ Use matplotlib to visualize eigenspace of graph Laplacian if k=3 in this “show\_eigenvector” function, and the “gif” function is designed for plotting the gif of clustering procedure by importing write\_gif.

```
# For 3-d datas
def show_eigenvector(x,y,z,C):
    """
    x: (#datapoint) array
    y: (#datapoint) array
    z: (#datapoint) array
    C: (#datapoint) array, class
    """
    fig=plt.figure(figsize =(15,15))
    ax=fig.add_subplot(111,projection='3d')
    markers=['+', 'o', '^']
    for marker,i in zip(markers,np.arange(3)):
        ax.scatter(x[C==i],y[C==i],z[C==i],marker=marker)

    ax.set_xlabel('eigenvector 1-d')
    ax.set_ylabel('eigenvector 2-d')
    ax.set_zlabel('eigenvector 3-d')
    plt.show()

def gif(dataset,gif_name):
    for i in range(len(dataset)):
        dataset[i] = dataset[i].transpose(1, 0, 2)
    write_gif(dataset, gif_name, fps=2)
```

Part 2: Here shows the code of clustering (both Kernel K-means and Spectral Clustering):

✓ Data process and Kernel function :

First, we need to define the kernel to compute the Gram matrix.

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

```
# Kernel functions
from scipy.spatial.distance import pdist, squareform

# Get kernel function: k(x, x') = exp(-r_s * ||S(x) - S(x')||**2) * exp(-r_c * ||C(x) - C(x')||**2)
def kernel(X, r_s=1, r_c=1):
    ...
    X: (10000, rgb=3) ndarray
    r_s: gamma of spacial
    r_c: gamma of color
    return : K (10000, 10000) array
    ...

    n = len(X)
    S = np.zeros((n, 2)) # S is the spacial information: [0, 1], [0, 2]...[99, 99]
    for i in range(n):
        S[i] = [i // 100, i % 100]
    K = squareform(np.exp(-r_s * pdist(S, 'sqeuclidean')) * squareform(np.exp(-r_c * pdist(X, 'sqeuclidean'))))
    return K
```

At beginning, I first calculated “S” as spatial information and original rgb data as color information. Then, I calculated the kernel by using both spatial information and color information. By the way, the original rgb data needs to be flatten, the code shows below.

```
# Deal with data
import numpy as np
import cv2

img1 = cv2.imread('image1.png')
img2 = cv2.imread('image2.png')

# Flatten the image; img[x] as an array with shape (100, 3)
def process(img):
    X, Y, C = img.shape
    flatten = np.zeros((X * Y, C))
    for x in range(X):
        flatten[x * Y:(x + 1) * Y] = img[x]
    return flatten, X, Y
```

✓ Kernel K-means:

The main function of Kernel Kernel K-means shows below:

What we need to do is try to get the minimized difference between data and the center of the cluster.

Firstly, we can get the initial center of each cluster by “initial\_cluster” function which I’ll explain the details later.

There are two steps:

E-step: Fix cluster center and calculate the distance of each data point to the center.

M-step: Update new center by calculating the mean of every cluster.

Keep running these two steps till the error (difference between data and the center) is less than EPS (1e-9).

```
def k_means(X,k,H,W,init):
    """
    X: (#datapoint,#features) array
    k: num of clusters
    H: image H
    W: image W
    return: (#datapoint) ndarray, Ci: belonging class of each data point
    """
    mu=initial_cluster(X,k,init)

    # Classes of each Xi
    C=np.zeros(len(X),dtype=np.uint8)
    pic=[]
    diff=1e5
    count=1
    while diff>EPS :
        # E-step
        for i in range(len(X)):
            dist=[]
            for j in range(k):
                dist.append(np.sqrt(np.sum((X[i]-mu[j])**2)))
            C[i]=np.argmin(dist)

        # M-step
        New_mu=np.zeros(mu.shape)
        for i in range(k):
            belong=np.argwhere(C==i).reshape(-1)
            for j in belong:
                New_mu[i]=New_mu[i]+X[j]
            if len(belong)>0:
                New_mu[i]=New_mu[i]/len(belong)

        diff = np.sum((New_mu - mu)**2)
        mu=New_mu

    # visualize
    p = visual(C,k,H,W)
    pic.append(p)
    print('iteration {}'.format(count))
    for i in range(k):
        print('k={}: {}'.format(i + 1, np.count_nonzero(C == i)))
    print('diff {}'.format(diff))
    print('-----')
    cv2.imshow('', p)
    cv2.waitKey(1)

    count+=1
    return C,pic
```

Then I use these codes below to get the outcome of kernel k means clustering and plot it as gif files. First I flattened the image data and calculated the Gram matrix by “kernel” function. Then used the “k\_means” function to get the data of the cluster of each data and get the pictures during the loop of E-step, M-step. In the last, I use “gif” function to transform the pictures into the gif file. All parameters I tried: gamma\_s = 0.001, gamma\_c = 0.001, k=2/3/4, k\_means\_init = 'forgy'/'random'/'k\_means\_plusplus'

```
# Kernel k-means main
|
image_flat,H,W=process(img1)
gamma_s=0.001
gamma_c=0.001
k=3 # k clusters
k_means_init='forgy'


Gram_matrix=kernel(image_flat,gamma_s,gamma_c)
belongings,pic=k_means(Gram_matrix,k,H,W,init=k_means_init)
gif(pic,'text2.gif')

cv2.waitKey(0)
cv2.destroyAllWindows()
```

✓ Spectral Clustering:

We need to use different Laplacian in order to get the minimization of Ratio Cut/ Normalized Cut.

- Unnormalized Laplacian  $L=D-W$  serve in the approximation of the minimization of RatioCut
- Normalized Laplacian  $D^{-1/2} L D^{-1/2}$  serve in the approximation of the minimization of NormalizedCut.



ah-ha!

**The main step of both are:**

Same as the preprocess of kernel k-means, I firstly flattened the image data and calculated the similarity matrix “W” by “kernel” function.

Then I used the np.diag to get the degree matrix “D” from “W” and calculate the Laplacian L by the equation upon. The next move is to get the eigenvalue & eigenvector of L by using powerful np.linalg.eig tool. The eigenvectors of the k smallest eigenvalue will be collected into the matrix U, which contains these eigenvectors as columns. The final step is to use “k\_means” function to get the clustering pictures and makes the gif file. The only difference between Ratio Cut/ Normalized Cut is they use different kind of Laplacian.

## Normalized Cut:

The normalized Laplacian  $L_{\text{sym}} (D^{-1/2} L D^{-1/2})$  serves in the approximation of the minimization of Normalized Cut.

All parameters I tried:  $\gamma_s = 0.001$ ,  $\gamma_c = 0.001$ ,  $k=2/3/4$ ,  $k\_means\_init =$  'forgy'/'random'/'k\_means\_plusplus'

```
# main of normalized cut

EPS=1e-9

image_flat,HEIGHT,WIDTH=process(img1)
gamma_s=0.001
gamma_c=0.001
k_means_init='k_means_plusplus'
k=2 # k clusters

# similarity matrix
W=kernel(image_flat,gamma_s,gamma_c)
# degree matrix
D=np.diag(np.sum(W,axis=1))
# approximation of normalized cut
L=D-W
D_inverse_square_root=np.diag(1/np.diag(np.sqrt(D)))
L_sym=D_inverse_square_root@L@D_inverse_square_root

eigenvalue,eigenvector=np.linalg.eig(L_sym)
sort_index=np.argsort(eigenvalue)
# U
U=eigenvector[:,sort_index[1:1+k]]

# k-means
belonging,pic=k_means(U,k,HEIGHT,WIDTH,init=k_means_init)

gif(pic,'img1_s2_k++.gif')
if k==3:
    show_eigenvector(U[:,0],U[:,1],U[:,2],belonging)

cv2.waitKey(0)
cv2.destroyAllWindows()
```

### Ratio Cut (unnormalize Laplacian):

The unnormalized Laplacian  $L$  ( $D-W$ ) serves in the approximation of the minimization of Ratio Cut.

All parameters I tried:  $\gamma_s = 0.001$ ,  $\gamma_c = 0.001$ ,  $k=2/3/4$ ,  $k\_means\_init =$  'forgy'/'random'/'k\_means\_plusplus'

```
# main of ratio cut

image_flat, HEIGHT, WIDTH = process(img1)
gamma_s = 0.001
gamma_c = 0.001
k_means_init = 'k_means_plusplus'
k = 3 # k clusters

# similarity matrix
W = kernel(image_flat, gamma_s, gamma_c)
# degree matrix
D = np.diag(np.sum(W, axis=1))
# approximation fo ratio cut
L = D - W

eigenvalue, eigenvector = np.linalg.eig(L)
sort_index = np.argsort(eigenvalue)
# U
U = eigenvector[:, sort_index[1:1+k]]

# k-means
belonging, pic = k_means(U, k, HEIGHT, WIDTH, inite=k_means_init)

gif(pic, 'text3.gif')
if k == 3:
    show_eigenvector(U[:, 0], U[:, 1], U[:, 2], belonging)

cv2.waitKey(0)
cv2.destroyAllWindows()
```

### Part 3: The initialization of k-means clustering:

✓ The different initialization of setting the center of clusters:

I choose different initialized methods including the original kmeans++, forgy initialization and random partition Initialization.

Reference: <https://medium.com/analytics-vidhya/comparison-of-initialization-strategies-for-k-means-d5ddd8b0350e>

Forgy Initialization: chooses any k points from the data at random as the initial points. This method is one of the faster initialization methods for k-Means.

Random Initialization: chooses the center from Gaussian distribution with the mean / variance of the data points.

Kmeans++ Initialization: chooses initial centers which are as far apart from each other as possible.

```
# K-means function

EPS=1e-9
# init type: 'forgy', 'random', 'k_means_plusplus'

def initial_cluster(X,k,init):
    """
    X: data, array
    k: num of clusters
    return: Cluster: (k,#features) array
    Kij: cluster i's j-dim value
    """
    cluster = np.zeros((k, X.shape[1]))
    if init == 'forgy':
        cluster=X[np.random.choice(range(X.shape[0]), replace = False, size = k), :]

    elif init == 'random':
        mu=np.mean(X,axis=0)
        std=np.std(X,axis=0)
        for i in range(X.shape[1]):
            cluster[:,i]=np.random.normal(mu[i],std[i],size=k)

    else: # init == 'k_means_plusplus'
        # pick first cluster mean
        cluster[0]=X[np.random.randint(low=0,high=X.shape[0],size=1),:]
        # pick k-1 cluster mean
        for c in range(1,k):
            dist=np.zeros((len(X),c))
            for i in range(len(X)):
                for j in range(c):
                    dist[i,j]=np.sqrt(np.sum((X[i]-cluster[j])**2))
            dist_min=np.min(dist,axis=1)
            dist_sum=np.sum(dist_min)*np.random.rand()
            for i in range(len(X)):
                dist_sum-=dist_min[i]
                if dist_sum<=0:
                    cluster[c]=X[i]
                    break

    return cluster
```

#### Part 4 (Mostly explained in the descriptions above):

- ✓ For spectral clustering (both normalized cut and ratio cut), you can try to examine whether the data points within the same cluster do have the same coordinates in the eigenspace of graph Laplacian or not. You should plot the result and discuss it in the report.

I think I explained clearly of the spectral clustering code in Part 2 above.

#### For plotting the eigenspace:

I use matplotlib.pyplot to plot the eigenspace.

```
# For 3-d datas
def show_eigenvector(x,y,z,C):
    ...
    x: (#datapoint) array
    y: (#datapoint) array
    z: (#datapoint) array
    C: (#datapoint) array, class
    ...

    fig=plt.figure(figsize =(15,15))
    ax=fig.add_subplot(111,projection='3d')
    markers=['+', 'o', '^']
    for marker,i in zip(markers,np.arange(3)):
        ax.scatter(x[C==i],y[C==i],z[C==i],marker=marker)

    ax.set_xlabel('eigenvector 1-d')
    ax.set_ylabel('eigenvector 2-d')
    ax.set_zlabel('eigenvector 3-d')
    plt.show()
```



## ■ b. experiments settings and results (20%) & discussion (30%)

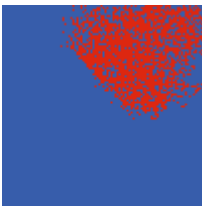
Part 1 & Part2:

### Kernel K-means:

Image1



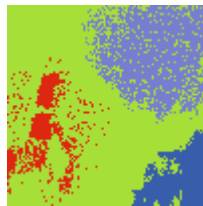
Image2



K=2



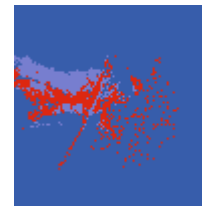
k=3



k=4



K=2



k=3



k=4

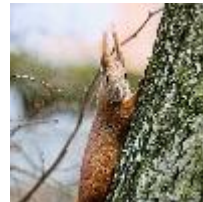
All of these operations used:  $\gamma_s = 0.001$ ,  $\gamma_c = 0.001$ ,  $k\_means\_init = 'k\_means\_plusplus'$ .

### Spectral Clustering with Normalized Cut:

Image1



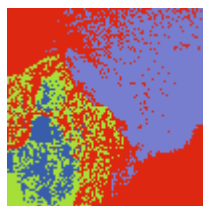
Image2



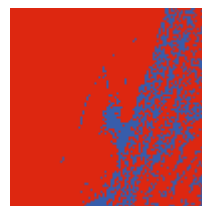
K=2



k=3



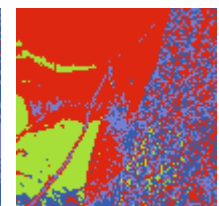
k=4



K=2



k=3



k=4

All of these operations used:  $\gamma_s = 0.001$ ,  $\gamma_c = 0.001$ ,  $k\_means\_init = 'k\_means\_plusplus'$ .

## Spectral Clustering with Unnormalized Cut:

Image1



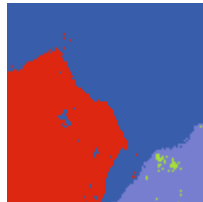
Image2



K=2



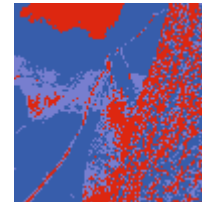
k=3



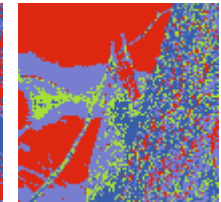
k=4



K=2



k=3



k=4

All of these operations used:  $\gamma_s = 0.001$ ,  $\gamma_c = 0.001$ ,  $k\_means\_init = 'k\_means\_plusplus'$ .

### Discussion:

Overall, I observe that by using spectral clustering, we'll get the "clearer" clustering picture than the kernel k-means method. For instance, in the following case (with "k\_means\_plusplus"):

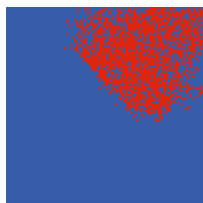
Case A: spectral clustering with normalized cut, image1, k=2

Case B: Kernel k-means, image1, k=2

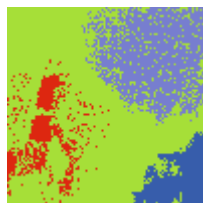
Case C: Kernel k-means, image1, k=4



A



B



C



Original image

In my opinion, I think the spectral clustering with normalized cut method can capture the character in the picture better than kernel k-means. As you can see, in this 2 clusters case, picture A shows the outline of the island of original picture, and the kernel k-means method groups few data points of the island until using k=4.

### Part 3: Different initialization

#### Kernel K-means:

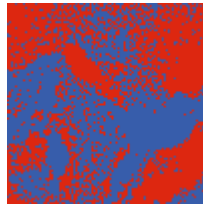
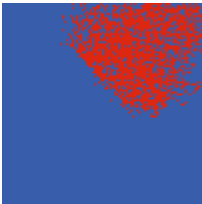
Image1



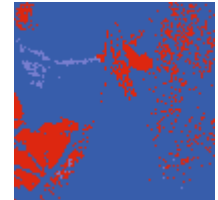
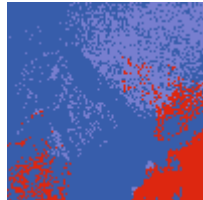
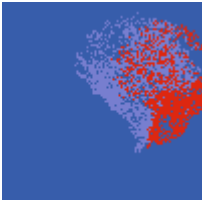
Image2



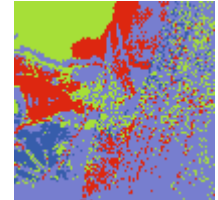
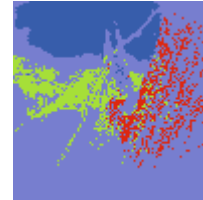
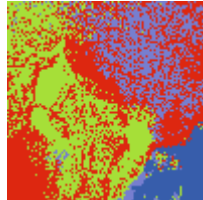
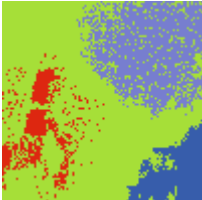
K=2



K=3



K=4



K-means++

Forgy

Random

K-means++

Forgy

Random

All of these operations used:  $\gamma_s = 0.001$ ,  $\gamma_c = 0.001$

In Spectral Clustering part, I took only  $k=4$  as the example here.

All of these operations used:  $\gamma_s = 0.001$ ,  $\gamma_c = 0.001$

### Spectral Clustering with Normalized Cut:

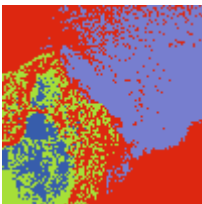
Image1



Image2



K=4



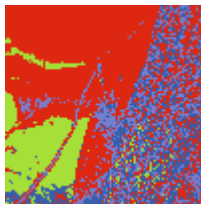
K-means++



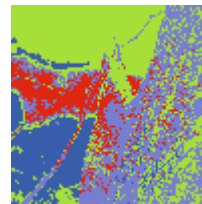
Forgy



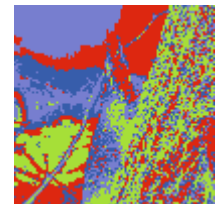
Random



K-means++



Forgy



Random

### Spectral Clustering with Unnormalized Cut:

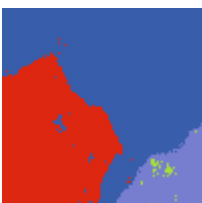
Image1



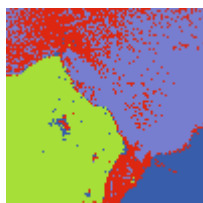
Image2



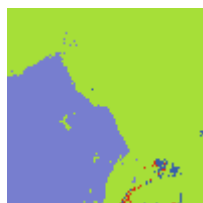
K=4



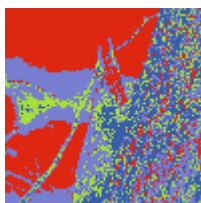
K-means++



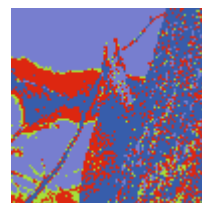
Forgy



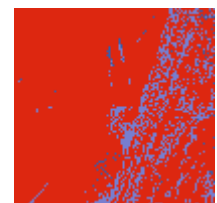
Random



K-means++



Forgy



Random

### Discussion:

Overall, I think in the kernel k-means cases, the Forgy method shows better ability of clustering the data of the island in the image1 case than K-means++ and Random.

In the spectral clustering cases, I think both initialization methods show better ability of clustering image1/image2 than kernel k-means cases.

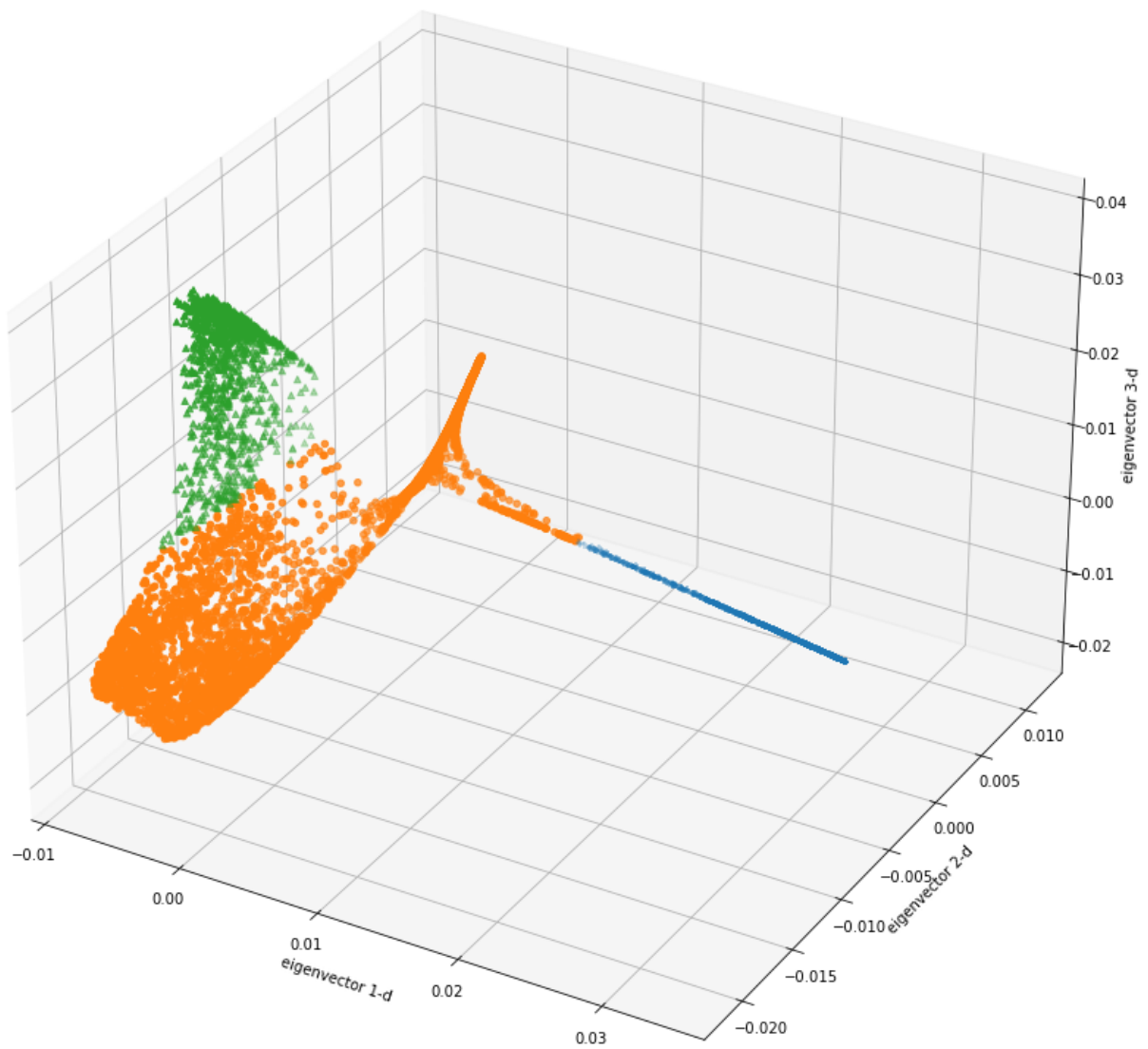
#### Part 4:

✓ For spectral clustering, plotting the eigenspace:

**Image1**

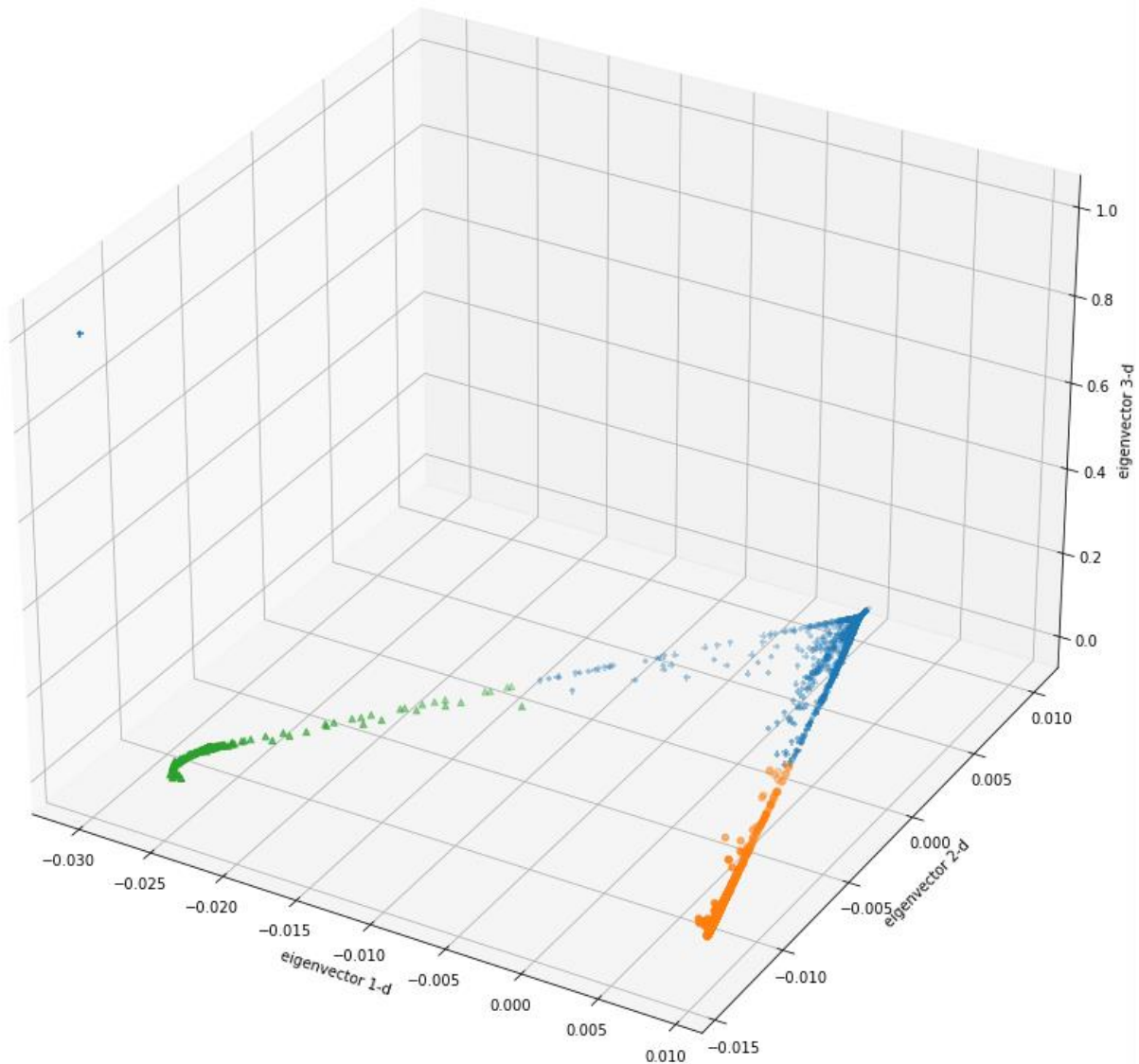
**Normalized cut:**

$k=3$ ,  $\gamma_s = 0.001$ ,  $\gamma_c = 0.001$ ,  $k\_means\_init = 'k\_means\_plusplus'$



### Unnormalized cut:

$k=3$ ,  $\gamma_s = 0.001$ ,  $\gamma_c = 0.001$ ,  $k\_means\_init = 'k\_means\_plusplus'$



### Discussion:

I observed that:

In normalized case, 1-d eigenvector range (-0.01~0.03) is slightly larger than 2-d eigenvector range (-0.02~0.01), and the range of 3-d eigenvector range is largest (-0.02~0.04).

In unnormalized case, 1-d eigenvector range (-0.03~0.01) is slightly larger than 2-d eigenvector range (-0.015~0.01), and the range of 3-d eigenvector range is almost 0 except the outlier with is around 0.9.

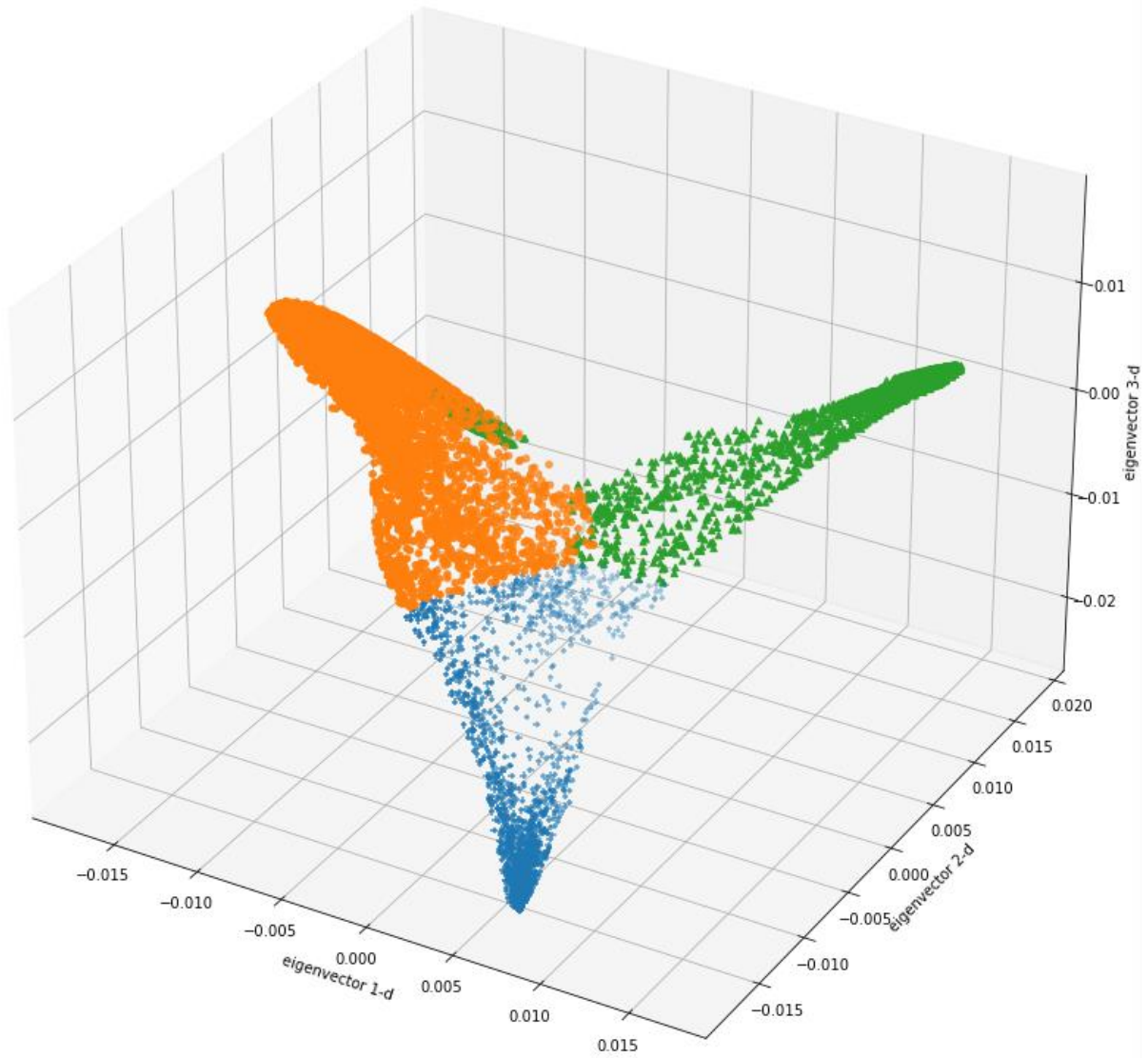
The summation of each dimension of eigenvector is around 0.



## Image2

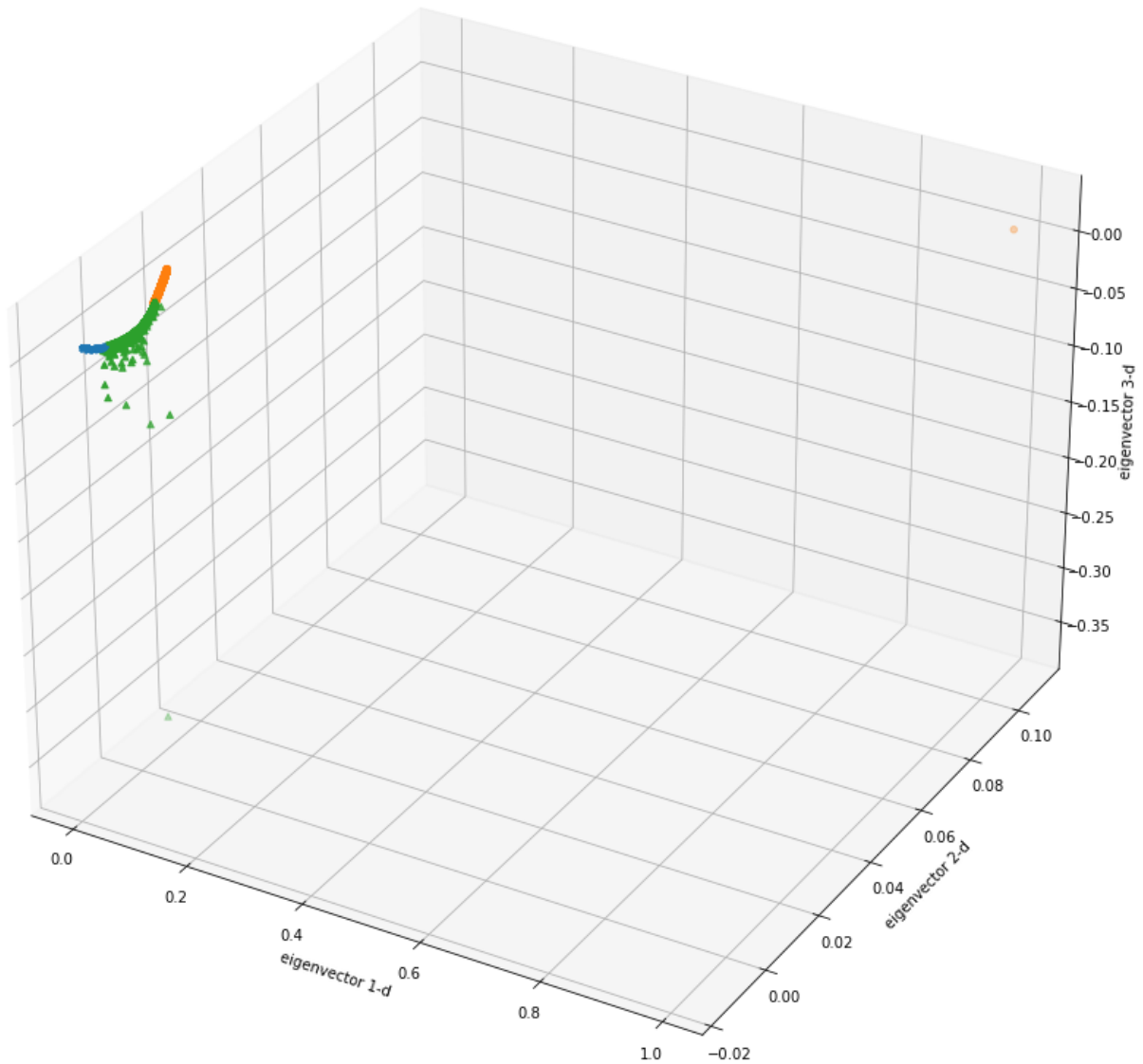
**Normalized cut:**

$k=3$ ,  $\gamma_s = 0.001$ ,  $\gamma_c = 0.001$ ,  $k\_means\_init = 'k\_means\_plusplus'$



### Unnormalized cut:

k=3, gamma\_s = 0.001, gamma\_c = 0.001, k\_means\_init = 'k\_means\_plusplus'



### Discussion:

Overall, I found that in this image2 cases, the range of each eigenvector dimension doesn't really have the clear relation. Same as image1 cases, the summation of each dimension of eigenvector is around 0. It's interesting that the normalized and unnormalized pictures are quite dislike.



### ■ c. observations and discussion (10%)

The first thing I'd like to discuss first is that I observe in the kernel k-means part, "Forgy" and "Random" initialized cases needed more iterations to get to the minimized error ( $< EPS=1e-9$ ) than "k-means++" method. For example, when I run the "Random" method of image1, it need 56 iterations ( $k=4$ ). I think the reason is the randomness of "Forgy" and "Random" method, it might have the possibility to get bad initialization than "k-means++". The "Forgy" method is one of the faster initialization methods for k-means and it just choose any  $k$  points from the data at random as the initial points, so it definitely shows some randomness. The main idea of "k-means++" is to choose initial points which are as far apart from each other as possible, and maybe it is a good initialization sometimes.

The second thing is in the spectral clustering cases, it takes more running time because it need to calculate the Laplacian and also the eigenvalue/eigenvector, but after running the "k-means" function, it takes fewer iteration than kernel k-means ( $<20$  iterations). I can conclude that by calculate the eigenvector obtaining matrix  $U$ , it can provide better feature for computer to do the clustering by k-means method.