Machine Learning HW6

313553007 楊宜祥

1. Code with detailed explanations

I. Part1: Clustering & Visualization

Main program: part1()

For part 1, I run the code below. I use setting s: 1.0, c:1.0, num_cluster = 2. Respectfully, run Kmeans, Spectral Clustering in ratio cut & normalized cut mode for 2 images.

```
def part1():
    global IMG
    output_folder = './part1'
    for image_pth in ("image1.png", "image2.png"):
        IMG = Path(image_pth).stem
        launch_process(image_pth=image_pth, s=1.0, c=1.0, model=Kmeans, num_cluster=2, output_folder =output_folder)      You, 28 分鐘前 •
        launch_process(image_pth=image_pth, s=1.0, c=1.0, model=Spectral, num_cluster=2, mode='ratio', output_folder =output_folder)
        launch_process(image_pth=image_pth, s=1.0, c=1.0, model=Spectral, num_cluster=2, mode='normalized', output_folder =output_folder)
```

launch_process()

This function first parsed model arguments to initialize the model. Then get the features from the current read images. Execute *record_process()* to start recording. Start fitting the data. Finally, release the recorded video. If the model is Spectral and the number of cluster equals to 2 or 3, it'll draw the coordinate in eigenvector coordinates and save it.

```
def launch_process( image_pth:str,
                    s:float, c:float,
                    model:CLUSTER_SET,
                    num_cluster:int,
                    mode=None, init_method='random',
                    output_folder=''):
    model_args = {
        'num_cluster': num_cluster,
        'mode':mode,
        'init_method':init_method
    }
    model = model(**model_args)
    im = cv2.imread(image_pth)
    feature = get_feature(im)
    Path(output_folder).mkdir(exist_ok=True, parents=True)
    video_fpth = str(
        Path(output_folder) / Path(f"{Path(image_pth).stem}_{model}_{init_method}_num_cluster:{model.num_cluster}_s:{s}_c:{c}.mp4")
    )
    model.record_process(video_fpth=video_fpth, image=im)
    guess, W = model.fit(feature, s, c,)
    model.release_record()
    if model.__class__ is Spectral and num_cluster <=3:
        fname = f"{Path(image_pth).stem}_{model}_{init_method}_num_cluster:{num_cluster}_s:{s}_c:{c}.png"
        out_fpth = Path(output_folder) / Path(fname)
        draw_eigenvector(str(out_fpth), guess, W)
```

Other functions description

get_feature()

The extracted features contain pixels' coordinates and their colors. Use *np.meshgrid()* to get image-shape coordinates. Here, I normalize the value of

coordinates and colors to [0, 1]. Finally, concatenate those 2 features, to N*5 matrix.

```
def get_feature(image):
    row, col, ch = image.shape
    r, c = np.meshgrid(np.arange(row), np.arange(col), indexing='ij')
    space = np.stack((r,c), axis=2).reshape(-1, 2)
    space = space / np.max(space)
    color = image.reshape(-1, 3) /255 # normalized
    feature = np.concatenate([space, color], 1)
    return feature
```

gram_matrix_exp()

This function defines the kernel function and get the similarity matrixes, W. Here, I use the specified kernel function to calculate the kernel value:

$$K(x,\ x') \ = \ e^{-s\ ||Coordinate(x) - Coordinate(x')||^2} \times e^{-\gamma_c\ ||Color(x) - Color(x')||^2}$$

```
def gram_matrix_exp(feature1, feature2,  s, c):      You, 44 分鐘前
    '''
    s: parameter of spatial information\n
    c: parameter of color information
    '''
    space1, color1 = feature1[:,:2], feature1[:,2:]
    space2, color2 = feature2[:,:2], feature2[:,2:]
    return np.exp(-s*(cdist(space1, space2, 'euclidean')**2)) * \
           np.exp(-c*(cdist(color1, color2, 'euclidean')**2))
```

gram_matrix_dist()

This function will be used in Spectral.init_guess(). Return the pairwise distance between two 2D matrices.

```
def gram_matrix_dist(coor1, coor2):
    return cdist(coor1, coor2, 'euclidean')
```

Class definition: Cluster()

Both *Kmeans* & *Spectral* inherit this class. This class mainly define some utils functions and structure for clustering algorithm.

record_process() & release_record()

These 2 define the video writer and related variables. Here use mp4 as the video format.

class attribute: guess()

Here define the variable, guess. This variable stored the temporary predictions for clustering. If *self.RECORD* is true, every time update *self.guess*, video writer will write the current result to the video. This part avoids the redundant code in *clustering_W()*.

terminate_condition()

This part define the condition of early stopping the clustering.

Meet …

condition 1: # (guess(t) != guess(t-1)) / N < 0.05 and iteration times>40, N: # pixels.

or

condition 2: guess(t) == guess(t-1)

```python
class Cluster:      You, 前天 • hw6:Finish Kmeans
    def __init__(self, num_cluster, init_method:str):
        self.num_cluster:int = num_cluster
        self.init_method:str = init_method
        self._guess = None
    def record_process(self, video_fpth:str, image):
        self.RECORD = True
        self.image_template = image
        self.__colors = [ (0, 0, 255), (0,255,0), (255,0,0), (200,200,0), (0,200,200), (205,0,205)]
        fourcc = cv2.VideoWriter_fourcc(*'mp4v')
        im_shape = image.shape[:2]
        self.__record_video = cv2.VideoWriter(video_fpth, fourcc, 30.0, im_shape)

    def release_record(self):
        self.RECORD =False
        self.__record_video.release()

    @property
    def guess(self):
        return self._guess

    @guess.setter
    def guess(self, value):
        self._guess = value
        if hasattr(self, "RECORD") and self.RECORD:
            # create coordinates.
            im = np.copy(self.image_template)
            im_shape = im.shape[:2]
            for cluster, color in zip(range(self.num_cluster), self.__colors):
                interest = (value == cluster).reshape(im_shape)
                im = np.where( np.repeat(interest[..., np.newaxis], 3, axis=2),
                               0.3* im + 0.7 * np.array(color),
                               im)
            self.__record_video.write(im.astype(np.uint8))

    def terminate_condition(self, tmp_guess, iter):
        N = len(self.guess)
        return np.sum(tmp_guess != self.guess)/N < 0.05 and iter > MAX_ITERATION*0.8 or \
               np.sum(tmp_guess != self.guess) ==0
```

clustering(), get_W(), init_guess(), fit()

These functions should be defined in the children classes.

```python
def clustering(self, W):        You,
    raise NotImplementedError()

def get_W(self, s, c ):
    raise NotImplementedError()

def init_guess(self, N, **kwargs):
    raise NotImplementedError()

def fit(self, feature, s, c):
    raise NotImplementedError()
```

Class definition: Kmeans()

fit() & get_W()

get_W() : Get the simlarity matrix. Here, use the defined kernel.

fit(): Interface for fitting data.

```python
def get_W(self, feature:np.ndarray, s, c):
    '''args: s, c'''
    W = gram_matrix_exp(feature, feature, s, c)
    return W

def fit(self, feature, s, c, **kwargs):
    W = self.get_W(feature, s, c )
    self.init_guess(N =len(W), feature=feature, s=s, c=c)
    result = self.clustering(W)
    return result, W
```

init_guess()

It defines how Kmeans() function initializesthe guess attribute. Include ‘random’ and ‘kmeans++’ modes. For the mode, ‘kmeans++’, it’ll first randomly select a pixel as the first centroid. Then, it’ll calculate distances between those recorded temporary centroids and all pixels. This distance will affect the probability of pixels being selected as the next centroid. Extract centroids until it meet the requirement. This method helps to get the centroids far from each other. Finally, calculate the initial guess via these centroids.

Here, use the defined kernel as the evaluation metrics.

```python
def init_guess(self, N, feature, s, c):
    if self.init_method == 'random':
        self.guess = np.random.randint(0, self.num_cluster, size=N)
    elif self.init_method == 'kmeans++':
        N, col = feature.shape
        # Get centroids
        centroids = [feature[np.random.choice(np.arange(N), size=1)].ravel()]
        for i in range(1, self.num_cluster):
            dist_centroids = gram_matrix_exp( feature, np.array(centroids).reshape(-1, col), s=s, c=c).min(axis=1)
            prob = dist_centroids / np.sum(dist_centroids)
            centroids.append(feature[np.random.choice(np.arange(N), p=prob), :])
        # Update Guess based on centroids
        self.guess = gram_matrix_exp(feature, np.array(centroids).reshape(-1,col), s=s, c=c).argmin(axis=1)
    else:
        raise AssertionError("Wrong init_method.")
```

clustering()

This part, it mainly try to find the best clusters for each pixel so as to get the minimum objective function**.** During the process, it'll do…

1. Calculate the objective value for each data of input to all clusters.
2. Assign the best cluster for each data.

Objective function:

$$argmin_{(C_1,\mu_1),\ldots,(C_k,\mu_k)} \sum_{i=1}^{k} \sum_{x_j \in C_i} \left|\left| x_j - \mu_i \right|\right|^2 \ldots(1)$$

$$\left|\left| \phi(x_j) - \mu_k^{\phi} \right|\right| = K(x_j, x_j) - \frac{2}{|C_k|} \sum_{n \in cluster\ k} K(x_j, x_n) + \frac{1}{|C_k|^2} \sum_{x_q \in C_k} \sum_{x_p \in C_k} K(x_p, x_q) \ldots(2)$$

Based on (2), the first term is the kernel value of the data itself, corresponding to *self_W*, which gets the diagonal value of W. This value won't change during whole iterations.

For each iteration, calculate *to_all_cluster*, corresponding to the 2nd term, and *cluster_W*, corresponding to the 3rd term, for each cluster.

self_W: N*1 matrix.

to_all_cluster: N*k matrix. k: number of clusters.

cluster_W: 1*k matrix.

After calculating the value for each cluster, it'll aggregate them to the objective value, N*k matrix, and use *np.argmin()* to get the index of the best suitable cluster for each pixel. Update self.guess. It'll simultaneously record the temporary results.

```python
def clustering(self, W):      You, 8 分鐘前 • Uncommitted changes
    N = len(W)
    self_W = np.diag(W)[:, None]
    for iter in tqdm(range(MAX_ITERATION), desc="Kernel Kmeans: "):
        # * M step: Calculate objective function
        cluster_W = np.zeros([1,self.num_cluster]) # Save for summation of each cl
        to_all_cluster_W = np.zeros((N, self.num_cluster)) # (N, num_cluster): Sav
        for i in range(self.num_cluster):
            num_data = np.sum(self.guess == i )
            indexes = np.where( self.guess == i)[0]
            to_all_cluster_W[:, i]  = np.sum(W[:, indexes], axis=1) *2 / num_data
            cluster_W[0, i]= np.sum( W[indexes,:][:, indexes]) / (num_data**2)

        # * E step: Assign each clusters
        new_guess =np.argmin(self_W - to_all_cluster_W + cluster_W , axis=1)
        if self.terminate_condition(new_guess, iter): break
        self.guess = new_guess
    self.guess = new_guess
    return self.guess
```

Class: Spectral()

init_guess()

This function is almost the same with Kmeans.init_guess() except for distance metrics. This part use simple euclidean distance as the evaluation metrics.

```python
class Spectral(Cluster):

    def __init__(self, num_cluster, mode:str, init_method='random', *args, **kwargs,):
        super().__init__(num_cluster, init_method)
        self.mode = mode # * 'normalized' '| 'ratio'

    def init_guess(self, N, evector:np.ndarray = None):
        if self.init_method == 'random':
            self.guess = np.random.randint(0, self.num_cluster, size=N)
        elif self.init_method == 'kmeans++' and evector is not None:
            N, col = evector.shape
            # Get centroids
            centroids = [evector[np.random.choice(np.arange(N), size=1)].ravel()]
            for i in range(1, self.num_cluster):

                dist_centroids = gram_matrix_dist( evector ,np.array(centroids).reshape(-1, col)).min(axis=1)
                prob = dist_centroids / np.sum(dist_centroids)

                centroids.append(evector[np.random.choice(np.arange(N), p=prob), :])
            # Update Guess based on centroids
            self.guess = gram_matrix_dist(evector, np.array(centroids).reshape(-1, col)).argmin(axis=1)
        else:
            raise AssertionError("Wrong init_method.")
```

get_W()

This function mainly defines how to get a similarity matrix, W, for normalized cuts & ratio cuts. Both of the modes will calculate the similarity matrix and degree matrix. Then calculate the eigenvectors of the Laplacian matrix or normalized Laplacian matrix.

I add some features to dump the temporary results to .npy so as to accelerate the whole process.

Prevent values being complex matrix, I add some condition to filter out unwanted elements and get the real parts of them. Finally, filter out those vectors with eigenvalues close to 0, and get a certain number of eigenvectors to compose the H matrix.

```python
def get_W(self, feature, s, c):
    global IMG
    W = gram_matrix_exp(feature, feature, s, c)
    D = np.diag(np.sum(W, axis=1))
    L = D - W

    if Path(f'{IMG}_{self.mode}_{s}_{c}_evector.npy').exists() and \
        Path(f'{IMG}_{self.mode}_{s}_{c}_evalue.npy').exists():
        evector = np.load(f'{IMG}_{self.mode}_{s}_{c}_evector.npy')
        evalue = np.load(f'{IMG}_{self.mode}_{s}_{c}_evalue.npy')
    else:
        if self.mode == 'ratio':
            evalue, evector = np.linalg.eig(L)
        elif self.mode == 'normalized':
            D_inv = np.divide(1, D, out=np.zeros_like(D), where= D > 1e-4)
            evalue, evector = np.linalg.eig( D_inv @ L )
        np.save(f'{IMG}_{self.mode}_{s}_{c}_evector.npy' , evector)
        np.save(f'{IMG}_{self.mode}_{s}_{c}_evalue.npy', evalue)
    close_real = np.isclose(np.imag(evalue), 0)         # Chose eigen valu
    evalue = evalue[close_real].real
    evector =evector[:, close_real].real
    # Treat those eigen value < 1e-4 ~= 0.0, which is out of constraint. F
    evalue_large_enough = ~np.isclose(evalue, 0)
    evalue = evalue[evalue_large_enough]
    evector = evector[:, evalue_large_enough]
    indx = np.argsort(evalue)
    H = evector[..., indx[:self.num_cluster]]
    return H
```

clustering() & get_centroids()

For clustering, use the traditional K-means algorithm to get the updated guess. Treating each row vector of H as the coordinate of data, we can get centroids of each cluster and reassign each data to the most suitable one.

```
def clustering(self, H):
    for iter in tqdm(range(MAX_ITERATION), desc=f"Spectral Clustering-{self.mode}"):
        centroids = self.get_centroids(H)
        distance = gram_matrix_dist(H, centroids)
        new_guess = np.argmin(distance, axis=1)
        if self.terminate_condition(new_guess, iter): break
        self.guess = new_guess
    self.guess = new_guess

def get_centroids(self, H):
    centroids = []
    for i in range(self.num_cluster):
        ex_indx = np.where(self.guess == i)[0]
        centroids.append(np.mean(H[ex_indx, :], axis=0))
    return np.array(centroids)
```

II. Part2: Try more clusters.

This part, I set the range of clusters [2, 4]. Try to check different clustering results for 3 different models.

```
def part2():
    global IMG
    output_folder = './part2'
    for image_pth in ("image1.png", "image2.png"):
        IMG = Path(image_pth).stem
        for nc in range(2, 5):      You, 3 秒前 • Uncommitted changes
            launch_process(image_pth=image_pth, s=1.0, c=1.0, model=Kmeans, num_cluster=nc, output_folder =output_folder)
            launch_process(image_pth=image_pth, s=1.0, c=1.0, model=Spectral, num_cluster=nc, mode='ratio', output_folder=output_folder)
            launch_process(image_pth=image_pth, s=1.0, c=1.0, model=Spectral, num_cluster=nc, mode='normalized', output_folder=output_folder)
```

III. Part3: Try different initializations

This part will try initialized methods, kmeans++ and random for 3 different models.

```
def part3():
    global IMG
    output_folder = './part3'
    for image_pth in ('image1.png', "image2.png"):
        IMG = Path(image_pth).stem
        for init_method in ('kmeans++', 'random'):
            launch_process(image_pth=image_pth, s=.4, c=1.0, model=Kmeans, num_cluster=2, init_method=init_method, output_folder=output_folder)      You, 38 秒前 • Un
            launch_process(image_pth=image_pth, s=1.0, c=1.0, model=Spectral, num_cluster=2, mode='ratio', init_method=init_method, output_folder=output_folder)
            launch_process(image_pth=image_pth, s=1.0, c=1.0, model=Spectral, num_cluster=2, mode='normalized', init_method=init_method, output_folder=output_folder)
```

IV. Part4: Experiments on the coordinates in the eigenspace

draw_eigenvector()

From previous parts, it will generate the eigenvectors plots for those results. I use this function to draw 2D scatter plot (num_cluster=2) and 3D scatter plots (num_cluster =3). Mainly use matplotlib to achieve this feature.

```
def draw_eigenvector( out_fpth, guess, W):
    colors = [ (0, 0, 255), (0,255,0), (255,0,0)]
    num_cluster = len(np.unique(guess))
    fig = plt.figure()
    ax = fig.add_subplot( projection='3d')

    for i, color in zip(range(num_cluster), colors):
        ex_row = np.where(guess == i)[0]
        ex_coor = W[ex_row, :]
        x = ex_coor[:, 0]
        y = ex_coor[:, 1]
        if ex_coor.shape[1] == 2:
            z = np.zeros_like(x)
        elif ex_coor.shape[1] == 3:
            z = ex_coor[:, 2]

        ax.scatter(x, y, z, color, label=f'cluster {i}' )
    plt.legend()
    plt.xlabel('x')
    plt.ylabel('y')
    plt.title("Eigenvector coordinate")
    plt.savefig(out_fpth)
```

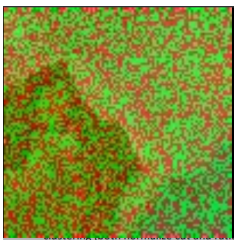
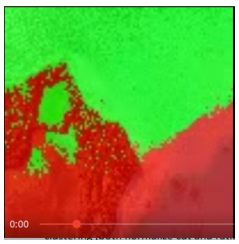
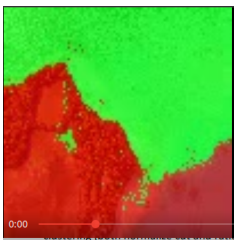
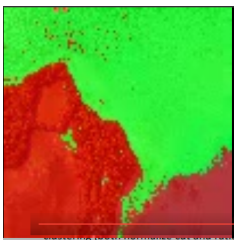
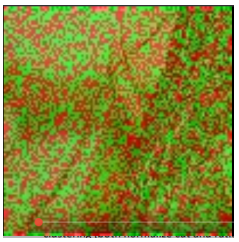
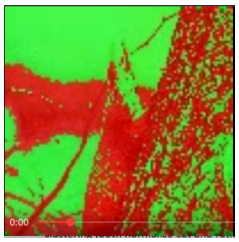
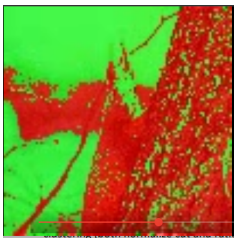
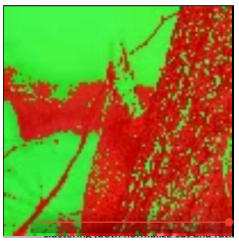
2. Experiments settings & results & discussion

I. Part1

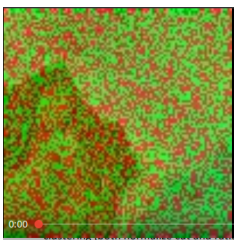
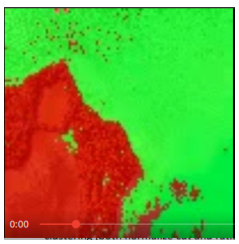
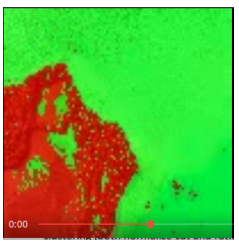
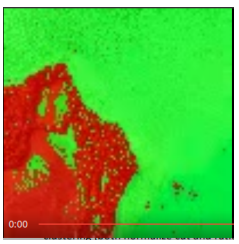
All methods shown below use the same parameters setting in this part.:
Spatial parameter: s = 1.0, Color parameter: c = 1.0, Maximum iterations: 50
Initialize method: random

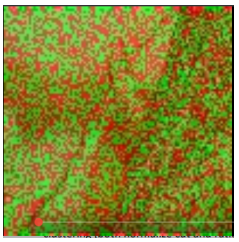
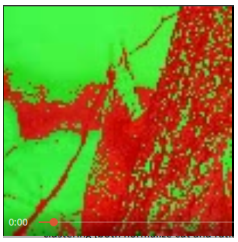
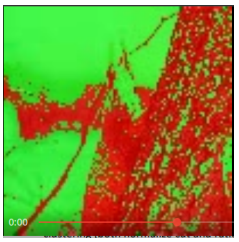
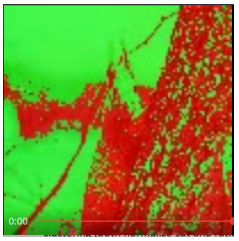
Below are the results from different methods. For K-means. it separates the light blue area from others. Even though the area is far away, the defined kernel still takes that area with a similar color into the same cluster.
For spectral with ratio cuts, the total number of pixels in each cluster is balanced in contrast to normalized cuts.

K-means:

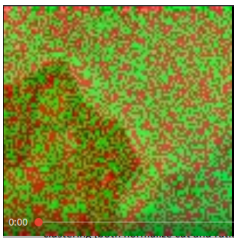
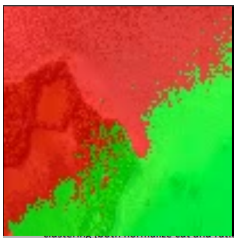
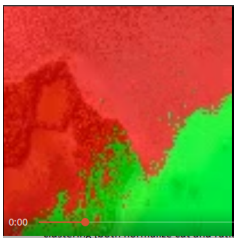
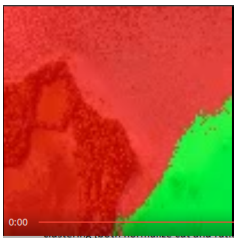
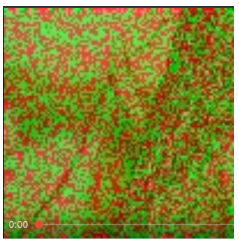
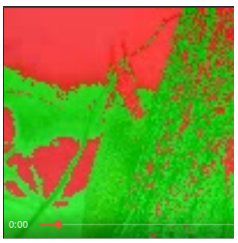
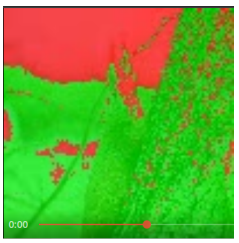
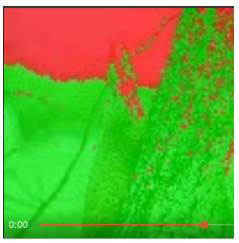
File name	Initialize	Sequence 1	Sequence t	Sequence End
image1_Kmeans_random_num_cluster:2_s:1.0_c:1.0.mp4				
image2_Kmeans_random_num_cluster:2_s:1.0_c:1.0.mp4				

Spectral Clustering- ratio cuts

File name	Initialize	Sequence 1	Sequence t	Sequence End
image1_Spectral_ratio_random_num_cluster:2_s:1.0_				

c:1.0.mp4				
image2_Spectral_ratio_random_num_cluster:2_s:1.0_c:1.0.mp4				

Spectral Clustering- normalized cuts

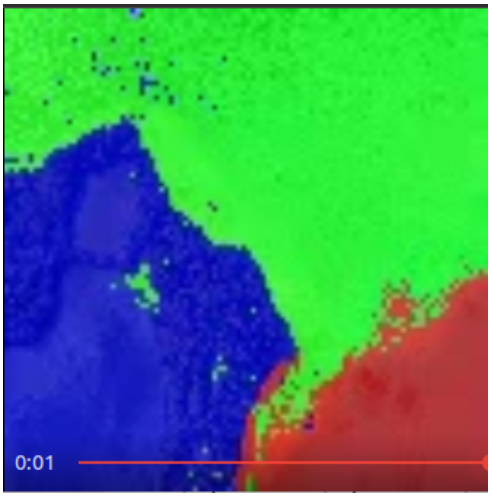
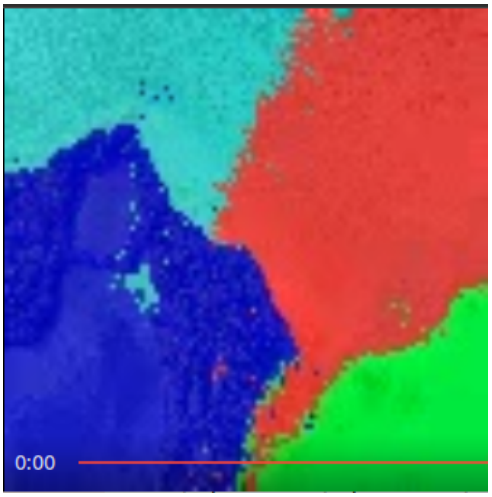
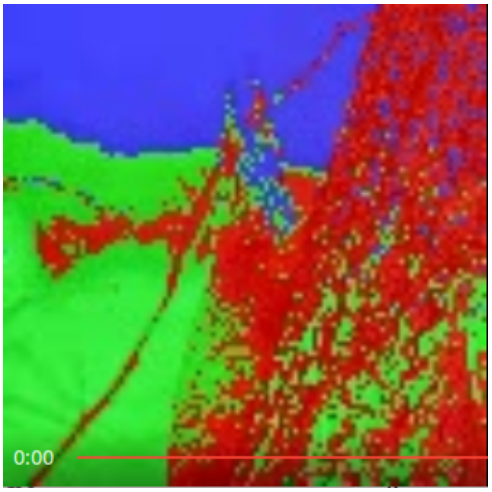
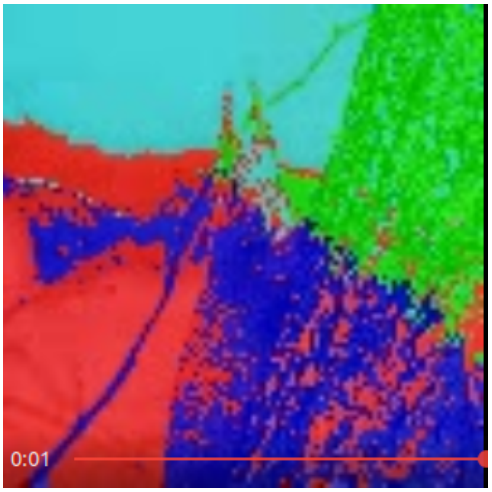
File name	Initialize	Sequence 1	Sequence t	Sequence End
image1_Spectral_normalized_random_num_cluster:2_s:1.0_c:1.0.mp4				
image2_Spectral_normalized_random_num_cluster:2_s:1.0_c:1.0.mp4				

II. Part2

setting: s=1.0, c=1.0, random initialized

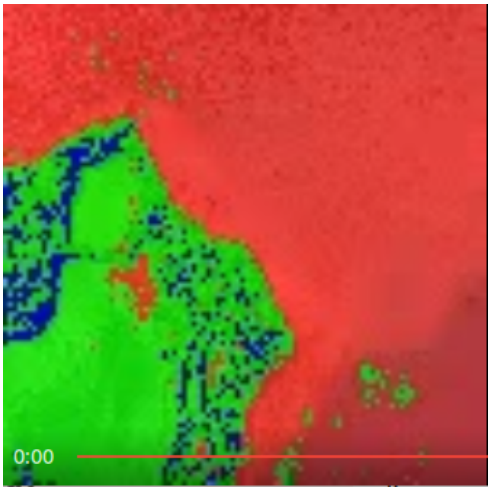
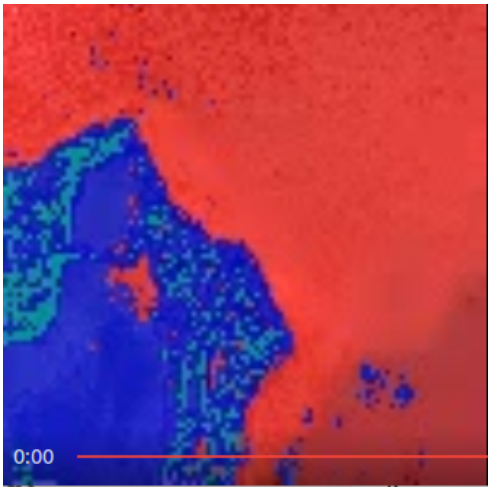
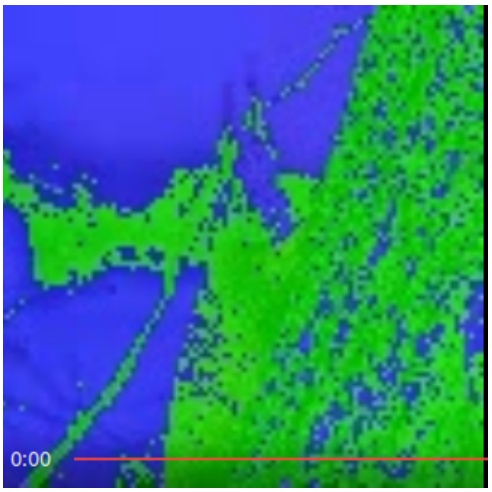
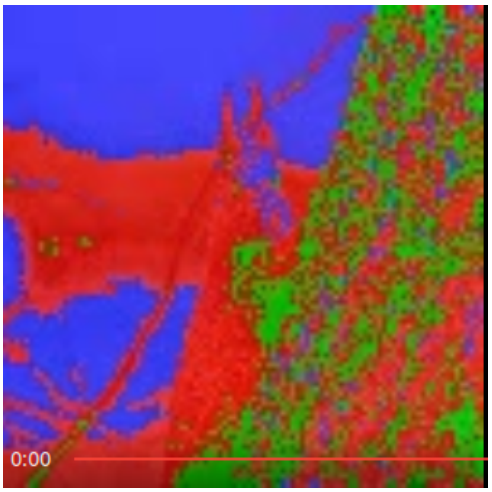
For Kmeans and Spectral-normalized cuts, they'll retrieve the corresponding number of clusters. But for Spectral-ratio cuts, it only retrieve n-1 cluster for n cluster. In my opinion, there is an unrepresentative eigenvector being selected. Lead to retrieve clusteres failed. It may be related to the parameters, s and c. But here I just use these s=1.0, c=1.0 and leave the results.

K-means

Cluter	3	4
File name	image1_Kmeans_random_num_cluster:3_s:1.0_c:1.0.mp4	image1_Kmeans_random_num_cluster:4_s:1.0_c:1.0.mp44
Results		
File name	image2_Kmeans_random_num_cluster:3_s:1.0_c:1.0.mp4	image2_Kmeans_random_num_cluster:4_s:1.0_c:1.0.mp4
Results		

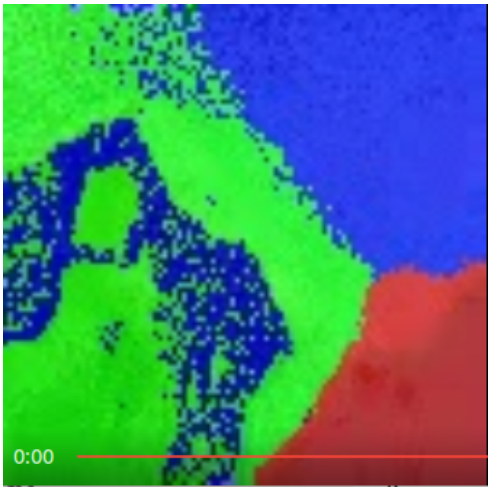
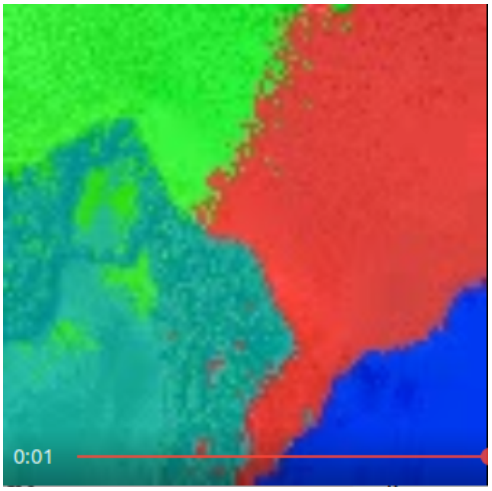
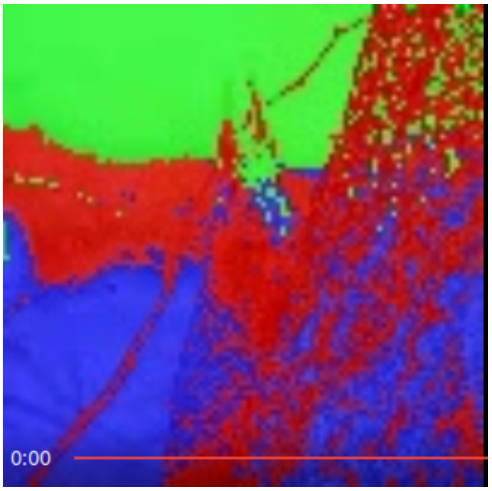
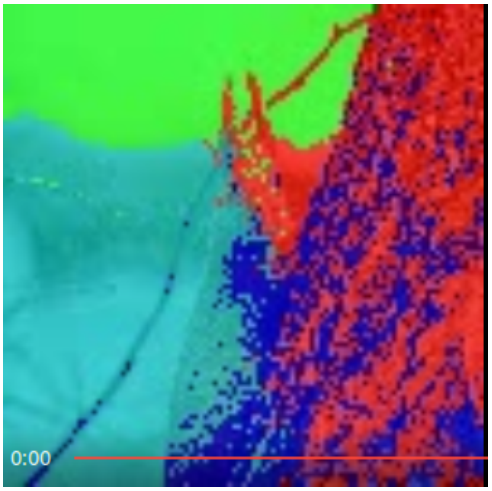
Spectral-ratio cuts

Cluster	3	4
File name	image1_Spectral_ratio_random_num_cluster:3_s:1.0_c:1.0.mp4	image1_Spectral_ratio_random_num_cluster:4_s:1.0_c:1.0.mp4

Results		
File name	image2_Spectral_ratio_random_num_cluster:3_s:1.0_c:1.0.mp4	image2_Spectral_ratio_random_num_cluster:4_s:1.0_c:1.0.mp4
Results		

Spectral-normalized cuts

Cluster	3	4
File name	image1_Spectral_normalized_random_num_cluster:3_s:1.0_c:1.0.mp4	image1_Spectral_normalized_random_num_cluster:4_s:1.0_c:1.0.mp4

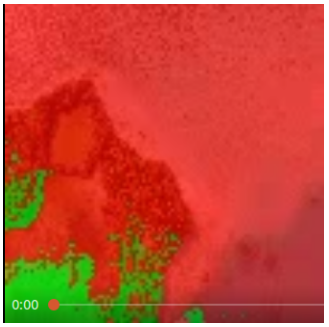
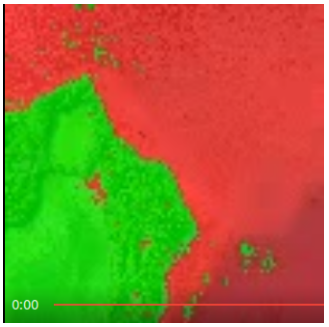
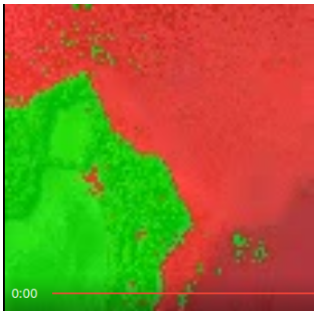
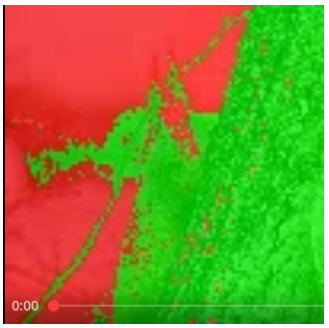
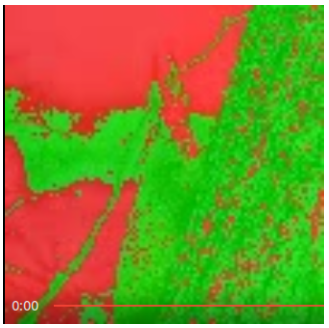
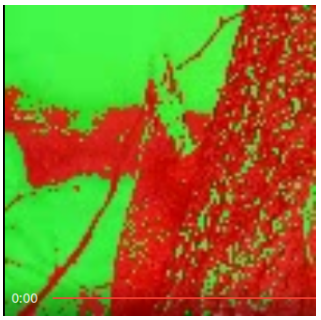
Results		
File name	image2_Spectral_normalized_random_num_cluster:3_s:1.0_c:1.0.mp4	image2_Spectral_normalized_random_num_cluster:4_s:1.0_c:1.0.mp4
Results		

III. Part3

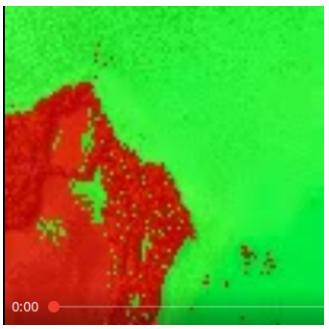
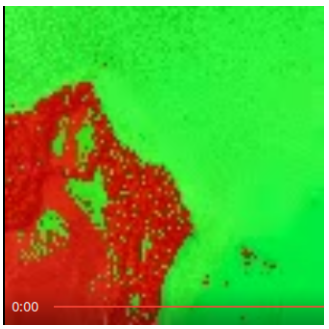
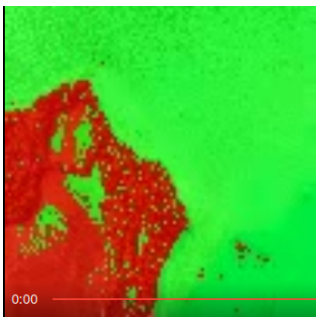
All methods: s = 1.0, c=1.0 cluster = 2
Below are different models with different initialized methods for clustering. Although the initialized state is different, they all have the same results as random initialization and fewer iteration times for convergence.

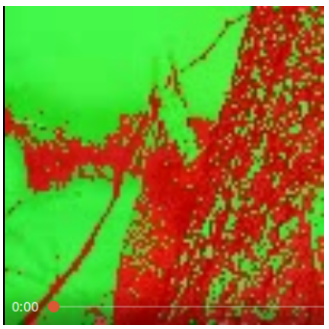
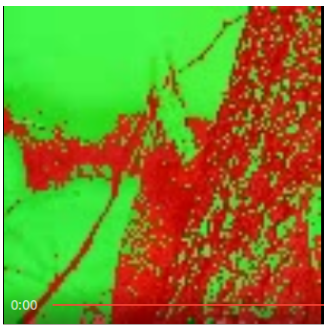
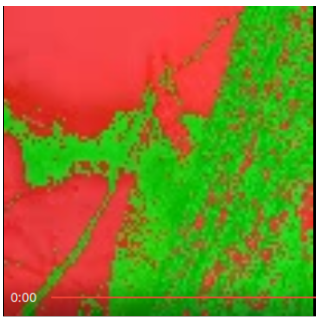
Kernel Kmeans- Kmeans++ initialization method v.s Randomly initialization

Category	Kmeans ++ initialize	Kmeas ++ results	Random results
File name	image1_Kmeans_kmeans++_num_cluster:2_s:1.0_c:1.0.mp4		image1_Kmeans_random_num_cluster:2_s:1.0_c:1.0.mp4

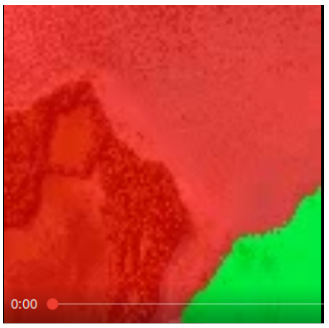
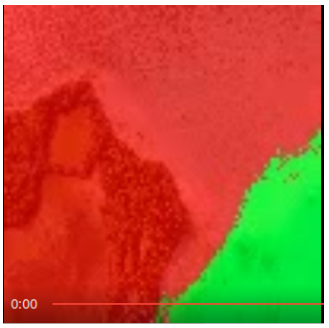
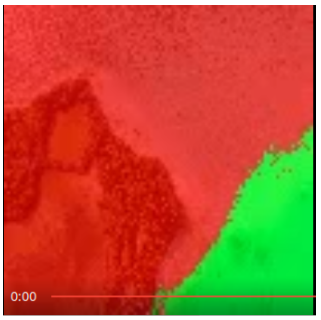
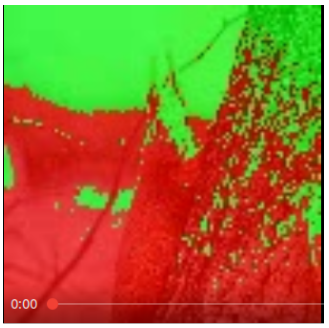
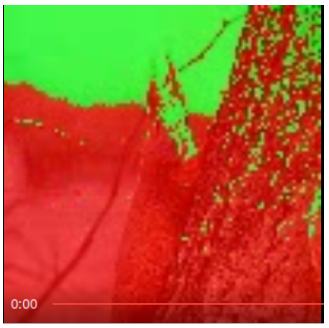
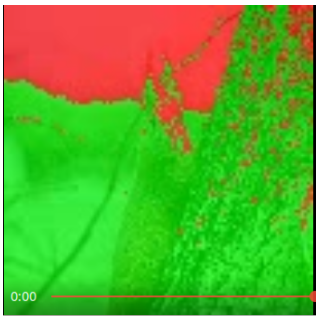
			
File name	image2_Kmeans_kmeans++_num_cluster:2_s:1.0_c:1.0.mp4		image2_Kmeans_random_num_cluster:2_s:1.0_c:1.0.mp4
			

Spectral Clustering-ratio cuts
Kmeans++ initialization method v.s Randomly initialization

Category	Kmeans++ initialize	Kmeans ++ results	Random results
File name	image1_Spectral_ratio_kmeans++_num_cluster:2_s:1.0_c:1.0.mp4		image1_Spectral_ratio_random_num_cluster:2_s:1.0_c:1.0.mp4
			
File name	image2_Spectral_ratio_kmeans++_num_cluster:2_s:1.0_c:1.0.mp4		image2_Spectral_ratio_random_num_cluster:2_s:1.0_c:1.0.mp4

Spectral Clustering- normalized cuts
Kmeans++ initialization method v.s Randomly initialization

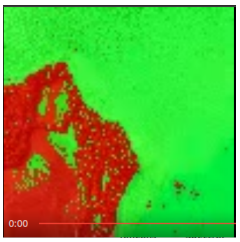
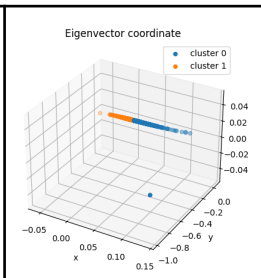
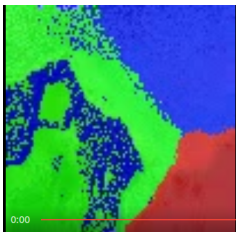
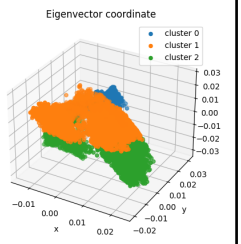
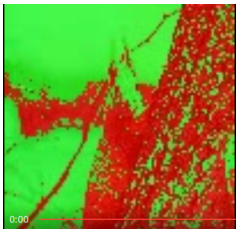
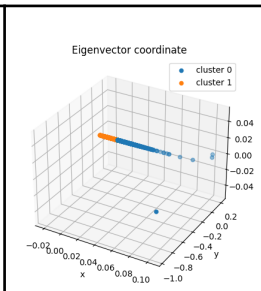
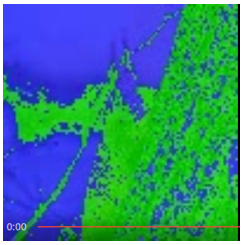
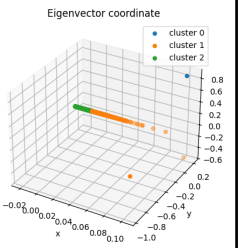
Category	Kmeans++ initialize	Kmeans ++ results	Random results
File name	image1_Spectral_normalized_kmeans++_num_cluster:2_s:1.0_c:1.0.mp4		image1_Spectral_normalized_random_num_cluster:2_s:1.0_c:1.0.mp4
			
File name	image2_Spectral_normalized_kmeans++_num_cluster:2_s:1.0_c:1.0.mp4		image2_Spectral_normalized_random_num_cluster:2_s:1.0_c:1.0.mp4
			

IV. Part4

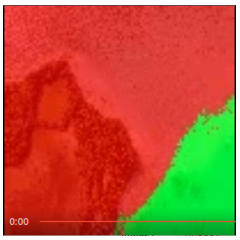
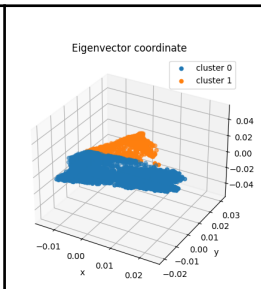
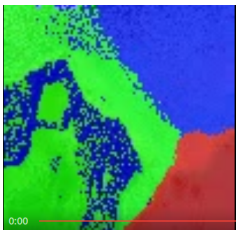
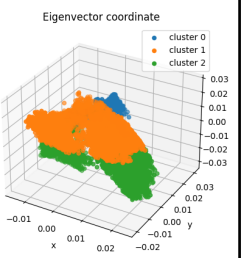
For this part, show the eigenvector coordinate in the 3D scatter points. Those in same clusters are near each other in eigenvector coordinates. For ratio cuts, because of failed

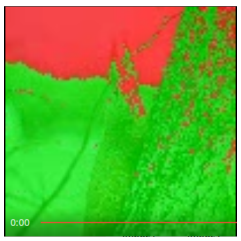
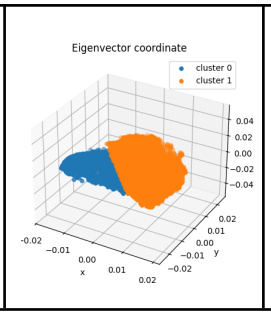
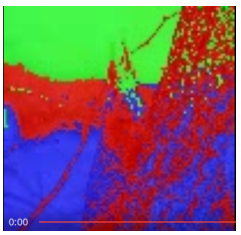
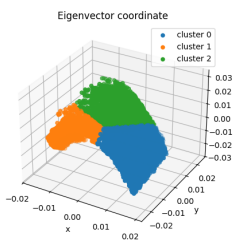
retrieving the corresponding number of clusters (referred in Part2), it only show 2 clusters for Spectral Clustering ratio cuts. Besides the clustering results, the drawn plot is also similar to the plot in 2 clusters plot.

Spetral Clustering-ratio cuts random initialization s=1.0, c=1.0

File name	image1_Spectral_normalized_rando m_num_cluster:2_s:1.0_c:1.0.mp4		image1_Spectral_normalized_rando m_num_cluster:3_s:1.0_c:1.0.mp4	
				
File name	image2_Spectral_ratio_random_num _cluster:2_s:1.0_c:1.0.mp4		image2_Spectral_ratio_random_num _cluster:3_s:1.0_c:1.0.mp4	
				

Spectral Clustering-normalized cuts s=1.0, c=1.0,

File name	image1_Spectral_normalized_rando m_num_cluster:2_s:1.0_c:1.0.mp4		image1_Spectral_normalized_rando m_num_cluster:3_s:1.0_c:1.0.mp4	
				
File name	image2_Spectral_normalized_rando m_num_cluster:2_s:1.0_c:1.0.mp4		image2_Spectral_normalized_rando m_num_cluster:3_s:1.0_c:1.0.mp4	

3. Observations and discussion

i. For Kernel K-means, it's suitable for all situations. It can always retrieve meaning for clusters e.g. ocean, light blue, ground, and so on.
For Spectral clustering - ratio cuts, it can't retrieve right number of clusters from image features. Maybe is because of the normalized features. But the clustering results show that except for the not-retrieved cluster, each cluster has a similar amount of pixels.
For Spectral clustering - normalized cuts, it can retrieve correct number of clusters. But it somehow divides the cluster with a hyper meaning e.g. texture of tree. It's not straight to understand the meaning of normalized cuts.

ii. Normally, Spectral clustering need more time to finish a trail because of calculating eigenvectors. Each trial needs to deal with 10000*10000 matrix, and calculation is a O(n^3) algorithm.
Consider initialization method, Kmeans++ can give a more convinced initial guess so reduce the iteration for clustering.

iii. I try to directly use clustering method from kernel K-means. But the value is unstable so in each iteration, pixels has the large probability of changing the cluster to another. Lead videos keep flicking. Based on the formula, this should not happened. But I can't find the reason. At the end, I rewrite the code and use the traditional K-means algorithm to conquer that issue.