Machine Learning Homework 6
Kernel K-means and Spectral Clustering
310706043 肇綺筠

a. Code with detailed explanations + b. experiments settings and results & discussion

- ### Part1

First of all, the kernel function defined in this HW is shown below:

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

Which is used both in kernel k-means and spectral clustering method.
In ***def kernel(x,gs=1,gc=1):***

```
def kernel(x,gs=1,gc=1):
    n = len(x) #x:100x100x3,the length will be 10000
    s = np.zeros((n,2))#columns: width & height -> s records the h & w
of each data point(hxw)
    #sqeuclidean -> vector 間歐式距離的平方
    for i in range(n):
        s[i] = [i//100,i%100] #d1:(0,1),(0,2)...
    k = squareform(np.exp(-
gs*pdist(s,'sqeuclidean')))*squareform(np.exp(-
gc*pdist(x,'sqeuclidean')))
    return k
```

Further on, in both method we need to do kmeans(EM steps) to update the center of each cluster until it converges. I use “kmeans++” to initialize parameter(mean) in this section. The kmeans implementation:

In ***def kmeans(x,k,h,w,type='random',gifpath='default.gif'):***

```
def kmeans(x,k,h,w,type='random',gifpath='default.gif'):

    mean = ini_mean(x,k,type) #return the initial mean of each cluster
    #mean:(# of clusters , the data(mean)feature)
    #record the class of each datapoint(100x100)
    Classes = np.zeros(len(x),dtype=np.uint8)
    segs = []
    diff = 1e8
```

```
    count=1
    while diff > eps:
        #the E-step, find alphank
        for i in range(len(x)):
            dist = []
            for j in range(k):
                dist.append(np.sqrt(np.sum((x[i]-mean[j])**2))) #there will only be k distances in dist
            Classes[i] = np.argmin(dist) #classes[i] records the index of min distance between data and class

        #the M-step:update the center(mean)of the cluster after first classification
        new_mean = np.zeros(mean.shape)
        for i in range(k): #traverse all clusters
            match = np.argwhere(Classes == i).reshape(-1) #this will return the data index that belongs to cluster i
            for j in match:#traverse all the datapoint in cluster i and find the new mean
                new_mean[i]=new_mean[i]+x[j]
            if len(match)>0:
                new_mean[i]=new_mean[i]/len(match)
        diff = np.sum((new_mean-mean)**2)
        mean = new_mean

        seg = visualize(Classes,k,h,w) #do color assignment to each datapoint within different clusters
        segs.append(seg)
        print('step:{}'.format(count))

        for i in range(k):#record how many data is classified in specific cluster during different steps(iteration)
            print('cluster k = {}: data {}'.format(i+1,np.count_nonzero(Classes == i)))
        print('parameter diff = {}'.format(diff))
        print('=====================')
        cv2.imshow('',seg)#the color
        cv2.waitKey(1)
        count+=1
```

```
return Classes,segs
```

Explanations:

- **E-step:** Classifying each data to specific clusters according to the distance
- **M-step:** Update the centroid of each cluster
- **Convergence:** diff < eps = 1e-8
- Diff: ***diff = np.sum((new_mean-mean)**2)***

1. **Kernel K Means:**

   In this part, I chose image 1 as input to visualize the process of clustering each pixels on the image using Kernel K Means method.

   The initial settings:

```
#parameter settings
imgp= 'image1.png'
flat_img,h,w = read_img(imgp)
gs = 0.001
gc = 0.001
k = 3
init_type = 'kmean++'#'kmean++' #'random_k' 'gaussian'
#gif_p =
os.path.join('GIF','{}_{}'.format(imgp.split('.')[0],'kernel_k_means.gi
f'))
#compute the kernel
K = kernel(flat_img,gs,gc)
matches,segs = kmeans(K,k,h,w,type=init_type)
```

   where gs means gamma s for spectral information, gc implies gamma c for color information. K=3 is the number of clusters.

The Clustering result of image1 with gs=gc=0.001 / k=2:	The clustering result of image2 with gs=gc=0.001 / k=2:
	
======================== step:9 cluster k = 1: data 7421 cluster k = 2: data 2579 parameter diff = 0.0 ========================	======================== step:12 cluster k = 1: data 8341 cluster k = 2: data 1659 parameter diff = 0.0 ========================

2. **Spectral Clustering:**

2.1 Ratio Cut (Unormalized):

In ratio cut, we need to find the eigenvalues and vectors of L = D-W, and then select the first k eigenvector to form U (10000 x # of clusters) where the row is the coordinate of each data point. We use U to do the kmeans mentioned above.

```
imgp= 'image2.png'
flat_img,h,w = read_img(imgp)
gs = 0.001
gc = 0.001
k = 3
init_type = 'kmean++'#'kmean++' #'random_k' 'gaussian'
#compute the kernel,W: the association matrix for each graph
W = kernel(flat_img,gs,gc)
#degree matrix, sum up all degree and place them at the diagnol to
form D
#axis = 1 as W = (datapoint,features)
```

```
D = np.diag(np.sum(W,axis=1))
#print(D.shape)
#Laplacian,unormalized -> ratiocut:Trace(H'LH)
L = D-W
#calculating this will take long time, remember to save the path and
load it for next test

eg_value,eg_vector = np.linalg.eig(L)
np.save('{}eg_value_gs{}_gc{}_RCut.npy'.format(imgp.split('.')[0],gs
,gc),eg_value)
np.save('{}eg_vector_gs{}_gc{}_RCut.npy'.format(imgp.split('.')[0],g
s,gc),eg_vector)

#load the precomputed eigen values/vectors
eg_value =
np.load('{}eg_value_gs{}_gc{}_RCut.npy'.format(imgp.split('.')[0],gs
,gc))
eg_vector =
np.load('{}eg_vector_gs{}_gc{}_RCut.npy'.format(imgp.split('.')[0],g
s,gc))
print(eg_vector.shape)
sorted_u = np.argsort(eg_value) #sort the first k eigenvalue, this
indicates the number of connected components(clusters)

U = eg_vector[:,sorted_u[1:1+k]]#pick the eigenvector of L from u2-
uk to form U
#matches:the classes assigned to each datapoint(10000x1)
matches,segs = kmeans(U,k,h,w,type=init_type)
```

Explanations:

- Association matrix W is computed by kernel function
  W = kernel(flat_img,gs,gc)
- Degree matrix
  D = np.diag(np.sum(W,axis=1))
- Laplacian L = D-W
- Compute the eigenvalue of L using ***np.linalg.eig(L)***, then select no.2 to no.k eigenvectors to form U
- Finally throw U into kmeans function and assign the pixels to each

The Clustering result of image1 with gs=gc=0.001 / k=2:	The clustering result of image2 with gs=gc=0.001 / k=2:
	
====================== step:2 cluster k = 1: data 3390 cluster k = 2: data 6610 parameter diff = 1.2926261067661887e-11 ======================	====================== step:4 cluster k = 1: data 5648 cluster k = 2: data 4352 parameter diff = 8.761368294418597e-09 ======================

2.2 Normalized Cut

In normalized cut, we will use Lsym = D^(-1/2)@L@D^(-1/2) to find the eigenvalue and vectors for Kmeans clustering. Moreover, we need to normalize the rows of U to norm1, using T to do kmeans.

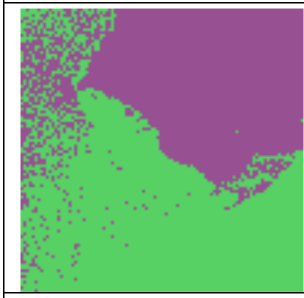
```
W = kernel(flat_img,gs,gc)
D = np.diag(np.sum(W,axis=1))
L = D-W
#do normalization:Lsym
#find:D^(-1/2)
D_inv_sq = np.diag(1/np.diag(np.sqrt(D)))
Lsym = D_inv_sq@L@D_inv_sq

eg_value,eg_vector = np.linalg.eig(Lsym)
np.save('{}eg_value_gs{}_gc{}_NormalizeCut.npy'.format(imgp.split('.
')[0],gs,gc),eg_value)
np.save('{}eg_vector_gs{}_gc{}_NormalizedCut.npy'.format(imgp.split(
'.')[0],gs,gc),eg_vector)

#load the precomputed eigen values/vectors
```

```
eg_value =
np.load('{}eg_value_gs{}_gc{}_NormalizeCut.npy'.format(imgp.split('.')[0],gs,gc))
eg_vector =
np.load('{}eg_vector_gs{}_gc{}_NormalizedCut.npy'.format(imgp.split('.')[0],gs,gc))

sorted_u = np.argsort(eg_value) #sort the first k eigenvalue, this indicates the number of connected components(clusters)
U = eg_vector[:,sorted_u[1:1+k]]#pick the eigenvector of L from u2-uk to form U
denom = np.sqrt(np.sum(np.square(U),axis=1)).reshape(-1,1)
T = U/denom
matches,segs = kmeans(T,k,h,w,type=init_type)
```

The Clustering result of image1 with gs=gc=0.001 / k=2:	The clustering result of image2 with gs=gc=0.001 / k=2:
	
step:4 cluster k = 1: data 3471 cluster k = 2: data 6529 parameter diff = 0.0	====================== step:22 cluster k = 1: data 3752 cluster k = 2: data 6248 parameter diff = 0.0 ======================

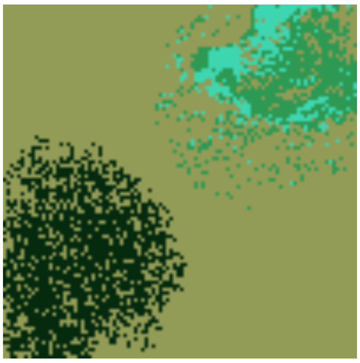
- ## Part2

More clustering results:

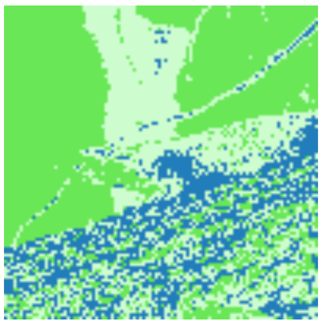
1. **Kernel-Kmeans clustering**

The Clustering result of image1 with gs=gc=0.001 / k=3:	The clustering result of image2 with gs=gc=0.001 / k=3:

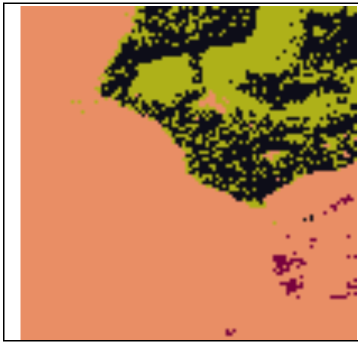
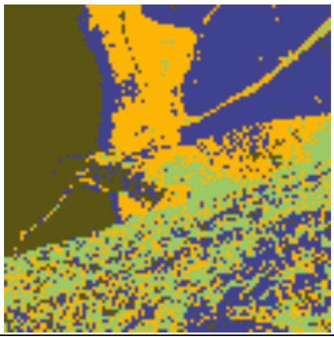
	

The Clustering result of image1 with gs=gc=0.001 / k=4:	The clustering result of image2 with gs=gc=0.001 / k=4:
	

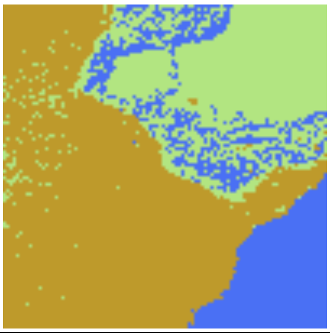
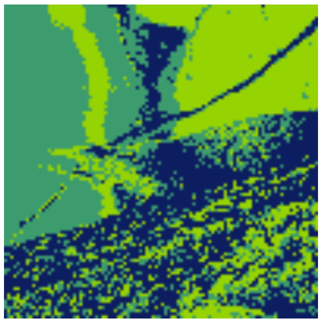
2. RatioCut (unnormalized L)

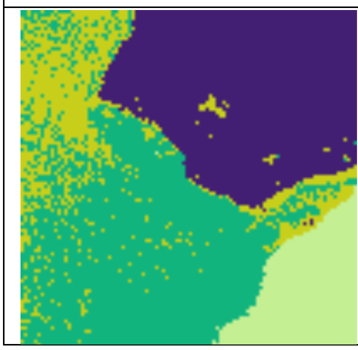
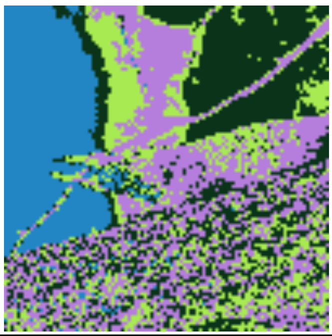
The Clustering result of image1 with gs=gc=0.001 / k=3:	The clustering result of image2 with gs=gc=0.001 / k=3:
	

The Clustering result of image1 with gs=gc=0.001 / k=4:	The clustering result of image2 with gs=gc=0.001 / k=4:

3. Normalized Cut

The Clustering result of image1 with gs=gc=0.001 / k=3:	The clustering result of image2 with gs=gc=0.001 / k=3:
	

The Clustering result of image1 with gs=gc=0.001 / k=4:	The clustering result of image2 with gs=gc=0.001 / k=4:
	

- Part3

I've chosen three different kinds of initialization method to do clustering. Which is kmeans++ / random choice / Gaussian sample.

1. **Kmeans++**

   The algorithm is defined as below steps:

   1. *Randomly select the first centroid from the data points.*
   2. *For each data point compute its distance from the nearest, previously chosen centroid.*
   3. *Select the next centroid from the data points such that the probability of choosing a point as centroid is directly proportional to its distance from the nearest, previously chosen centroid. (i.e. the point having maximum distance from the nearest centroid is most likely to be selected next as a centroid)*
   4. *Repeat steps 2 and 3 until k centroids have been sampled*

Code Detail:

```
    if type == "kmean++":
        #the first cluster mean, randomly choose a datapoint as mean(center of the cluster)
        cluster[0] = x[np.random.randint(low=0,high=x.shape[0],size=1),:]

        for c in range(1,k): #len(x) = number of datapoints
            d = np.zeros((len(x),c)) #d records the distance between data point and the center of each cluster
            for i in range(len(x)): #pick one data
                for j in range(c): #calculate the distance between the data and  centers (mean) of all other clusters
                    d[i,j] = np.sqrt(np.sum((x[i]-cluster[j])**2))
            d_min = np.min(d,axis=1) #filter out the min distance
            sum = np.sum(d_min)*np.random.rand()
            for i in range(len(x)):
                sum-=d_min[i]
                if sum<=0: #the min distance makes sum <=0(max), we should select it as centorid
                    cluster[c] = x[i]#find the new mean and update the center of cluster
```

The final return is cluster[c](k*pixel feature), which record each initial centroid of k clusters.

The clustering result:

I used kmeans++ as default initialization method in part1&2.

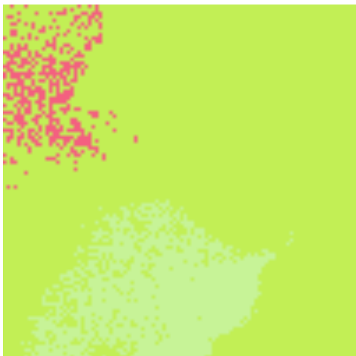
2. random select (type == random_k)

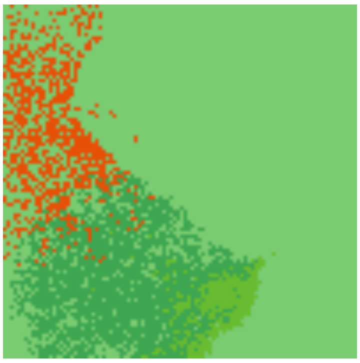
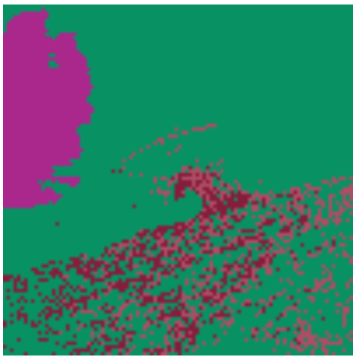
Randomly sample k datapoints as cluster centers

```
elif type == 'random_k': #randomly choose k numbers as center
    rand = np.random.randint(low=0,high=x.shape[0],size=k)
    cluster = x[rand,:]
```

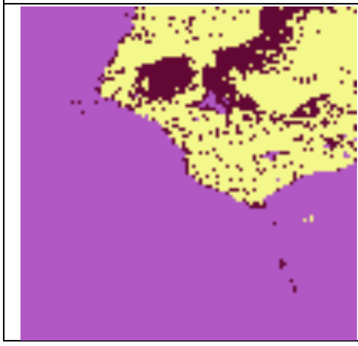
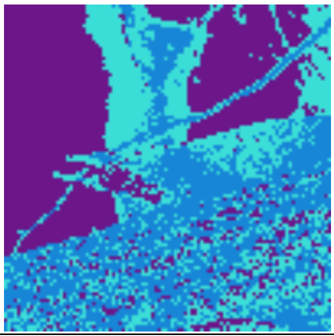
The clustering result:

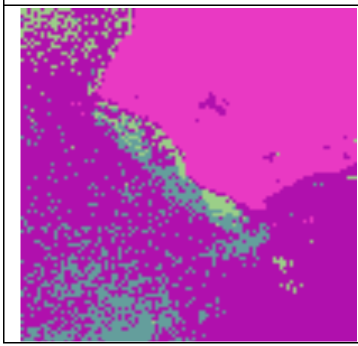
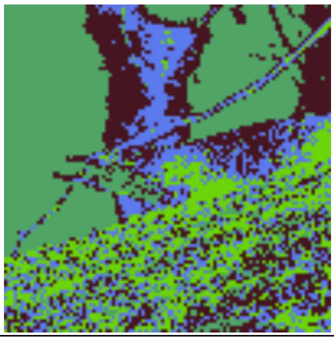
1. Kernel-Kmeans clustering – random initialization

<table>
<tr><td>The Clustering result of image1 with gs=gc=0.001 / k=3:</td><td>The clustering result of image2 with gs=gc=0.001 / k=3:</td></tr>
<tr><td></td><td></td></tr>
</table>

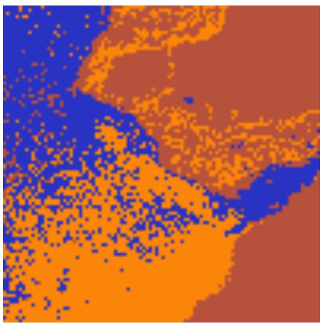
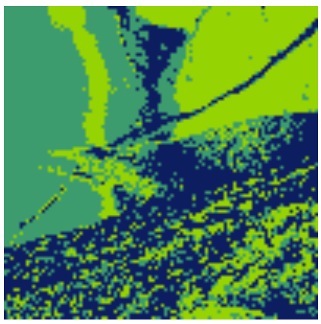
<table>
<tr><td>The Clustering result of image1 with gs=gc=0.001 / k=4:</td><td>The clustering result of image2 with gs=gc=0.001 / k=4:</td></tr>
<tr><td></td><td></td></tr>
</table>

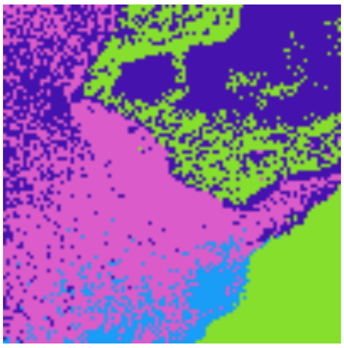
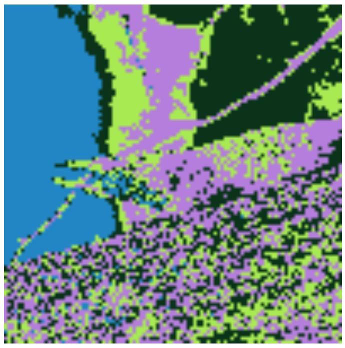
2. RatioCut (unnormalized L) – random initialization

The Clustering result of image1 with gs=gc=0.001 / k=3:	The clustering result of image2 with gs=gc=0.001 / k=3:
	

The Clustering result of image1 with gs=gc=0.001 / k=4:	The clustering result of image2 with gs=gc=0.001 / k=4:
	

3. Normalized Cut– random initialization

The Clustering result of image1 with gs=gc=0.001 / k=3:	The clustering result of image2 with gs=gc=0.001 / k=3:
	

The Clustering result of image1 with gs=gc=0.001 / k=4:	The clustering result of image2 with gs=gc=0.001 / k=4:
	

3. **Gaussian Sample**

   In this part, I initialize the centroid of each cluster by using Gaussian sample. The algorithm is shown below:

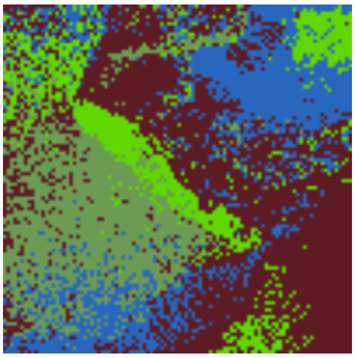
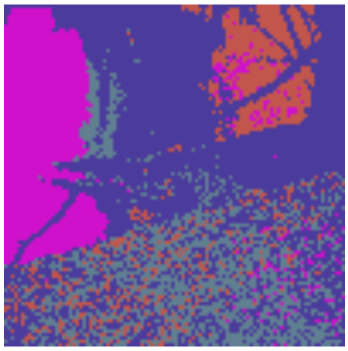
   1. *Compute the mean and standard deviation of dataset(k(x,x))*
   2. *Sample k datapoints with the mean and std obtained in step1. using Gaussian (normal distribution), treat these k datapoints as centers of each cluster.*

```
else: #gaussian sample, use the mean and  variance of dataset to find
the center of the cluster
      mean = np.mean(x,axis=0)#we need to calculate the mean according
to 0 axis(feature)of data
      std = np.std(x,axis=0)
      for c in range(x.shape[1]):#c traverse the feature of x!
         cluster[:,c] =
np.random.normal(mean[c],std[c],size=k)#cluster c use mean[c]/std[c]
```

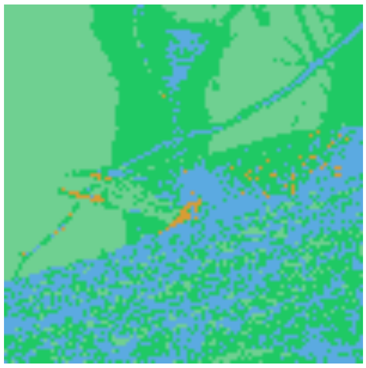
The clustering result:(I only show k=4 for simplicity)

1. **Kernel-Kmeans clustering – Gaussian**(this seemed to take more steps to converge!)

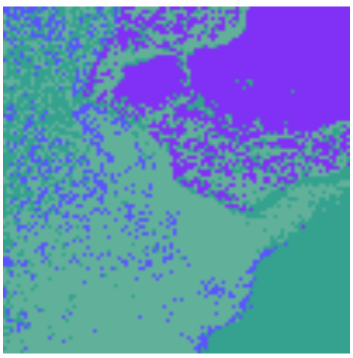
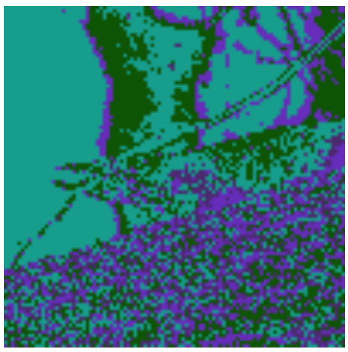
The Clustering result of image1 with gs=gc=0.001 / k=4:	The clustering result of image2 with gs=gc=0.001 / k=4:

2. RatioCut (unnormalized L) – Gaussian

The Clustering result of image1 with gs=gc=0.001 / k=4:	The clustering result of image2 with gs=gc=0.001 / k=4:
	

3. Normalized Cut– random initialization

The Clustering result of image1 with gs=gc=0.001 / k=4:	The clustering result of image2 with gs=gc=0.001 / k=4:
	

- ## Part4

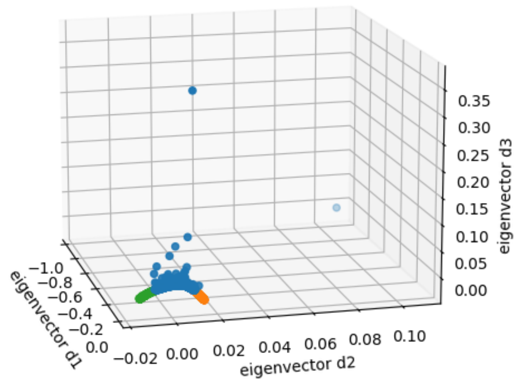
For the plotting part, I graphed the eigenspace with k=3, where each data has its eigenvectors u0,u1,u2 for each dimension(x,y,z axis)

```
U = eg_vector[:,sorted_u[1:1+k]]#pick the eigenvector of L from u2-uk to
form U
#matches:the classes assigned to each datapoint(10000x1)
matches,segs = kmeans(U,k,h,w,type=init_type)
if k ==3:
    plotting(U[:,0],U[:,1],U[:,2],matches)
```

```
  def plotting(x,y,z,classes):
    graph = plt.figure()
    #subplot1 with row=col=1
    ax = graph.add_subplot(111,projection='3d')
    for i in range(3):
        ax.scatter(x[classes==i],y[classes==i],z[classes==i])
        ax.set_xlabel('eigenvector d1')
        ax.set_ylabel('eigenvector d2')
        ax.set_zlabel('eigenvector d3')
    plt.show()
```

1. Ratio Cut:

Eigenspace visualization (initial: kmeans++/image 2/k=3/gs = gc = 0.001)



a. From the graph above, we can clearly see that the data within the same cluster gather, and the summation of the eigenvector of each dimension = 0

b. The steps it took to converge = 4

   Result:

   step:4

   cluster k = 1: data 3565

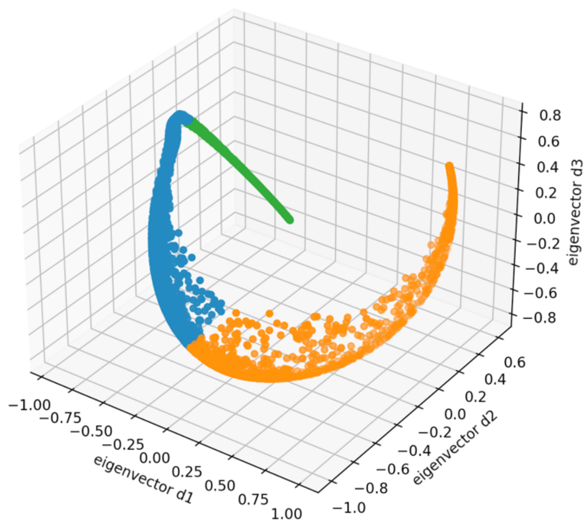
   cluster k = 2: data 4157

   cluster k = 3: data 2278

   parameter diff = 1.2923036759645723e-09

2. Normalized Cut:

Eigenspace visualization (initial: kmeans++/image 2/k=3/gs = gc = 0.001)



a. From the graph above, we can clearly see that the data within the same cluster gather, and the summation of the eigenvector of each dimension = 0. Moreover, the graph is pretty dislike the one with unnormalized clustering

b. The steps it took to converge = 11

   step:11

   cluster k = 1: data 4007

   cluster k = 2: data 3667

   cluster k = 3: data 2326

   parameter diff = 0.0

c. Observations and discussion

1. Execution time:

Kernel k-means < Ratio(unnormalized) Cut < Normalized Cut

As we need to compute the "eigenvalue and vectors" of Lapacian, so it takes much more time to do classification using spectral clustering.

2. The Convergence

With k=3,image=2,gs=gc=0.001

Cluster / Initial	k-means++	Random-k	Gaussian
Kernel-k-means	Step:7/16…	Step:7/30…	Step:8
Ratio Cut	Step:8	Step:5	Step:9
Normalized Cut	step:11	Step:16	Step:13

Observation:

a. As image2 has a more complex pattern, it asks for more steps to converge compared to image1.

b. Mostly, random-k initialization method requires more steps to converge and is more "unstable", while k-means++ usually outperforms other methods. For Gaussian, I found a "85-steps" convergence of image2 clustering using kernel-k-means, I assume the randomness caused by normal distribution sampling may somehow affect the stability of convergence.