Index

a. code with detailed explanations

a. kernel k-mean - Part1

1.

Import data (image1.png) ,and threat it as 10000 datas.

“pixel” represent color, “index of pixel” represent its position.

Then, call “cluster” function to do kernel k-mean.

```
filename = 'data/image1.png'
storename = 'visualization/image1'
pixel, index_of_pixel = read_input(filename)
#print(pixel)

gif_num=cluster(pixel,index_of_pixel)
```

2.

First, we will call “initial” function to do initial (please see part 3 for more explanation of initial).

Second, we will call “kernel” function to compute gram matrix.

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

Then, keep doing E_step and M_step until converge.

(“visualization” function will generate image for gif)

```
def cluster(pixel,index_of_pixel):
    #initial
    this_ri,this_aik,this_Ck,mu= initial(pixel,index_of_pixel,initial_method)
    all_kernel=kernel(pixel,index_of_pixel)

    for i in range(len(mu)):
        x=mu[i] // 100
        y=mu[i] % 100

        print("mean is :" ,x,y)

    itera=1
    for i in range(epochs):
        itera+=1

        pre_ri=np.copy(this_ri)
        #E_step
        this_ri=E_step(this_aik,this_Ck,all_kernel)
        visualization(this_ri,iteration=i)
        #M_step
        this_aik,this_Ck=M_step(this_ri)

        error=calculate_error(this_ri,pre_ri)
        if(error<=10):
            break



    visualization(this_ri,iteration='final_iteration')

    return itera
```

```
def kernel(pixel,coord):
    spatial_sq_dists = squareform(pdist(coord, 'sqeuclidean'))
    spatial_rbf = np.exp(-gamma_s * spatial_sq_dists)


    color_sq_dists = squareform(pdist(pixel, 'sqeuclidean'))
    color_rbf = np.exp(-gamma_c * color_sq_dists)
    kernel = spatial_rbf * color_rbf


    return kernel
```

```
def visualization(ri,iteration):

    img = Image.open(filename)
    width, height = img.size
    pixel = img.load()
    for i in range(img.size[0]):
        for j in range(img.size[1]):
            pixel[j, i] = color[int(ri[i * num + j])]
    img.save(storename+"method_"+initial_method+"_K-" +str(k)+ '_'+str(time.time())+'_' + str(iteration) + '.png')
```

3.E_step

“E_step” function will use “aik” and “ck” to compute new “ri” and return “ri”.

The E_step formula come from the lecture

First, we use this formula to compute “objective”.

Then, we call “calculate_ri(objective)” to compute ri.

$$\left\|\phi(x_j) - \mu_k^{\phi}\right\| = \left\|\phi(x_j) - \frac{1}{|C_k|}\sum_{n=1}^{N}\alpha_{kn}\phi(x_n)\right\|$$

$$= \mathbf{k}(x_j, x_j) - \frac{2}{|C_k|}\sum_{n}\alpha_{kn}\mathbf{k}(x_j, x_n) + \frac{1}{|C_k|^2}\sum_{p}\sum_{q}\alpha_{kp}\alpha_{kq}\mathbf{k}(x_p, x_q)$$

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

```
def E_step(aik,Ck,all_kernel):
    print("E_STEP")
    objective=np.zeros((10000,k))


    #gram_martrix
    third=np.zeros((k))
    for j in range(k):
        for m in range(10000):
            for n in range(10000):
                third[j] += aik[m][j] * aik[n][j] *all_kernel[m][n]


    for i in range(10000):
        for j in range(k):
            first=all_kernel[i][i]
            second=0
            for m in range(10000):
                second += aik[m][j]*all_kernel[i][m]


            objective[i][j]=first - 2 * second / (Ck[j]+1) + third[j]/((Ck[j]+1)**2)


    this_ri=calculate_ri(objective)

    return this_ri
```

```
def kernel(pixel,coord):
    spatial_sq_dists = squareform(pdist(coord, 'sqeuclidean'))
    spatial_rbf = np.exp(-gamma_s * spatial_sq_dists)


    color_sq_dists = squareform(pdist(pixel, 'sqeuclidean'))
    color_rbf = np.exp(-gamma_c * color_sq_dists)
    kernel = spatial_rbf * color_rbf


    return kernel
```

```
def calculate_ri(objective):
    ri=np.zeros((10000))
    miniz=np.argmin(objective,axis=1)
    for i in range(10000):
        ri[i]=miniz[i]

    return ri
```

4.m_step

“M_step” function will use “ri” to compute new “aik” and new ”ck” and return them.

```
def M_step(this_ri):
    print("M_STEP")
    this_Ck=np.zeros((k),dtype='int64')
    this_aik=np.zeros((10000,k),dtype='int64')
    for i in range(pixel_num):
        i_cluster=int(this_ri[i])
        #print(i_cluster)
        this_aik[i][i_cluster] = 1
        this_Ck[i_cluster] += 1

    return this_aik,this_Ck
```

5.converge condition

this function compute the different between new ri and previous ri。

```
def calculate_error(this_ri,pre_ri):
    error=0
    for i in range(len(this_ri)):

        if this_ri[i]!=pre_ri[i]:
            error+=1

    return error
```

6. gif

This function will generate gif.

```
def get_gif(pics_dir,n):
    imgs = []
    files = os.listdir(pics_dir)
    for i in files:
        pic_name = os.path.join(pics_dir, i)
        print(pic_name)
        temp = Image.open(pic_name)
        imgs.append(temp)

    save_name =storename+"k_"+str(k)+"ini_method_"+initial_method + '{}.gif'.format(pics_dir)

    imgs[0].save(save_name, save_all=True, append_images=imgs, duration=400)
    return save_name
```

a. kernel k-mean – Part2

if we want to change cluster number, we can change ‘k’ here.

```
k=5
epochs = 200
gamma_c = 1/(255*255)
gamma_s = 1/(100*100)

num = 100
color = [(0,0,0), (100, 0, 0), (0, 255, 0), (255,255,255),(255,0,0),(0,0,255),(255,0,255),(0,255,255),(255,255,0)]
gif_num=0


Ck=np.zeros((k),dtype='int64')
aik=np.zeros((10000,k),dtype='int64')
ri=np.zeros((10000),dtype='int64')
initial_method='kmean++'
pixel_num=10000

filename=""
storename=""


if __name__ == "__main__":
```

a. kernel k-mean – Part3

“initial” function will use different way to initial according to “initial_method”.
First, it will choose k centers according to “initial_merhod”.(include random, kmean++,c-kmean++)
Initial method “random” will random choose k data as center.

Then, calling “find_center” function to initial aik,ck,ri with chosen k centers.

```python
def initial(data,index_of_pixel, initial_method):
    this_ri=np.copy(ri)
    this_aik=np.copy(aik)
    this_Ck=np.zeros((k))



    if initial_method == 'random':
        rand=np.random.randint(low=0,high=10000,size=(1 * k))
        print(rand)

        for i in range(pixel_num):
            cluster=find_center(center=rand,i=i,index_of_pixel=index_of_pixel)
            this_ri[i]=cluster
            this_aik[i][cluster] = 1
            this_Ck[cluster] += 1
        return this_ri,this_aik,this_Ck,rand
    elif initial_method == "kmean++":
        rand=k_mean_plus(data)
        print(rand)
        for i in range(pixel_num):
            cluster=find_center(center=rand,i=i,index_of_pixel=index_of_pixel)
            this_ri[i]=cluster
            this_aik[i][cluster] = 1
            this_Ck[cluster] += 1
        return this_ri,this_aik,this_Ck,rand
    elif initial_method=="c-kmean++":
        rand=c_k_mean_plus(data)
        print(rand)
        for i in range(pixel_num):
            cluster=find_center(center=rand,i=i,index_of_pixel=index_of_pixel)
            this_ri[i]=cluster
            this_aik[i][cluster] = 1
            this_Ck[cluster] += 1
        return this_ri,this_aik,this_Ck,rand
```

```python
def find_center(center,i,index_of_pixel):

    index=index_of_pixel[i]
    x=index[0]
    y=index[1]

    min_dis=10000000000
    min_center=-1

    for center_num in range(k):


        c_x=index_of_pixel[int(center[center_num])][0]
        c_y=index_of_pixel[int(center[center_num])][1]

        this_dis=((x-c_x) ** 2) + ((y-c_y) ** 2)

        if this_dis<min_dis:
            min_dis=this_dis
            min_center=center_num

    return min_center
```

“k_mean_plus” function will use kmean++ algorithm to choose k center.

(k-mean++ algo. was found on the website.)
Choose first one center randomly.
Then, compute distance for all data to their closest center. And choose the data which has longest distance as a new center.
Keep doing above step until we find k center.

```python
def k_mean_plus(data):

    mean=np.zeros((k,2))


    mean[0,0]=np.random.randint(low=0,high=100)
    mean[0,1]=np.random.randint(low=0,high=100)

    for u_num in range(1,k):
        all_distance=np.zeros((10000))

        for i in range(10000):
            x=i // 100
            y=i % 100
            this_dis=np.zeros(u_num)
            for j in range(u_num):
                this_dis[j]=(mean[j,0]-x) ** 2 + (mean[j,1]-y) ** 2

            all_distance[i]=this_dis[np.argmin(this_dis)]

        max_distance=np.argmax(all_distance)
        mean[u_num,0]=max_distance  // 100
        mean[u_num,1]=max_distance %100

    rand=np.zeros((k))

    for i in range(k):
        rand[i]=mean[i,0] + mean[i,1] *100

    return rand
```

“c_k_mean_plus” function will use kmean++ algorithm to choose k center.
It also use k-mean++ algo. but this method will consider both position and color factors. If color of two data are similar, the distance between the two data will be closer.

```python
def c_k_mean_plus(data):

    mean=np.zeros((k,2))
    total=256 ** 2 * 3

    mean[0,0]=np.random.randint(low=0,high=100)
    mean[0,1]=np.random.randint(low=0,high=100)

    for u_num in range(1,k):
        all_distance=np.zeros((10000))

        for i in range(10000):
            x=i // 100
            y=i % 100
            this_dis=np.zeros(u_num)
            for j in range(u_num):
                this_dis[j]=((mean[j,0]-x) ** 2 + (mean[j,1]-y) ** 2) *total / ( (data[j,0]-data[int(mean[j,0]),0]) ** 2
                            + (data[j,1]-data[int(mean[j,1]),1]) ** 2 +(data[j,2]-data[int(mean[j,1]),2]) ** 2)

            all_distance[i]=this_dis[np.argmin(this_dis)]

        max_distance=np.argmax(all_distance)
        mean[u_num,0]=max_distance // 100
        mean[u_num,1]=max_distance %100

    rand=np.zeros((k))

    for i in range(k):
        rand[i]=mean[i,0] + mean[i,1] *100

    return rand
```

a.Spectral Cluster – Part1

1.

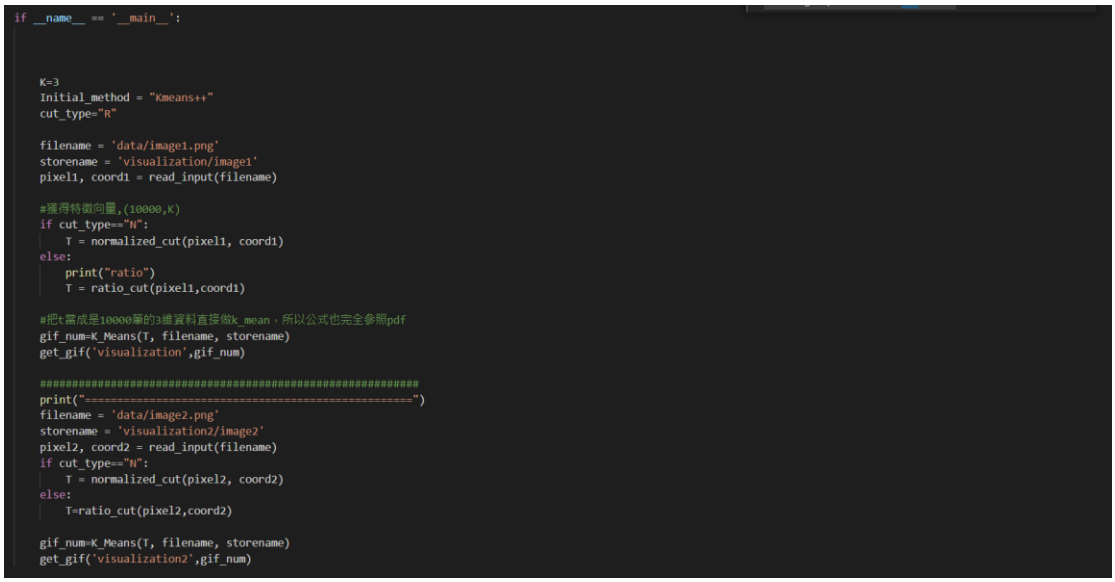
In the main, we can set ‘K’ for cluster number, “Initial_method” for initial way and “cut_type” for cut type .

If “cut_type” is “N”, it will do normalize cut. If “cut_type” is not “N”, it will do ratio cut.

First, it will do normalize_cut to get eigenvector matrix “T” which is 10000*K matrix.

Second, call “K_Means” function with input “T”(T will be treated as 10000 data with K dimension).

Finally, making gif.



```
if __name__ == '__main__':


    K=3
    Initial_method = "Kmeans++"
    cut_type="R"

    filename = 'data/image1.png'
    storename = 'visualization/image1'
    pixel1, coord1 = read_input(filename)

    #獲得特徵向量,(10000,K)
    if cut_type=="N":
        T = normalized_cut(pixel1, coord1)
    else:
        print("ratio")
        T = ratio_cut(pixel1,coord1)

    #把T當成是10000筆的3維資料直接做k_mean，所以公式也完全參照pdf
    gif_num=K_Means(T, filename, storename)
    get_gif('visualization',gif_num)

    ############################################################
    print("============================================")
    filename = 'data/image2.png'
    storename = 'visualization2/image2'
    pixel2, coord2 = read_input(filename)
    if cut_type=="N":
        T = normalized_cut(pixel2, coord2)
    else:
        T=ratio_cut(pixel2,coord2)

    gif_num=K_Means(T, filename, storename)
    get_gif('visualization2',gif_num)
```

2.normalize cut

(I use np.save to save time.)

(Normalize cut formula come from the lecture.)

First, compute Lsym according to the formula.

Second, find eigen vector and eigen value of Lsym by using np.linalg.eig().

Then, choose number 1~k eigen vector as new matrix U (not choose number 0 eigen vector).

Finally, comput matrix T (normalize the row of U to norm1), and return it.

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

Normalized spectral clustering according to Ng Jordan and Weiss (2002)

Input: Similarity matrix $S \in \mathbb{R}^{n \times n}$, number k of clusters to construct.

- 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_{sym} $D^{-1/2} L D^{-1/2}$
- **Compute the first k eigenvectors $u_1, \ldots, u_k$ of L_{sym}.**
- Let $U \in \mathbb{R}^{n \times k}$ be the matrix containing the vectors $u_1, \ldots, u_k$ as columns.
- **Form the matrix $T \in \mathbb{R}^{n \times k}$ from U by normalizing the rows to norm 1,** that is set $t_{ij} = u_{ij} / (\sum_k u_{ik}^2)^{1/2}$.
- For $i = 1, \ldots, n$, let $y_i \in \mathbb{R}^k$ be the vector corresponding to the i-th row of T.
- Cluster the points $(y_i)_{i=1,\ldots,n}$ with the k-means algorithm into clusters $C_1, \ldots, C_k$.

Output: Clusters $A_1, \ldots, A_k$ with $A_i = \{j | \, y_j \in C_i\}$.

```python
def normalized_cut(pixel, coord):

    if filename == 'data/image1.png':
        print("img1")
        try:
            W=np.load('W.npy')
            D=np.load('D.npy')
            D_root=np.load('D_root.npy')
            L_sym=np.load('L_sym.npy')
            eigen_values=np.load('eigen_values.npy')
            eigen_vectors=np.load('eigen_vectors.npy')
        except:
            W = compute_kernel(pixel, coord)
            D = np.diag(np.sum(W, axis=1))
            D_root = np.diag(np.power(np.diag(D), -0.5))
            L_sym = np.eye(W.shape[0]) - D_root @ W @ D_root
            eigen_values, eigen_vectors = np.linalg.eig(L_sym)
            np.save('W', W)
            np.save('D', D)
            np.save('D_root', D_root)
            np.save('L_sym',L_sym)
            np.save('eigen_values',eigen_values)
            np.save('eigen_vectors',eigen_vectors)
    elif filename == 'data/image2.png':
        print("img2")
        try:
            W=np.load('W2.npy')
            D=np.load('D2.npy')
            D_root=np.load('D_root2.npy')
            L_sym=np.load('L_sym2.npy')
            eigen_values=np.load('eigen_values2.npy')
            eigen_vectors=np.load('eigen_vectors2.npy')
        except:
            W = compute_kernel(pixel, coord)
            D = np.diag(np.sum(W, axis=1))
```

```
            W=np.load('W2.npy')
            D=np.load('D2.npy')
            D_root=np.load('D_root2.npy')
            L_sym=np.load('L_sym2.npy')
            eigen_values=np.load('eigen_values2.npy')
            eigen_vectors=np.load('eigen_vectors2.npy')
        except:
            W = compute_kernel(pixel, coord)
            D = np.diag(np.sum(W, axis=1))
            D_root = np.diag(np.power(np.diag(D), -0.5))
            L_sym = np.eye(W.shape[0]) - D_root @ W @ D_root
            eigen_values, eigen_vectors = np.linalg.eig(L_sym)

            np.save('W2', W)
            np.save('D2', D)
            np.save('D_root2', D_root)
            np.save('L_sym2',L_sym)
            np.save('eigen_values2',eigen_values)
            np.save('eigen_vectors2',eigen_vectors)

    K_eigen_v = np.argsort(eigen_values)[1: K+1]
    U = eigen_vectors[:, K_eigen_v].real.astype(np.float32)


    # normalized
    normalize_sum = np.power(U, 2)
    normalize_sum = np.sum(normalize_sum, axis=1) ** 0.5
    normalize_sum = normalize_sum.reshape(-1, 1)


    T = U.copy()
    for i in range(normalize_sum.shape[0]):
        if normalize_sum[i][0] == 0:
            normalize_sum[i][0] = 1
        T[i][0] /= normalize_sum[i][0]
        T[i][1] /= normalize_sum[i][0]
    return T
```

3.ratio cut

(I use np.save to save time.)

(Ratio cut formula come from the lecture.)

First, compute L according to the formula.

Second, find eigen vector and eigen value of L by using np.linalg.eig().

Then, choose number 1~k eigen vector as new matrix U (not choose number 0 eigen vector).

Finally, return matrix U.

$$L = D - W$$

```
def ratio_cut(pixel, coord):
    if filename == 'data/image1.png':
        print("img1")
        try:
            W_r=np.load('W_r.npy')
            D_r=np.load('D_r.npy')
            L_r=np.load('L_r.npy')
            eigen_values_r=np.load('eigen_values_r.npy')
            eigen_vectors_r=np.load('eigen_vectors_r.npy')
        except:

            W_r = compute_kernel(pixel, coord)
            D_r = np.diag(np.sum(W_r, axis=1))
            L_r = D_r - W_r
            eigen_values_r, eigen_vectors_r = np.linalg.eig(L_r)

            np.save('W_r',W_r )
            np.save('D_r', D_r)
            np.save('L_r', L_r)
            np.save('eigen_values_r',eigen_values_r)
            np.save('eigen_vectors_r',eigen_vectors_r)
    elif filename == 'data/image2.png':
        print("img2")
        try:
            W_r=np.load('W_r2.npy')
            D_r=np.load('D_r2.npy')
            L_r=np.load('L_r2.npy')
            eigen_values_r=np.load('eigen_values_r2.npy')
            eigen_vectors_r=np.load('eigen_vectors_r2.npy')
        except:
```

```
        except:

            W_r = compute_kernel(pixel, coord)
            D_r = np.diag(np.sum(W_r, axis=1))
            L_r = D_r - W_r
            eigen_values_r, eigen_vectors_r = np.linalg.eig(L_r)

            np.save('W_r2',W_r )
            np.save('D_r2', D_r)
            np.save('L_r2', L_r)
            np.save('eigen_values_r2',eigen_values_r)
            np.save('eigen_vectors_r2',eigen_vectors_r)



    idx = np.argsort(eigen_values)[1: K+1]
    U = eigen_vectors[:, idx].real.astype(np.float32)

    return U
```

4.Kmean

First, use “initial” function to do initial. (please see part 3 for more explanation of initial).

Second,keep doing E_step and M_step unitil it converge

Finally, call “draw_eigenspace” function to draw eigen space. (please see part 4 for more explanation of drawing eigen space).

```python
def K_Means(data, filename, storename):

    initial_method=Initial_method
    center, mean, ri = initial(data, initial_method)
    mu=np.zeros((K,K))
    print(center)

    for u_num in range(K):
        mu[u_num,:]=data[int(center[u_num]),:]

    print(mu)
    error = -10000
    prev_error = -10001

    iteration = 0
    while(iteration <= epochs):
        iteration += 1
        prev_ri = ri


        #Estep
        ri = E_step(data, mu)
        visualization(ri,iteration,initial_method)
        #Mstep
        mu = M_step(data, mu, ri)

        error = calculate_error(ri, prev_ri)
        print(error)
        if error == prev_error:
            break
        prev_error = error

    draw_eigenspace(filename, storename, iteration, ri, initial_method, data)
    return iteration
```

5.E_step

Use mu to caculate new “ri” and return it.

(The formula come from the lecture.)

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

E-step

```python
def E_step(data, mu):
    ri = np.zeros((10000), dtype=np.int)
    for dataidx in range(10000):
        distance = np.zeros(K, dtype=np.float)
        for cluster in range(K):

            minus=data[dataidx,:]- mu[cluster,:]
            minus=abs(minus)
            distance[cluster]=np.square(minus).sum(axis=0)

        ri[dataidx] = np.argmin(distance)

    return ri
```

6.M_step

(The formula come from the lecture.)

Use ri to compute new mu and return it.

$$\mu_k = \frac{\sum_n r_{nk} x_n}{\sum_n r_{nk}}$$ M-step

```python
def M_step(data, mu, ri):

    new_mu = np.zeros(mu.shape, dtype=np.float32)
    total = np.zeros(mu.shape, dtype=np.int)
    plus = np.ones(mu.shape[1], dtype=np.int)

    for dataidx in range(10000):
        new_mu[ri[dataidx]] += data[dataidx]
        total[ri[dataidx]] += plus

    #防止zero devide
    for i in range(K):
        if total[i][0] == 0:
            total[i] += plus

    new_mu=new_mu / total

    return new_mu
```

7.converge condition

this function compute the different between new ri and previous ri。

```python
def calculate_error(ri, prev_ri):
    error = 0
    for i in range(10000):
        if ri[i] != prev_ri[i]:
            error+=1

    return error
```

8.gif

This function will generate gif.

```
def get_gif(pics_dir,n):
    imgs = []
    files = os.listdir(pics_dir)
    for i in files:
        pic_name = os.path.join(pics_dir, i)
        print(pic_name)
        temp = Image.open(pic_name)
        imgs.append(temp)

    save_name =storename+"k_"+str(K)+"ini_method_"+Initial_method + '{}.gif'.format(pics_dir)

    imgs[0].save(save_name, save_all=True, append_images=imgs, duration=400)
    return save_name
```

a.Spectral Cluster – Part2

we can change k to change cluster number here.

```
if __name__ == '__main__':



    K=3
    Initial_method = "Kmeans++"
    cut_type="R"
```

a.Spectral Cluster – Part3

initial function will generate k center and return it.

“Random” method will randomly choose k data as center and return it.
“Kmeans++” method will use kmean++ algorithm to choose center.

(k-mean++ algo. was found on the website.)
However, I found Kmeans++ is sometime Not doing well if we just choose extreme value(furthest point) when we are choosing a new center. So I used probability distribution combined with K-means++ instead of just choosing the Furthest point as center. And this distribution is according to Euclidean distance (higher distance, higher probability).
(I will discuss this part more detail in ***c. observations and discussion***)

```python
def initial(data, initial_method):
    prev_ri = np.random.randint(K, size=data.shape[0])

    if initial_method == 'random':
        mu=np.zeros((K,K))
        rand = np.random.randint(10000, size=K)

        for u_num in range(K):
            this_data=data[rand[u_num],:]
            for dim in range(K):
                mu[u_num][dim]=this_data[dim]

        return rand,mu, prev_ri
    elif initial_method == 'Kmeans++':
        mu=np.zeros((K,K))
        rand = np.random.randint(10000, size=K)



        center=[np.random.choice(range(10000))]
        centroids = [data[center[-1], :]]
        for i in range(K - 1):
            dist = scipy.spatial.distance.cdist(data, centroids, 'euclidean').min(axis=1)
            prob = dist / np.sum(dist)
            center.append(np.random.choice(range(10000), p=prob))
            centroids.append(data[center[-1]])

        centroids = np.array(centroids)
        return center, centroids, prev_ri
```

a.Spectral Cluster – Part4

I plot first three column of the matrix T(which is 10000 * K matrix) to 3D space .
And plot them as different image(circle, triangle, square) according to “ri”.

```python
def draw_eigenspace(filename, storename, iteration, ri, initial_method, data):

    fig=plt.figure()
    ax=fig.add_subplot(111,projection='3d')
    markers=['o','^','s']

    for marker,i in zip(markers,np.arange(3)):
        ax.scatter(data[ri==i,0],data[ri==i,1],data[ri==i,2],marker=marker)

    ax.set_xlabel('1st dim')
    ax.set_ylabel('2nd dim')
    ax.set_zlabel('3rd dim')
    plt.show()
```

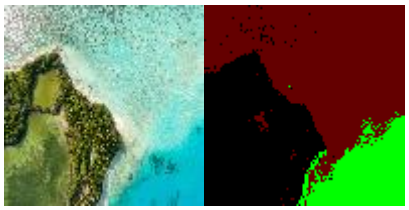
b. experiments settings and results

b. Kernel k-mean – Part1

k=3,

initial_method= random

(all of these have .gif file)

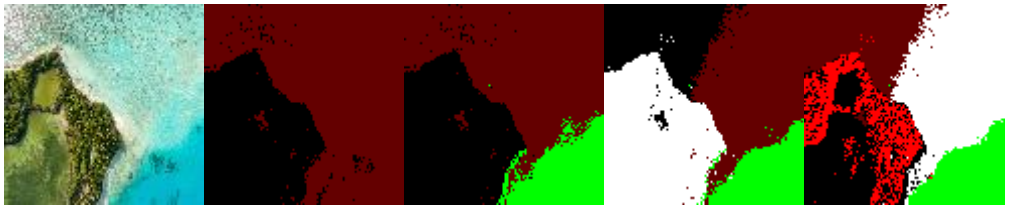


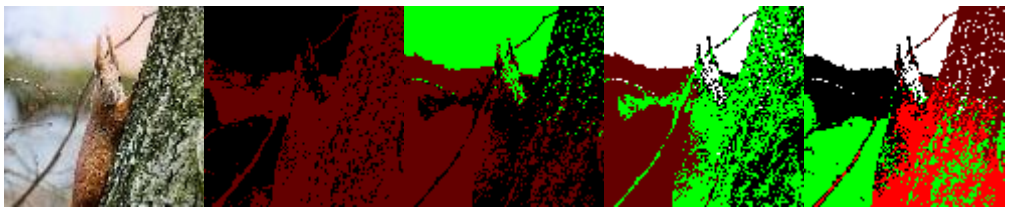
b. Kernel k-mean – Part2

initial_method: random

image > 2 clusters > 3 clusters > 4 clusters > 5 clusters

(all of these have .gif file)



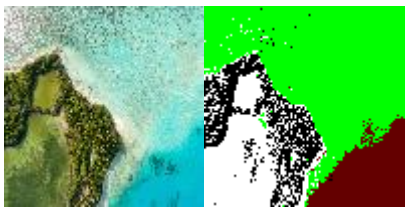


b. Kernel k-mean – Part3

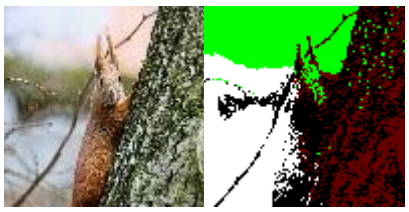
K=4

I also show which data I choose as center data by the initial method.
(all of these have .gif file)

Initial method: Random:

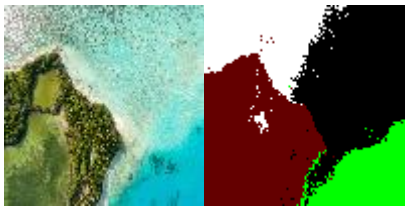


Initial Center data: [5617, 7091, 1027, 9627]
Center 1 index: 56, 17
Center 2 index: 70, 91
Center 3 index: 10, 27
Center 4 index: 96, 27



Initial Center data: [4476, 5499, 222, 4258]
Center 1 index: 44, 76
Center 2 index: 54, 99
Center 3 index: 2, 22
Center 4 index: 42, 58

Initial method: Kmean++:

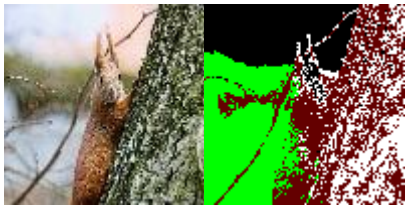


Initial Center data: [1768, 9900, 9999, 0.]
Center 1 index:17.0, 68.0

Center 2 index:99.0, 0.0

Center 3 index: 99.0, 99.0

Center 4 index: 0.0, 0.0



Initial Center data: [819., 9999., 9900., 1899.]

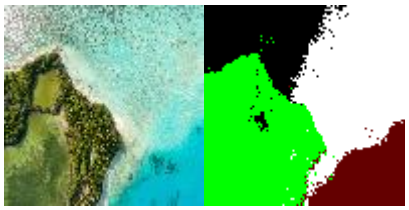
Center 1 index: 8.0, 19.0

Center 2 index: 99.0, 99.0

Center 3 index: 99.0, 0.0

Center 4 index: 18.0, 99.0

Initial method: c-Kmean++:



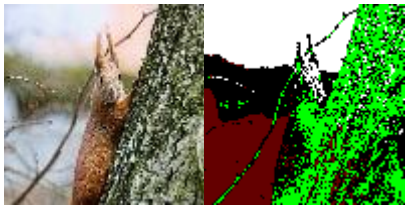
Initial Center data: [1425., 9999., 9968., 3499.]

Center 1 index: 14.0, 25.0

Center 2 index: 99.0, 99.0

Center 3 index: 99.0, 68.0

Center 4 index :34.0, 99.0



Initial Center data: [955., 9900., 99., 33.]

Center 1 index: 9.0, 55.0

Center 2 index: 99.0, 0.0

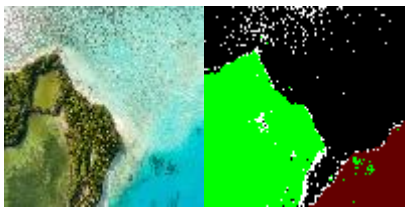
Center 3 index: 0.0, 99.0

Center 4 index: 0.0, 33.0

b. Spectral Cluster – Part1

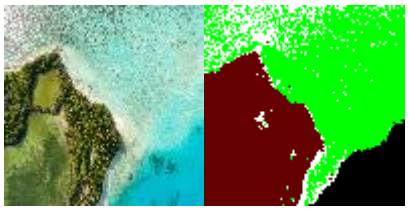
Normalize Cut

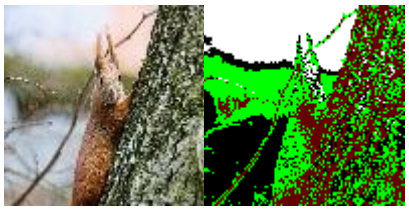
K=4, random



Ratio cut

K=4 random



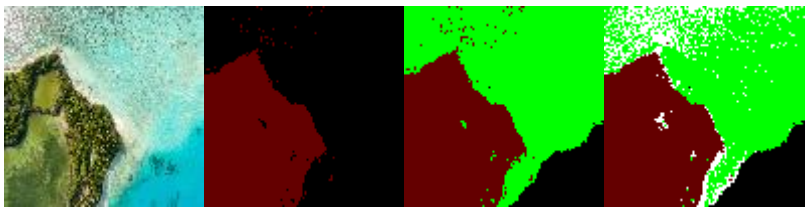


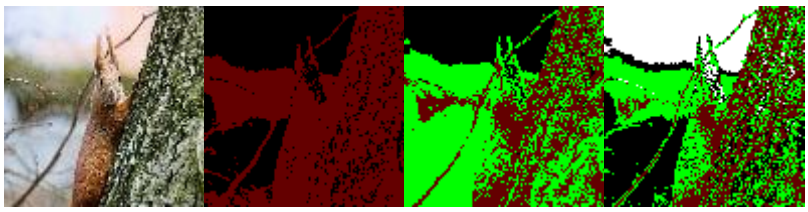
b. Spectral Cluster – Part2

K=2、3、4　random

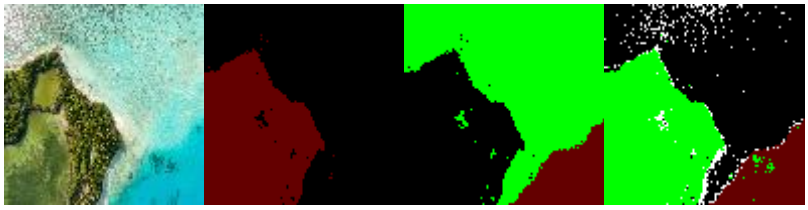
(all of these have .gif file)

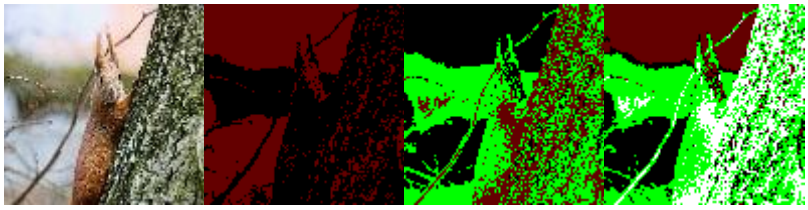
Normalize cut





Ratio cut





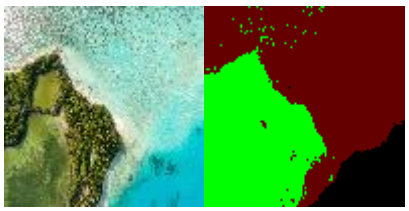
b. Spectral Cluster – Part3

Normalize cut

k=3

(all of these have .gif file)

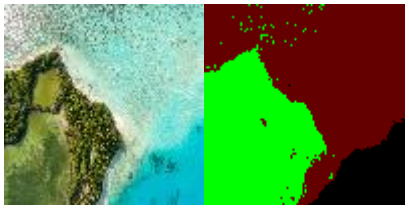
Initial method: Random



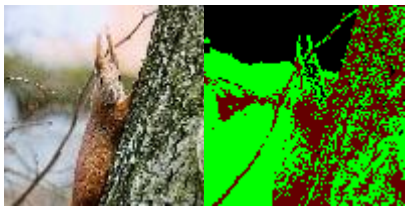
Initial Center data: [9184, 6876, 7338]

Initial Center data:[5153, 9480, 907]

Initial method: Kmean++



Initial Center data: [3925, 5280, 7618]

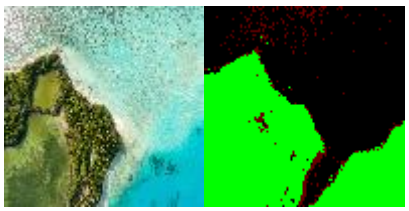


Initial Center data: [58, 1191, 8929]

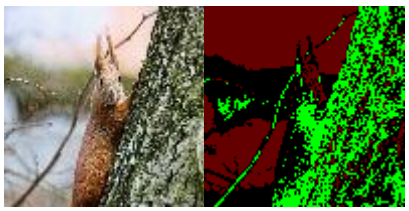
Ratio cut

k=3

Initial method: Random

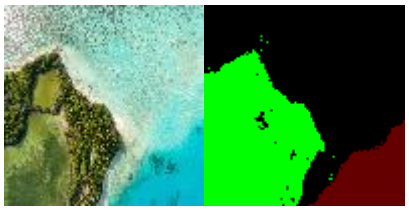


Initial Center data:[3852, 4879, 3131]

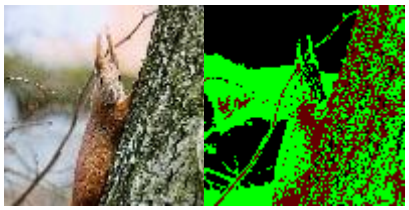


Initial Center data: [3517, 1751, 5513]

Initial method: Kmean++



Initial Center data: [675, 9264, 4503]



Initial Center data: [1239, 6373, 7203]

b. Spectral Cluster – Part4

I set k=3. And plot matrix T (10000*3) to 3D space according to the “ri”.
We can see the data points within the same cluster do have the same coordinates in the eigenspace of graph Laplacian.

Normalize cut

image 1

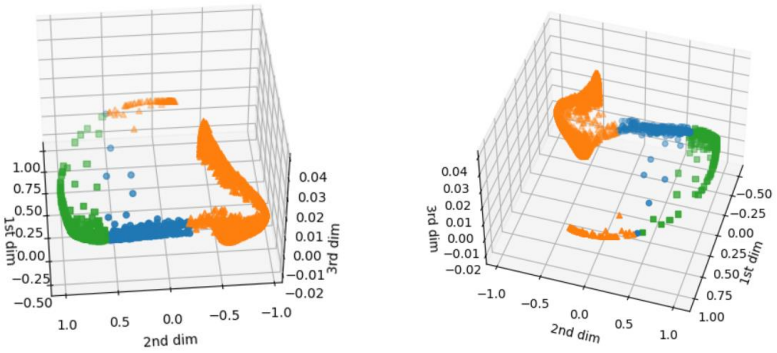
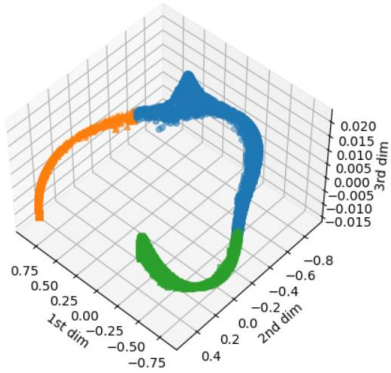
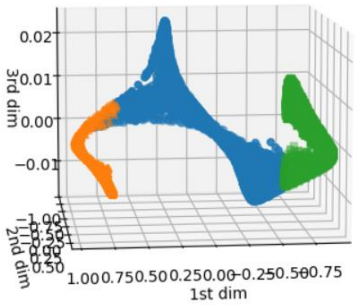


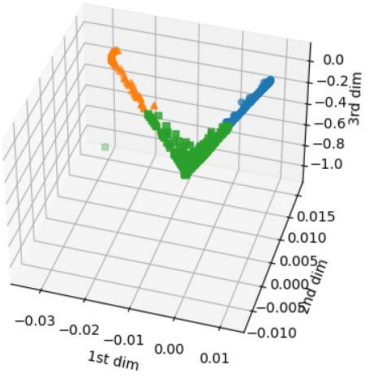
image 2





Ratio cut

image 1



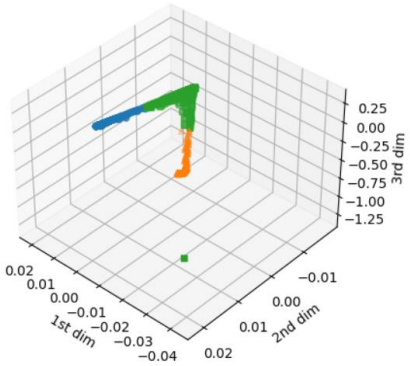
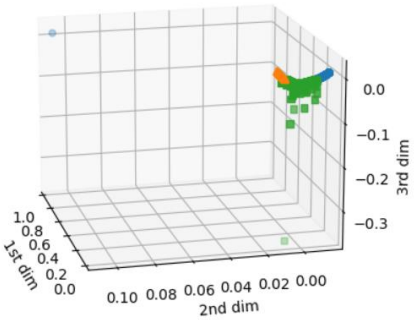
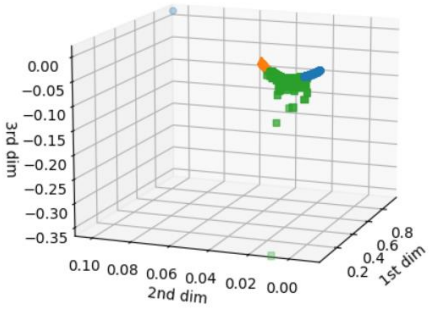


image 2

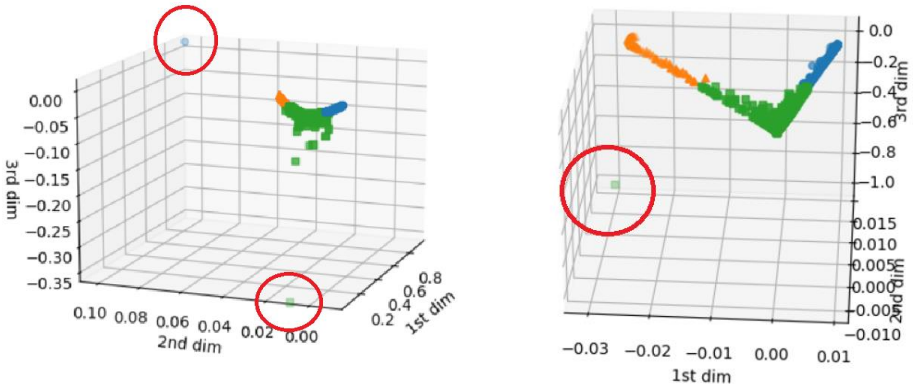




c.observations and discussion

I found that kernel k-mean is Influenced by initial center more easily than spectral cluster. Kernel k-mean with “random” initial method sometime work Not well, so “k-mean++” initial method is a good choice.

However, in spectral cluster with Ratio Cut, If I use k-means++, it still work Not well. After observation, I found that the problem maybe here (please see picture below).



We can found that there are some extreme point in these 3D data points . In “Kmeans++” algo. , we always choose the extreme point, so we will always choose these weird point as center. (and I guess it is the cause of the bad result).

Therfore, I made some small change to Kmeans++ algo. . In each iteration in kmean++, instead of choosing extreme point, I choose the data point in probability distribution. And this distribution is according to Euclidean distance (higher distance, higher probability).